Diploma Thesis

Interrupt-Related Covert Channels from an Attacker's Perspective

Richard Gay

December 1, 2008

RWTH Aachen University
Department of Computer Science

1st Assessor: Prof. Dr.-Ing. Heiko Mantel (TU Darmstadt)
2nd Assessor: Prof. Dr. Ir. Joost-Pieter Katoen (RWTH Aachen)

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 1. Dezember 2008

Richard Gay

Abstract

Interrupt-related covert channels allow two processes running on a single system to communicate with each other, circumventing the security mechanisms contained in standard operating systems. These channels belong to the class of timing channels and use the CPU as their shared resource. They transmit information by a sender process executing an operation that induces a hardware interrupt, and a receiver process detecting the interrupt. Both ends of the channel thus do not exploit that they share the CPU, but rather that the receiver and sender-initiated interrupts share the CPU. Consequently this channel is in general not prohibited by time-partitioning the CPU among all processes, as the literature about timing channels suggests.

This thesis contributes to the understanding of interrupt-related covert channels with the practical implementation of an exploit that is able to transmit information through such a channel. A patch for the Linux O(1) scheduler is developed which provides a reasonable environment for the exploit. Based on the exploit implementation, a theoretical model is constructed with whose help upper bounds for the implemented channel's bandwidth are computed. In addition, practical experiments are conducted on a standard computer, running the modified Linux kernel, to obtain also lower bounds for the bandwidth.

The implemented exploit contains free parameters, whose values are shown to have a significant influence on the performance of the channel. For configuring the exploit tailored to a given target system, a generic framework is constructed that captures the interdependencies between the system, the exploit and the expected resulting bandwidth. An application of the framework to a concrete computer system demonstrates its benefit for an attacker. Since noise is difficult to model precisely and often unknown to an attacker, techniques are presented that reduce its impact on the channel.

Acknowledgments

First and foremost, I would like to thank Henning Sudbrock for introducing me to the topic of interrupt-related covert channels and supporting me with his advice during the course of this work. I am as well indebted to Prof. Heiko Mantel, who gave me the opportunity of writing this thesis under his supervision and whose suggestions helped me not to lose sight of the "big picture."

I want to express my gratitude to my parents for their support during the work on this thesis (including paying the electricity bills . . .).

Finally I also want to thank all the unnamed persons that were and are involved in developing the free and open source software that I used for the work on this thesis. Particularly, without the availability of an open source operating system, an important part of this thesis would not have been possible.

Contents

1. Introduction **1**
- 1.1. Structure . . . 2
- 1.2. Conventions . . . 2
- 1.3. Communication Channels . . . 3
- 1.4. Information Theory . . . 5
- 1.5. The Testing Environment . . . 7

2. The Exploit **11**
- 2.1. Fixed Quantum Scheduling . . . 14
  - 2.1.1. Choice of Operating System & Scheduler . . . 16
  - 2.1.2. Scheduler Modifications . . . 17
- 2.2. Adaptation of the Exploit . . . 23
  - 2.2.1. Fixed-Quantum Compatibility . . . 23
  - 2.2.2. Higher Transmission Rates . . . 24
  - 2.2.3. Parameters . . . 25

3. Analysis **27**
- 3.1. Basic Channel Model . . . 27
- 3.2. Transmission Models . . . 29
  - 3.2.1. Lossless Transmission . . . 30
  - 3.2.2. Lossy Transmission . . . 30
- 3.3. Adjusting Exploit Parameters . . . 36
  - 3.3.1. A Generic Framework . . . 40
  - 3.3.2. A Concrete Instance . . . 47
  - 3.3.3. Applicability & Limitations . . . 53
- 3.4. False Positives . . . 54
  - 3.4.1. Classification of Noise . . . 55
  - 3.4.2. Incorporating Interruption Times . . . 56
  - 3.4.3. Generating Multiple Interrupts . . . 58

4. Conclusion **61**
- 4.1. Novel Countermeasures . . . 61
- 4.2. Surprises . . . 62
- 4.3. Related Work . . . 63
- 4.4. Summary . . . 64
- 4.5. Outlook . . . 65

A. Proofs **67**
A.1. Lossy Noiseless Transmission . . . 67
A.2. Thresholds . . . 69
A.3. Hamming Bit Error Rate . . . 71
A.4. Generic Framework . . . 72

B. Source Code **75**
B.1. Adapted Exploit . . . 75
B.1.1. Sender Implementation . . . 76
B.1.2. Receiver Implementation . . . 83
B.1.3. Original Exploit . . . 90
B.2. Modified FreeBSD Scheduler . . . 91

Bibliography **95**

Index **99**

1. Introduction

At the time of writing this thesis, even low-cost personal computers have the storage capacity for several hundred gigabytes of data. For a convenient handling of such large amounts of data, they are equipped with UDMA or Serial ATA hard disks, several USB 2.0 ports and Fast or even Gigabit Ethernet connectors, where each of the underlying protocols supports for transmission rates from a hundred million to more than a billion of bits per second.[1] Furthermore, the majority of German households has broadband access to the Internet and thereby usually more than a million of bits per second of "downstream" transfer speed [Smi07].

The demand for high transmission rates to quickly exchange large amounts of data nevertheless does not make the analysis of communication channels with low capacities become obsolete. As soon as, for example, encryption is used to ensure confidentiality and/or integrity, rather small amounts of data are of particular importance: Passwords and private keys. Usual passwords are short and not randomly chosen from the available character set (e.g. ASCII), so they probably often do not exceed 100 bits of information. Private keys are longer, but currently still in the range of few kilobits in size. Thus, even using a comparatively very slow channel of 1 bit per second, the transmission of such a password or private key would take less than few hours. For an attacker hoping to decrypt confidential data or forge the authenticity of data, these few hours could be worth the effort. Consequently, if an attacker somehow accomplished to infiltrate the target system with a program – the *sender* – that can access the secret data, then even a slow channel might suffice for this sender to transmit the data to the attacker.

In [MS07] and [MS08], Mantel and Sudbrock describe and analyze the class of *interrupt-related covert channels*, which will later be introduced in detail. These channels are established between two processes on a single computer and have the property of circumventing the security measures found in "normal" computers and operating systems, which is why they constitute a potential threat to secrets stored on such a system. With their model for analyzing the bandwidth of such channels, the articles adopt the point of view of a potential victim, who does not know what an attacker may be able to practically realize, and therefore assumes the worst case: A powerful attacker who is, if at all, subject to only minimal restrictions. Accordingly, the outcomes of the model – up to 50 bit/s in one example of [MS08] – constitute upper bounds on the transmission rate that might not be reachable by a realistic attacker.

This thesis therefore assumes an *attacker's* perspective for analyzing interrupt-related covert channels: Just like an attacker finally has to commit to a concrete implementation of a channel, also this thesis chooses a (rather simple) transmission model as the basis for an implementation and its theoretical and practical analysis. The evaluation of the implementation will not only show that interrupt-related covert channels are practically

[1] UDMA133: 133 MB/s, SATA/150: 1.5 Gbit/s, USB 2.0: 480 Mbit/s, Fast Ethernet: 100 Mbit/s, Gigabit Ethernet: 1 Gbit/s.

possible but can also give lower bounds on the reachable bandwidth and allows for analyzing the influence of the system environment on these channels.

What this thesis does explicitly *not* cover is the preparatory work that an attacker has to accomplish to gain access to secret data, like for instance with a Trojan horse. Analogously it does also *not* talk about how the receiving process could deliver the secret data to the actual person interested in it. In particular, this thesis and all depicted source code does *not* intend to represent a guide, motivation or template for unlawfully breaking into computer systems and stealing confidential data.

1.1. Structure

The content of this thesis is split into two main parts. The first, found in Chapter 2, covers the implementational aspects of an interrupt-related covert channel. It includes not only the actual exploit realizing such a channel but also a modification of the Linux operating system that establishes a reasonable environment for this exploit. After that, Chapter 3 examines the performance of the chosen exploit implementation in a theoretical model and with practical experiments. The chapter also provides a framework for finding exploit configurations tailored to a given target system, and shows techniques for reducing the impact of noise on the implemented channel.

The remainder of this first chapter mainly provides the reader with the background knowledge required to understand interrupt-related covert channels and how communication channels can be formally analyzed. At the end of this chapter also the setting is introduced which has been used for the practical evaluation of the exploit.

The fourth chapter concludes the thesis. It presents findings that were discovered during the work but did neither fit into the other chapters nor deserve their own chapters. Finally, it summarizes the results obtained in the preceding chapters and suggests directions for further research on the topic.

In order to not pollute the main part of the thesis with long calculations and large fragments of source code, these have been swapped out into the appendix. Appendix A contains the formal proofs for all (few) theorems presented in the thesis. The thesis ends with Appendix B showing the relevant parts of the actual exploit implementation together with remarks about how the different pieces work together. This chapter also comprises code for a modification of the FreeBSD kernel, which has been developed but was in the end not used for testing the exploit.

1.2. Conventions

Source code. All source code presented in this thesis is either written in C, C++ or a C++-like pseudo code, depending on whether Linux or FreeBSD kernel code, exploit source code, or demonstration code is concerned. Reserved code words are printed in boldface (e.g. **struct**), comments in italics (e.g. */* comment */*). To distinguish between function names and variable identifiers when they are referenced in the text, the former are suffixed by an empty set of parentheses (e.g. `sendto()`) while the latter are not (e.g. `skew`). Function-like macros, i.e. macros that take parameters, are treated like regular functions.

Text formatting. Besides the source code, also some other special items got a special formatting. Program and file names are printed in typewriter font (e.g. `foo`), as well as also shell code, which is additionally prefixed with a dollar sign (`$`).

The symbols of the binary alphabet are represented by the boldface digits "**1**" and "**0**" so that they can be distinguished from ordinal numbers. They could as well have been called "true" and "false".

Math notation. The natural numbers are represented by the symbol $\mathbb{N}$ and are defined to contain all nonnegative integers, i.e. including zero: $\mathbb{N} := \{0, 1, 2, \ldots\}$. This choice is not meant as a statement about whether 0 should generally be treated as a natural number or not, but is just suited better for the outcomes of counting processes.

To avoid confusions between the names of scheduling algorithms and the big O notation, the former are written with a normal font capital "O" (as in $\mathrm{O}(n)$) while the latter is referred to by a calligraphic "$\mathcal{O}$" (as in $\mathcal{O}(n)$).

Finally, wherever the basis of a logarithm is omitted, a binary logarithm is implicitly assumed: $\log(x) := \log_2(x)$.

1.3. Communication Channels

The very beginning of this chapter already mentions channels for the communication between different pieces of hardware inside a computer, between a computer and peripheral devices, and between computers or other networking devices. Additional channels exist for *interprocess communication* (IPC)[Wol05, Chapter 8], i.e. the communication between running instances of programs on a single computer system. Two (or more) concurrently running processes could use a *pipe* for unidirectional communication, as it is frequently done to redirect the output of one program to the input of another program. Another possibility would be *shared memory*, which can be used bidirectionally but requires for the processes to take care of synchronization. The list could be extended by further POSIX IPC mechanisms, like *sockets* or *message queues*, but also by operating system dependent ones as *DDE*, just to name one of Microsoft's Windows. All these channels are explicitly enabled by the operating system to be used for communication and, thus, belong to the group of *overt channels*. As a consequence, they are also under the control of the operating system and its security policy. Unwanted flow of information through such an overt channel can thus be prohibited, e.g., by rejecting unprivileged processes.

As the definition of overt channels suggests, there are also channels which are *not* intended to be used for communication. These are called *covert channels* and have first been mentioned by Lampson in [Lam73]. They can be further distinguished into *storage* and *timing* channels (compare [McH95]), depending on whether a stored value or the perception of time is used to transmit information. An example for the former category could be the existence of a file, whose entry in a directory list would be the stored value. To send a value, the sending instance could encode a **1** by creating a certain file, a **0** by deleting it again. If the receiver has read access to the directory, it could then determine the existence of the file and decode it as a **1** or **0** respectively. Note that even though it is not intended that processes communicate this way, the operating system has a chance to eliminate any such channel through the regular security primitives, simply by not giving the receiver read

access to any directory that the sender may create a file in. Unfortunately, this does not hold true for every kind of covert channel, as the remainder of this section will show.

Timing channels utilize the receiver's perception of time in order to transmit information, as denoted before. If, for instance, the sending and the receiving process are run on the same CPU, like on a uniprocessor system, then the sender could encode a **1** or a **0** by using more or, respectively, less of the computing time at its disposal. This then may have an influence on whether the receiving process is scheduled for execution again later or earlier, which it can detect by measuring its own execution times to finally deduct the sent bit. In [Hus78], this channel is called a *quantum-time channel*, and it will play an important role in Section 2.1. Another, yet similar timing channel is the *interquantum-time channel*, which encodes the information not by *how long* the sender process runs but by *when* it runs. Since the execution times of processes and their ordering is not subject to security mechanisms in common operating systems, this kind of channel cannot easily be restricted by setting certain access rights or the like.

Interrupt-related covert channels (IRCCs, see [MS07]) also belong to the category of timing channels, but here the sender uses an indirection for influencing the execution times of the receiver process: *Interrupts* [Tan01, Section 5.2]. In contrast to *polling*, where the operating system repeatedly asks hardware for status changes or new data, interrupts are used by hardware to actively inform the operating system. If the execution of code itself raises the interrupt, for example because an application wanted to execute an invalid or forbidden instruction, then it is a *synchronous* one, otherwise called *asynchronous*. In the following, the term *interrupt* will refer only to the latter kind, as the former always results in interrupts of their originator and are therefore of no further interest for IRCCs. An interrupt itself does not contain any specific information except for a *request number* referring to the respective device that issued the interrupt. On its occurrence, the processor stops its regular execution of code and switches to a specific *interrupt handler*, which has previously been installed by the operating system and in most cases essentially consists of device driver code. After completing the handler, the execution can return to the point where it had stopped when the interruption occurred.[2]

Now the basic idea behind IRCCs is that a sender may be able to perform some operation that causes an interrupt (or multiple) to occur later, when hopefully the receiver is executing. The receiver in turn may be able to detect the occurrences of interrupts and may then extract information, e.g., from how many of them occurred at what times and with which durations.

Timer chips generate interrupts with a fixed or, as usual, programmable frequency [Tan01, Section 5.5], in order to periodically awaken the operating system and simultaneously take away the control from a user process, which may have been executing for too long. In general, a user process cannot cause additional or, in particular, fewer timer interrupts, which it could exploit for establishing an IRCC with a receiver process. The opposite holds true for hard disks (HDs), which use interrupts to indicate the completion of a read from or write to the disk. Consequently, by calling an appropriate system function for reading from or writing to a disk file, an ordinary user process may have an influence on the generation of hard disk interrupts.[3] Furthermore, since disk operations are comparatively slow, the

[2] Depending on the type of the interrupt, the operating system may also decide to schedule another process than the one that was interrupted [BC05, Chapter 4].

[3] Throughout this thesis, "generating", "initiating" and "triggering" interrupts will be used synonymously to express that a process performs some operation that may finally result in the occurrence of a hardware interrupt.

respective interrupts occur with a certain delay after the causing system call. If the delay is long enough to allow for a process switch from the sender to the receiver, then the latter is interrupted and may, for example, decode the occurrence of the interruption as a **1** or the absence of it as a **0**.

In modern operating systems, direct output to or input from hard disks is avoided in favor of caches whenever possible. Thus, when repeatedly reading the same block from a file, a sender process very likely will cause an interrupt only with the first operation, as afterwards this block is present in the disk cache and read from there. The circumvention of block caches can easily become cumbersome, as the sender has to find blocks which are not yet in a cache with high probability. Therefore, it may have to consider read-ahead features of the respective file system (compare e.g. [BC05, Chapter 16]), but it likely has no knowledge of the actual block ordering on the disk.

Another source of influenceable interrupts, which is present in most of today's computers, is the network interface card (NIC). The NIC generates an interrupt after a packet has arrived at the cable, but also when packets have been sent out, which is to inform the operating system, for example, that further packets may be sent through the device. Hence, by sending packets over the network, even if to a nonexisting (Ethernet) address, the sender can cause interrupts whose occurrences have a chance to carry information to a receiver process. This still holds true when repeatedly sending the same packets, as network stacks do not have a counterpart to the block caches. Due to the implied simplicity of triggering NIC interrupts, they are chosen to be used for establishing IRCCs throughout the practical experiments of this thesis. The use of NIC interrupts for the covert channel may be questionable, as sending the information in a packet over the network directly would be much simpler and faster, but one might well imagine a situation where the sender cannot reach any potential receiver process over a network or via a local socket.

Concerning the implementation of the exploit, the dependency on this specific type of interrupt source has been reduced to a minimum, so that a substitution with another source would not require changes to the core of the channel implementation.

1.4. Information Theory

Besides the practical realization of an interrupt-based covert channel, this thesis is also concerned with modeling and analyzing the implemented channel's properties. For this purpose, the channel will later be considered to be a *discrete memoryless channel*, according to the information-theoretic framework of [CT06], which is also based on in [MS07] for the IRCC analysis.

From a theoretical point of view, a channel can be represented by triple of possible channel inputs, possible channel outputs and some transformation rule indicating how inputs translate to outputs when passing through the channel. Due to the possibility of distortions (noise), this transformation is not necessarily deterministic, but rather dictated by probabilities. Throughout this thesis, the *input alphabet*, i.e. the set of channel inputs, will be referred to as I, the *output alphabet* as O. For a *discrete* channel, both alphabets must be discrete, thus either finite or countably infinite.[4] As a convention, concrete input symbols will be named x or x_i for some i, output symbols y or y_i.

[4] In its definition of a discrete channel, [CT06] is vague about the meaning of the "discrete," but in the course of the book supports the interpretation given here.

A channel is *memoryless* if the transformation from inputs into outputs only depends on the respective current input symbol and, hence, is particularly independent of the preceding transmission history. Formally the transformation can then be expressed in form of a *transition matrix* $P = (p(y|x))_{x \in I, y \in O}$, where $p(y|x)$ is the *conditional probability* of obtaining output $y \in O$ if the input was $x \in I$. Together, the mentioned triple identifying a channel is (I, O, P). In the following, unless otherwise mentioned, it is assumed that all covered communication channels are memoryless.

Concerning the analysis of a covert channel, its transmission rate, i.e. the amount of information transmissible per time unit, is of particular interest. It is comprised of the amount of information that can be transferred with every *single use* of the channel and the rate with which the channel can be used. Since the latter is not included in the channel model, only the former, also referred to as the *channel capacity*, is further examined here. Therefore firstly two examples for the transition matrix P are considered:

- If a channel's output is completely random, no matter what the input symbols are, then this channel intuitively does not transport any information.
- If the channel transforms each input symbol into its own, distinct output symbol (e.g. $I = O$ and the output is always identical to the input), then no information is lost on the way through the channel and the output contains exactly as much information as the input.

The first example depicts a simple but rather useless channel only intended to demonstrate how the capacity of a channel can become zero. The second example is similarly simple to understand, but its resulting capacity might be less obvious.

Since the amount of information of a single, fixed input symbol is meaningless, input has to be considered a (discrete) random variable, which will be named X. Its probability distribution will be referred to as $p(X)$; the concrete probability of the input assuming a certain value $x_0 \in I$ as $p(X = x_0)$, or just $p(x_0)$ if the random variable is clear from the context. Analogously the output constitutes the random variable Y with distribution $p(Y)$.

The amount of information contained in a random variable like X is captured by the *entropy*, being defined as

$$H(X) = -\sum_{x \in I} p(x) \cdot \log_2(p(x)).$$

The logarithm to the base 2 makes the unit of the entropy be bits. For example, if alphabet I is finite and X assumes all values of I with the same likelihood, then $H(X) = \log_2 |I|$, i.e. the contained amount of information corresponds to the average number of bits needed to represent an element of I. On the contrary, if X always assumes the same value $x_0 \in I$, then the random variable X does not deliver any information (everything is known already in advance) and, consequently, $H(X) = 0$ can be computed for this case. Hence, entropy can also be considered a measure of *uncertainty* (about the future values of the respective random variable).

The intention of a communication channel is to reduce the uncertainty about the input, ideally completely revealing it. Thus, the channel capacity can be concretized as to what extent the uncertainty about the input is reduced by learning the output of the channel. This reduction can be expressed as the difference of the initial uncertainty $H(X)$ about X and the remaining uncertainty $H(X|Y)$ after learning Y, where

$$H(X|Y) = -\sum_{\substack{x \in I, \\ y \in O}} p(y) \cdot p(x|y) \log p(x|y)$$

is the conditional entropy of X, given Y. It can be shown that the reduction of uncertainty about X, $H(X) - H(X|Y)$, is equal to the reduction of uncertainty about Y if X is learned, $H(Y) - H(Y|X)$. Thus, the amount of information that one random variable reveals about another one is identical to the amount of information that this other variable reveals about the former one. This amount of information is consequently referred to as the *mutual information* $I(X;Y)$ (between random variables X and Y):

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) = \sum_{x,y} p(x,y) \cdot \log \frac{p(x,y)}{p(x)p(y)}.$$

For the first of the above two examples, one can compute that $H(Y) = H(Y|X) = 0$ and therefore also $I(X;Y) = 0$: There is no mutual information between input and output, as intuition suggests. In the second example it turns out that $H(X|Y) = 0$ and, thus, $I(X;Y) = H(X)$: All information contained in the input is also in the output of the channel. Unfortunately, the mutual information of the latter example cannot – without additional knowledge about $H(X)$ – be further simplified to a constant but depends on how "informative" the input is.

The *capacity* of a discrete, memoryless channel (I, O, P) is a measure independent of peculiarities of the input distribution and is therefore defined as the maximum mutual information of input and output, over all possible input distributions:

$$Cap(I, O, P) := \max_{p(X)} I(X;Y).$$

If the channel is clear from the context then the capacity may in the following be referred to just as "*Cap*".

While in the first above example $Cap(I, O, P) = 0$ follows immediately, the capacity of the second example's channel equals the maximum possible information of the input: $\log |I|$ (as before, in bits).

1.5. The Testing Environment

The exploit of IRCCs developed in this thesis will be evaluated in practice by conducting several experiments. All these experiments are conducted on the same computer, whose configuration will follow below. This configuration is to be seen as *one* example for a possible target system and does explicitly not raise the claim of being representative for *all* possible target systems of the exploit. In particular, the chosen hardware is not intended to be more susceptible to or resistant against interrupt-related covert channels than other hardware might be.

The *basic configuration* – comprising hardware, operating system and software – for most experiments is the following:

- AMD Athlon XP 1600+ CPU. This CPU does not support frequency scaling and, thus, operates at a constant frequency of about 1400 MHz.[5] It is the only CPU of the system.

[5]Measurings for later experiments returned a more accurate value of 1407.258871 MHz, relative to the time of the clock chip.

- AMD 761 north bridge and VIA 686B south bridge.
- 512 MB of DDR-RAM.
- Realtek 10 Mbit/s NIC with RTL8029(AS) controller, connected to a Fast Ethernet switch without any other machines attached. The *interrupt request number* (IRQ) of this NIC is not shared with other hardware devices.

  The controller is NE2000 compatible (see [Rea97]) and is enabled via the Linux PCI NE2000 driver. At system startup, Linux registers a function called `__ei_interrupt ()` from `drivers/net/lib8390.c` as the interrupt handler for this NIC, where in turn the transmit interrupts are delegated to `ei_tx_intr ()`. The code belonging to the latter function shows that transmit interrupts are only used as a notification for potentially sending further packets.
- Realtek 100 Mbit/s NIC with RTL8139 controller, only as a reference. During all experiments which do not explicitly involve this controller, it has not been connected to any other network device. Its IRQ is not shared.
- Seagate 40 GB UDMA 100 hard disk.
- GeForce2 MX VGA controller.
- PS/2 keyboard.
- Gentoo Linux, with a modified vanilla 2.6.22.9 kernel (see Section 2.1.2). The kernel configuration is minimal and all drivers are compiled into the kernel (i.e. there are no modules).
- Kernel timer frequency is set to 100 Hz (the lowest possible setting offered).
- `pdflush`'s writeback interval was changed from 5 s to 127 s (a prime to make the period for “collisions” with other timers large):

  ```
  $ echo 12700 > /proc/sys/vm/dirty_writeback_centisecs
  ```

  `pdflush` is a kernel task that writes cached data, which is kept in Linux's page cache, to the hard disk. The reason for the change of the writeback interval and for the choice of a prime will be given in Section 3.2.2 (in particular page 31).
- The commit interval for the root ext3 partition (the only mounted disk filesystem) was changed to 347 s (again prime).
- Toolchain used for the compilation of the exploit: `gcc` 4.1.2, `binutils` 2.18.
- Init program: It was discovered that the SystemV init process, `sysvinit`, has an unconfigurable 5 s timeout, after which some routine checks are performed. Similar timeouts seem to be included also in replacement programs for the default `init` process, like `initng`, `einit` or `minit`. Unless otherwise mentioned, `sysvinit` (version 2.86) will be used.

  For the basic configuration (and *only* for the basic configuration), the timeout in `sysvinit` was disabled without modifying the sources with a little hack:

```
$ umask 177 && rm /dev/initctl && touch /dev/initctl
```

This stops `sysvinit` from querying the FIFO `/dev/initctl` with the mentioned timeout and makes it `pause()` instead. The original functionality can finally be restored by

```
$ rm /dev/initctl && kill -s HUP 1
```

– Other processes: `udev` (version 119), `login` (from `shadow` 4.0.18.2), `bash` (version 3.2).

It is apparent that, compared to usual computer systems, the above setting is kept almost minimalistic. Concerning the hardware, for instance multiple CPUs or hard disks, attached USB devices or connected large networks are missing. Common software like a system logger, a daemon for scheduled commands (`cron`) and, depending on the purpose of the system, a desktop environment or network servers have also not been included. While the resulting setting is beneficial for obtaining results about the behavior of an IRCC exploit under minimal disturbances, it is probably unrealistic to have such a system as the target of the exploit. Therefore, the basic setting is supplemented by some additional hardware and software to form the *extended configuration*:

– PS/2 mouse.

– An additional ext3 partition with the default commit interval of 5 s.

– `syslog-ng` (version 2.0.9).

– KDE (version 3.5.9) in a default setup with `Xorg` (version 7.2), started via `xinit`.

Wherever the extended configuration was used to conduct an experiment, this will explicitly be mentioned. During all experiments, the testing environment was not used for doing other work in order to reduce unreproducible effects to a minimum.

2. The Exploit

This thesis is based upon an existing exploit, which first showed that interrupt-related covert channels are not only theoretically but also practically possible. Since the exploit is the result of a work as a student researcher which has not been published, this chapter first introduces its basic concepts, supported by the most relevant fragments of code. The remainder of the chapter then is a novel contribution and deals with creating an environment where utilizing IRCCs is reasonable and finally with adapting the exploit to the changed conditions.

In its original form, the exploit consisted of a set of command line programs for Linux, including the actual sender and receiver as well as other measurement and evaluation utilities. Later it was rewritten and extended by a graphical user interface (GUI) for demonstrating purposes. The implemented communication is unidirectional, i.e. only the sender transmits data through the channel and there is no feedback, even through potential other channels.[1]

The implemented IRCC requires that sender and receiver process are executed concurrently on the same computer. Moreover, the interrupts triggered by the sender process have to occur at the processor which the receiver is executed on. While on a uniprocessor system this is trivially the case, the exploit does not make any provision for achieving it on a multiprocessor system.

Basic concept. The exploit allows for the transmission of a finite bit sequence. Each bit is transmitted during its own *bit interval*. All bit intervals of a transmission are of the same *length*, upon which sender and receiver have to agree beforehand. The starting time of the transmission finally fixes the positions of all bit intervals and must therefore be the same for sender and receiver. All bits are from this time on transmitted consecutively (in the same ordering as in the bit sequence).

To send a **1** over the channel, the sender repeats the two steps of executing an interrupt-initiating system function, in this case `sendto()` for sending a UDP packet,[2] and immediately afterwards yielding the CPU. After each iteration, the receiver therefore has a chance of being scheduled and might then detect the interrupt. Finally, a **0** is represented by the sender not calling the mentioned function at all and, thus, not deliberately causing any interrupts.

For each bit, the receiver steadily tries to detect interruptions of its execution and measures their *durations*, counting only those whose duration is within a certain range. Ideally this range represents the characteristic durations of interrupts caused by the sender or, more precisely, by the hardware device that has been chosen by the sender. After a bit interval has elapsed, the counted number is the output of the channel for this bit. This output in

[1] Under the circumstance of an asymmetric security policy, the receiver may be allowed to, e.g., write to a file which the sender can read from, but not vice versa.

[2] UDP packets have the advantage that they do not require for a complex connection establishment like TCP does *and* are allowed to be sent without superuser privileges, in contrast to e.g. ICMP pings under Linux.

```
void send_bit(bool bit) {
    wait_bit_sync();
    if (bit == 1)
        run_for(bit_duration) {
            short_busy_wait(); // optional
            generate_interrupt();
            sched_yield();
        }
    else
        wait_for(bit_duration);
}
```

Listing 2.1. Original code to send a bit over an IRCC (pseudo code).

general does not equal the input (e.g. during longer bit intervals, more than one interrupt could be generated for input **1**).

This basic concept of the exploit is retained through the rest of this thesis due to its simplicity. Other channel encodings, like those proposed in [MS07, MS08], would have been possible as well but have deliberately not been chosen since (a) their implementational complexity would have been higher, (b) their susceptibility to external influences is expected to be higher, and (c) it would have been harder to distinguish between deficiencies resulting from the implementation and those of an external nature.

Sender implementation. The code which is executed by the sender for each bit is rather simple and shown in Listing 2.1. At the very beginning, the function waits for the bit's time interval to start. This could be and has been realized by sleeping until the Unix timestamp (the number of seconds since midnight, January 1, 1970) assumes an integral multiple of the bit interval length. The following then depends on whether a **0** or a **1** is to be sent. In the former case, the sender simply waits the bit's full interval and, thus, particularly does not actively cause any interrupts. More interesting is the latter case. The `run_for()` construct was defined as a C macro (see Section B.1.3) that does what its name indicates: Repeating the execution of the following code block for the given `bit_duration` with the help of an interval timer. The repeated block consists of three lines which deserve their own explanation:

(a) The first line contains an optional, short extra delay represented by `short_busy_wait()`. An explanation at the end of this section will make the intention of this seemingly senseless code clear.

(b) The second line consists of the interrupt-generating function, here `generate_interrupt()`, itself. All potentially required parameters for the interrupt-generation are omitted here, as they depend on which hardware device has been chosen as the source for the interrupts.

(c) The last line sets the sender asleep with a call to `sched_yield()`. This has the intention that while the sender is sleeping, the operating system can schedule another process, which ideally is the receiver.

```
timing_t measure_duration(int iterations) {
    timing_t t = gettime();
    double v = 1.0;
    for (int i=iterations; i>0; --i) v *= 1.5;
    return gettime() - t;
}

bool in_relevant_range(timing_t dt) {
    return (dt >= base+dmin && dt <= base+dmax);
}

void receive_bit(int &interrupts) {
    wait_bit_sync();
    run_for(bit_duration) {
        dt = measure_duration(iterations);
        if (in_relevant_range(dt))
            ++interrupts;
    }
}
```

Listing 2.2. Original code to receive a bit over an IRCC (pseudo code).

Receiver implementation. The code for the reception of a single bit is given in Listing 2.2. The structure of the receive_bit () function resembles the code for sending a bit: The waiting for the begin of the transmission is the same as for the sender, as well as the code repetition for the bit's duration. Interruptions are measured and counted only if their durations fall into the range [dmin, dmax]. This range depends on how long the operating system takes to handle the interrupt signal, which in turn relies on (a) the operating system and its version, (b) the used driver of the device causing the interrupt, and (c) the CPU's execution speed and other characteristics.

The one thing worth a closer look is *how* the interruptions are measured, namely by the measure_duration() function. To do so, the function first takes the time, then executes some piece of code which does not produce a result but merely should, if uninterrupted, always have the same execution time represented here by the variable base.[3] Afterwards, the time is taken again and the elapsed time – the duration dt – is computed. If no interruption has taken place, then the duration should equal base. Otherwise it is increased by the duration of the interruption. This code thus detects and measures all interruptions that occur while the receiver is executing the measure_duration() function. At the same time this also introduces the problem of a good choice of the iterations parameter, which determines the runtime of the loop in measure_duration(). This runtime should be long in comparison to the other, simple operations in the run_for () loop, to keep the execution in the measuring part of the receiver most of the time. (Interruptions occurring while the receiver is executing outside of the measuring code are not detected.) At the same time it has to be short enough to not collect multiple interruptions at once, which would tamper with the results. Later, in Section 3.3, this problem will be addressed again.

[3] Even if the code is exactly the same, this can hardly be achieved on modern CPUs because of, e.g., their elaborate caching and branch prediction.

In its aforementioned GUI implementation, the receiver additionally included code to map the counted numbers of interruptions back to the assumed corresponding binary inputs, **0** and **1**, with the help of thresholds (see page 34). This has been removed here as it is independent from the basic concept of the transmission itself.

Quantum-time channel artifacts. During the development of the exploit it was discovered that not only the interrupts generated by the sender have an influence on the receiver's measurings, but also the running sender process itself. This comes from the fact that the receiver cannot easily distinguish between the interruptions deliberately caused by the sender calling a system function, and interruptions resulting from an involuntary switch to and back from the sender process, since such task switches are also the result of interrupts, normally coming from a timer chip. Thus, a sender process running only for a certain short period of time before yielding the CPU again (here via sched_yield ()) may falsely be counted by the receiver as a sender-initiated hardware interrupt.

For that reason the additional waiting function short_busy_wait () has been added to the sender's code to artificially elongate the sender's execution time by a certain duration, such that the receiver can no longer confuse the sender itself with the interrupts it is causing. Of course one could, instead of filtering out the sender process's influence, also deliberately use it to transmit even more information per time unit through the channel. This, however, would result in a combination of an IRCC and a quantum-time channel, and would go beyond the scope of this thesis.

2.1. Fixed Quantum Scheduling

Neither the implementation nor the analysis of quantum-time channels are within the focus of this thesis, but nevertheless this kind of covert channel has an influence on the analysis of IRCCs: As long as an attacker has the opportunity to use a quantum-time channel for his/her malicious purposes, he/she would probably not use a harder to realize and potentially more conspicuous IRCC (compare the paragraph after this introduction). However the simple assumption alone that quantum-time channels have been disabled on the target system is insufficient, since disabling quantum-time channels requires for changes of the operating system which, in turn, potentially interfere with the way IRCC exploits operate. Thus, in order to obtain realistic results for the capacity of an IRCC implementation, the analysis environment must at least sufficiently resemble one where quantum-time channels *are* impossible.

To obtain such an environment, the instance responsible for the assignment of the CPU time – the operating system – has to be adapted. More precisely, this affects the algorithm responsible for determining the process scheduling of the system, also known as the *process scheduler*. In the following it will shortly only be referred to as the *scheduler*, but it is not to be confused with other scheduling algorithms that may be contained in an operating system, like for instance the I/O scheduler (compare e.g. “The I/O Scheduler” in [BC05, Chapter 14]).[4] It has to be remarked that the primary goal of the scheduler modifications is to disable quantum-time channels, but they in addition also simplify the later analysis.

[4]The I/O scheduler may have an influence on an IRCC if it affects the chosen interrupt source like it is the case for the hard disk. Compare [KW91] for an idea of a covert channel that could explicitly exploit an I/O scheduler.

If an operating system virtualizes the CPU for every task by fixing the starting time and the duration of every *timeslice*, as proposed in [MS07], then not only quantum-time but also interquantum-time channels are no longer available. But as a consequence, the whole systems performs worse, especially with respect to interactive processes,[5] since even short and/or seldom operations always have to reserve and consume at least a whole timeslice. Simultaneously, also the performance of IRCCs on such a system may be reduced to a certain amount and the exploit, at least in its original implementation, might not even work at all.

In the following, at first an operating system will be chosen, whose scheduler will afterwards be modified to disable quantum-time channels according to a simple information flow policy. Interquantum-time channels will not be considered for reasons discussed below, after this introduction. The information flow policy shall ensure that no information is transferred from a set of *sender tasks* via a quantum-time channel to a set of *receiver tasks*. The idea behind this policy, in comparison to, e.g., the suggestion of [MS07], is that quantum-time channels do not necessarily have to be forbidden between every pair of running tasks: If a sender had superuser privileges, it would certainly not have to use a covert channel to transmit information to any other task. Also, other groups of tasks, like tasks running on behalf of the same user, are likely able to use overt channels for the communication among each other.

To establish the aforesaid information flow policy, (a) sender tasks always get a timeslice of the same length, independent of their static or dynamic priority,[6] and (b) even if a sender task blocks or yields the CPU earlier inside a timeslice, no receiver task gets scheduled before the end of the *full* timeslice. This enforcement of timeslices with a constant length will be referred to as *fixed quantum scheduling*, which is however not to be confused with the "fixed quantum" that is sometimes used to describe that a scheduling algorithm *allows* every task – independent of its static or dynamic priority – to use a full timeslice of the same length, but, if the task blocks or voluntarily yields the CPU earlier, schedules another runnable task immediately.

Quantum-time and interquantum-time channels. Quantum-time channels as well as interquantum-time channels are both related to IRCCs in that they all exploit that the CPU is shared between multiple processes and, in particular, the sender and the receiver of the respective covert channel. Nevertheless both classes are treated differently in this thesis: The former are explicitly eliminated while the latter are ignored.

Quantum-time channels, similar to IRCCs, require for a receiver process that is able to detect interruptions. The difference is that the latter channels exploit deliberately caused hardware interrupts, while the former exploit interruptions by the sender process itself. Quantum-time channels therefore do not depend on the availability of a suitable source for hardware interrupts and do not rely on triggering potentially conspicuous operations (reading from and writing to a hard disk may cause acoustic noise; sending packets over a network may appear suspicious to a firewall or intrusion detection system). A quantum-time sender can easily transmit more than only binary information by yielding the CPU after different amounts of time. Therefore quantum-time channels can potentially transmit

[5]The Linux kernel decides that a task is interactive depending on the difference between static and dynamic priority (compare Listing 2.4), where the dynamic priority in turn is set based on the task's sleep time.

[6]Constant timeslice lengths are a sufficient but not a necessary condition for ruling out quantum-time channels. Compare the later Section 4.1 for the discussion of an alternative.

information faster than the implemented IRCC exploit and would, thus, be preferred. A realistic setting for IRCCs should therefore not permit quantum-time channels.

Prohibiting interquantum-time channels, e.g. by fixing timeslice starting times of all tasks, is very complex to realize in practice – at least if a serious impact on the overall system performance is to be avoided. The simple approach of enforcing a round-robin scheduling between all tasks would lead to a lot of reserved but unused timeslices, since most tasks in a contemporary computer system are sleeping or blocked most of the time. In such an environment, the scheduling algorithm could easily implement an information-flow policy similar to the aforementioned one, but against IRCCs, by sorting the tasks sensibly (e.g. by ensuring that receiver timeslices are never placed immediately after sender timeslices). This would eliminate the possibility of IRCCs completely, unless hardware interrupts can occur very late after their generation. Thus, if suchlike performance degradations are acceptable on a given system, then IRCCs do not constitute a threat to it. The design and/or implementation of a more efficient countermeasure against interquantum-time channels would, however, go beyond the scope of this thesis. Therefore, this class of covert channels will be ignored for the remainder of this thesis.

2.1.1. Choice of Operating System & Scheduler

Before a scheduler can be modified it has to be decided *which* scheduler under which operating system it will be. The set of operating systems considered here is firstly restricted to the free, open source operating systems (for obvious reasons), which additionally run on the available hardware, in this case the Intel x86 architecture PC described in Section 1.5. Secondly, the author's personal experience and preference further reduces the set to Linux and FreeBSD. FreeBSD in its current version 7 allows a choice between the traditional *4BSD* and the new *ULE*[7] scheduler. Linux offers no choice in its current version as well as in previous ones, but had different schedulers over the course of its history: The *O(n)* scheduler is found in the 2.4 and earlier kernel series, the *O(1)* scheduler was the default from 2.5 to 2.6.22, and finally the *Completely Fair Scheduler* (CFS) is used since 2.6.23 until now.[8] Their underlying approaches can be categorized into three groups (sorted by their age):

(a) 4BSD and O(n) stem from the traditional UNIX scheduler and are becoming more and more obsolete because of their linear, $\mathcal{O}(n)$, time complexity of every task switch [Aas05, Rob03], where n is the number of runnable *tasks*. This complexity is the result of doing CPU usage accounting for sleeping tasks and choosing the next task by doing a linear search through the list of all tasks.

(b) ULE and O(1) both aim to have a constant, $\mathcal{O}(1)$, scheduling overhead, independent of the number of tasks. This is achieved by doing CPU usage accounting only for the currently running task and by using queues of ready tasks to be executed next, in order to simply pick the first instead of completely searching through all.

(c) CFS tries to do a fair scheduling even more than all the previous algorithms. Therefore it manages a tree of runnable tasks, descendingly sorted by the "gravest need for more CPU time" [Mol07]. For the sake of completeness it has to be mentioned that the sorted tree introduces a logarithmic, $\mathcal{O}(\log n)$, scheduling complexity.

[7] To be precise, the ULE scheduler in FreeBSD 7.0 goes under the name of "ULE 3.0".
[8] At the time of writing, 2.6.27 is the most recent stable version.

Presumably it is possible to modify all of the aforementioned schedulers such that they apply fixed quantum scheduling. The only difference is the effort necessary to do so. Since "the CFS scheduler has no notion of timeslices" [Mol07], this algorithm is likely farther away from fixed length timeslices than the others, which all have a notion of timeslices. Thus, CFS is dropped from the list of candidates. On the other hand it is assumed, that the CPU usage accounting of sleeping tasks entails some difficulties (what about a task that is voluntarily sleeping but whose timeslice has not yet elapsed?). For that reason, 4BSD and O(n) are also eliminated, leaving only ULE and O(1).

Initially, a patch was written for each the O(1) and the ULE scheduler, but after the development and testing process it was chosen to use only the O(1) scheduler for the thesis. This decision partly resulted from the fact that FreeBSD usually boosts blocked processes by assigning them a much higher priority so that they can quickly be rescheduled again when they are unblocked, e.g., because requested data from a hard disk has been read. These higher priorities are handled by the kernel differently from "normal" priorities and attempts to simply remove the boost resulted in a system that occasionally locked up completely or hung until an external event (a key press) occurred. If the priority boost is left untouched, then an unblocked task immediately got scheduled, destroying the fixed quantum behavior.

This problem is not of a very important nature, as the exploit is not expected to use blocking system calls that would actually trigger the boosting mechanism. Nevertheless the potential weakness remains and, thus, the developed ULE scheduler modification will only be described in short in Section B.2, but is apart from that not further considered in the thesis. The following section will hence describe the steps necessary to turn Linux's O(1) scheduler into a fixed quantum scheduler.

2.1.2. Scheduler Modifications

The O(1) scheduler is described in great detail in [BC05, Chapter 6] and [Aas05], and in short also in [Mol02]. Another explanation at this position is therefore assumed to be superfluous and will be omitted. All listings presented in this section originate from some location inside the modified file `kernel/sched.c` of the Linux 2.6.22.9 source tree. The names of newly added functions are prefixed by "fixed_quantum_" to be easily recognizable.

The modifications consist of three steps: Ensuring constant timeslice lengths, ensuring round-robin scheduling and, most important, ensuring that a receiver task is barred from execution as long as the last sender timeslice has not fully finished. The first two steps mainly simplify the analysis of the exploit and only secondarily avoid quantum-time channels.[9]

Information flow domains. As mentioned earlier, the modified scheduler will not implement strictly fixed CPU quanta for all tasks, but instead only a notion of fixed quanta by not scheduling a receiver task before the last full sender task's timeslice has elapsed. Thus, the first step is to devise some criterion by which ordinary tasks can be classified into the domains "sender," "receiver," or "other." In general, criteria may be chosen almost freely,[10] but for a realistic scenario, user or group IDs seem to be the most reasonable. Users with access to confidential data would then be put into the "sender" domain, while all other or all untrusted users would belong to the "receiver" domain. In order to avoid unnecessary user

[9] A quantum-time channel could otherwise be established by the sender process varying its interactivity.
[10] Some criteria may have side effects, as for instance different task priorities ("niceness").

```
/* Determine sender and receiver by their effective group ID */
#define IS_QUANTUM_TIME_SENDER(tsk)   ((tsk)->egid == 2222)
#define IS_QUANTUM_TIME_RECEIVER(tsk) ((tsk)->egid == 2223)
#define IS_QUANTUM_TIME_RELATED(tsk) \
    (IS_QUANTUM_TIME_SENDER(tsk) || IS_QUANTUM_TIME_RECEIVER(tsk))
```

Listing 2.3. Determining sender and receiver tasks.

or group switches for the experiments, *effective group IDs* have been chosen to accomplish this task, as the macros in Listing 2.3 show.

For the protection of a real system against quantum-time channels, it may be desirable to have for more than just one domain for the senders and one domain for the receivers to, for example, protect the secrets of each user from being disclosed to any other user. An implementation of lots of domains together with a complex information flow policy however is expected to require for a more complicated algorithm and more complex data structures for the decision of whether a task is allowed to be scheduled or not. Since for the experiments conducted with the exploit for this thesis the two domains "sender" and "receiver" perfectly suffice, the implementation of a more general solution has been avoided.

Constant timeslice length. The unmodified O(1) scheduler assigns timeslices to tasks, whose length depends on the task's static priority. For the analysis this is not a problem, since in contrast to the dynamic priority, the static priority remains constant unless it is explicitly changed, for instance via the `nice()` system call. Nevertheless, the timeslice lengths for sender *and* receiver tasks are explicitly fixed by the patch to the default timeslice length of 100ms, independent of a static priority, by adding few lines of code to the already existing `task_timeslice()` function. Fixed receiver timeslice lengths are not required to forbid quantum-time channels but have been added to simplify the later analysis of the channel.

Despite the timeslice length, also the *timeslice granularity* has an influence on when a task gets withdrawn the CPU in favor of another task even before its timeslice has elapsed. It only affects interactive tasks, i.e. tasks which have not made much use of the CPU recently, with the intention "that they do not monopolize the CPU" [BC05, p. 273]. Both the sender and the receiver of the exploit are implemented in a way that keeps the CPU busy, at least after the adaptations in the end of this chapter, and should thus not be considered interactive. However, due to the fact that the scheduler needs some time to learn this empirically, the granularity has an undesired effect at the very beginning of the exploit's execution. As a simple but elegant solution, an additional condition is inserted into the `TASK_INTERACTIVE()` macro such that it never recognizes sender or receiver tasks as being interactive. Listing 2.4 shows both modified functions.

Round-robin scheduling. Roughly speaking, the O(1) scheduler already performs round-robin scheduling among tasks with the same priority. This time, however, not the static but the dynamic priority of the tasks is relevant. The dynamic priority is related to the interactivity of a task and may therefore again only have unwanted influence on the beginning of an exploit's execution. Listing 2.5 shows the modified `__normal_prio()` function, which does no longer take the `CURRENT_BONUS()` into account for sender and receiver tasks.

```
/* Timeslice length for all FQ tasks. */
#define FIXED_QUANTUM_TIMESLICE DEF_TIMESLICE
static inline unsigned int task_timeslice(struct task_struct *p)
{
  if (IS_QUANTUM_TIME_RELATED(p))
    return FIXED_QUANTUM_TIMESLICE;
  else /* Non-FQ tasks are treated as usual. */
    return static_prio_timeslice(p->static_prio);
}

/* FQ related tasks are never interactive. */
#define TASK_INTERACTIVE(p) \
  (!IS_QUANTUM_TIME_RELATED(p) && \
  ((p)->prio <= (p)->static_prio - DELTA(p)))
```

Listing 2.4. Constant timeslice lengths for senders and receivers.

Another deviation from the strict round-robin scheduling is that interactive tasks under some conditions are not put into the expired array but back into the active array (in the task_running_tick () function). Since this behavior relies on the TASK_INTERACTIVE() macro, the modification of the previous paragraph has already fixed it.

Blocking receivers. The main objective of the scheduler modification is to prevent a receiver task from being executed before the full timeslice of the last sender task has elapsed. Therefore the kernel needs to keep track of sender timeslices and needs to have some way of preventing receiver tasks from being executed too early. The following four steps, accompanied by the four parts of Listing 2.6, clarify how this was realized. Listing 2.7 shows the implementation of the used helper functions that query and update the “internal” data structures of the scheduler modification.

(a) All data that the blocking mechanism needs to remember is put into a C structure **struct** fixed_quantum, which is added to each CPU’s runqueue (**struct** rq), which in turn contains, among a lot of other data fields, the data structures with all tasks assigned to the respective CPU.

    The fq_wait_ticks structure member counts the number of “ticks” (timer interrupts) that have to occur until the next receiver is allowed to be scheduled (on this CPU). This matches the way the O(1) scheduler handles timeslice durations: Before scheduling a task it assigns a number of ticks to it (precisely, to the time_slice member of the task’s **struct** task_struct) and the task is then allowed to execute for no longer than this many ticks. Therefore, the tick counter of the task is decreased by 1 every time the task_running_tick () function is executed as a consequence of a timer interrupt. Once the counter approaches zero, a reschedule is enforced.

    The Boolean member fq_expired , which is set to **true** by fixed_quantum_mark_expired() in task_running_tick (), stores whether the time_slice of an expired task has already been refreshed. For such tasks which consumed their full timeslice, the function fixed_quantum_update_leaving() must not consider the new time_slice value as the task’s spare timeslice duration but instead reset fq_wait_ticks to zero.

```
static inline int __normal_prio(struct task_struct *p)
{
    int bonus, prio;

    /* FQ related tasks do not get any bonus. */
    if (IS_QUANTUM_TIME_RELATED(p))
        return p->static_prio;

    bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;

    prio = p->static_prio - bonus;
    if (prio < MAX_RT_PRIO)
        prio = MAX_RT_PRIO;
    if (prio > MAX_PRIO-1)
        prio = MAX_PRIO-1;
    return prio;
}
```

Listing 2.5. No dynamic priorities for senders and receivers.

The initialization of the structure members is done by fixed_quantum_setup(), which should be called on system startup, for example in sched_init ().

(b) Whenever a sender task stops executing – either voluntarily or involuntarily –, the data structure from the last bullet point has to be updated. The natural code location to do this is inside the main scheduler function, schedule(), somewhere near the beginning. Similar changes would also have been possible at other locations that may cause a task to yield the CPU as, e.g., sys_sched_yield (). This latter approach would allow for a more fine-grained control, which for this thesis was not determined to be necessary, but would require more changes of kernel code.

(c) Also in the schedule() function, but later, the next task to be executed is selected. The scheduling decision must be intercepted if (and only if) a receiver was chosen and the last sender's timeslice has not yet elapsed. The check is performed by fixed_quantum_run_check() and in case of a negative outcome, the CPU's idle task (rq−>idle) is chosen for execution – independently of whether other tasks are ready. The chosen implementation thus realizes a suboptimal allocation of CPU time, which may even be advantageous for the IRCC exploit (compare with the idea of inter-timeslice gaps of Section 4.1).

(d) The fixed_quantum_run_check() function introduced in the preceding step has to return **false** as soon as the last sender's timeslice has elapsed. Therefore, the fq_wait_ticks value is decremented with every timer interrupt in the scheduler_tick () function, just like it is done for time_slice (in task_running_tick ()).

    Once the counter has reached 0, a reschedule has to be requested (by executing set_tsk_need_resched ()) in order to check whether a previously blocked task can now be scheduled. This is necessary since the idle task does neither stop executing by itself, nor does it get withdrawn the CPU due to, e.g., an expired timeslice.

(a) Data structure for inclusion into **struct** rq.

```
struct fixed_quantum {
    /* Number of ticks until next receiver task may run. */
    u_int fq_wait_ticks;
    /* Did last sender task expire (in task_running_tick)? */
    bool fq_expired;
};
```

(b) Recording a leaving sender task in schedule().

```
    /* For a leaving sender task, store the number of
     * remaining ticks in the last timeslice. */
    if (IS_QUANTUM_TIME_SENDER(prev))
        fixed_quantum_update_leaving(rq, prev);
```

(c) Checking whether to block a receiver task in schedule().

```
    /* Is the chosen task already permitted to execute? */
    if (!fixed_quantum_run_check(rq, next)) {
        /* May not run! Simply fall back to idle here. */
        next = rq->idle;
        goto switch_tasks;
    }
```

(d) Updating data structure in scheduler_tick().

```
    bool fq_idle_resched; /* Should reschedule if currently idle? */

    update_cpu_clock(p, rq, now);

    /* Update the fixed-quantum wait value. */
    fq_idle_resched = fixed_quantum_clock_update(rq);

    if (!idle_at_tick)
        task_running_tick(rq, p);
    else if (fq_idle_resched) {
        /* Idle thread is running and the fixed_quantum_clock_update
         * has determined that a reschedule might be necessary. */
        set_tsk_need_resched(p);
    }
```

Listing 2.6. Steps in blocking receiver tasks.

```
static inline void fixed_quantum_setup(struct fixed_quantum *fq) {
    fq->fq_wait_ticks = 0;
    fq->fq_expired = false;
}

static inline bool fixed_quantum_clock_update(struct rq *rq) {
    /* Return true iff fq_wait_ticks just changed to 0. */
    if (rq->fq.fq_wait_ticks == 0) return false;
    return (--rq->fq.fq_wait_ticks == 0);
}

static inline void fixed_quantum_update_leaving(struct rq *rq,
        struct task_struct *tsk) {
    struct fixed_quantum *fq = &rq->fq;
    if (fq->fq_expired) {
        /* Expired tasks already have their time_slice refilled in
         * task_running_tick, but it has been 0 before. */
        fq->fq_wait_ticks = 0;
        fq->fq_expired = false;
    } else {
        /* Store the unused ticks and manually refill the time_slice
         * already for the next time it is schedule()'d. */
        fq->fq_wait_ticks = tsk->time_slice;
        tsk->time_slice = task_timeslice(tsk);
    }
}

static inline void fixed_quantum_mark_expired(struct rq *rq,
        struct task_struct *tsk) {
    if (IS_QUANTUM_TIME_SENDER(tsk))
        rq->fq.fq_expired = true;
}

static inline bool fixed_quantum_run_check(struct rq *rq,
        struct task_struct *tsk) {
    /* Receivers may only be scheduled if no wait ticks are left,
     * all other tasks may always be scheduled. */
    return (!IS_QUANTUM_TIME_RECEIVER(tsk) ||
        (rq->fq.fq_wait_ticks == 0));
}
```

Listing 2.7. Helper functions for fixed quantum scheduling.

```
void send(data[]) {
  wait_sync();
  while (!transmission_finished()) {
    sched_yield();
    wait_until(gettime() + timeslice_length - epsilon);
    idx = get_bit_index(gettime());
    if (data[idx] == 1)
      generate_interrupt();
  }
}
```

Listing 2.8. Sending a sequence of bits (pseudo code).

```
void receive(interrupts[]) {
  wait_sync();
  while (!transmission_finished()) {
    dt = measure_duration(iterations);
    if (in_relevant_range(dt)) {
      idx = get_bit_index(gettime() - skew);
      interrupts[idx] += 1;
    }
  }
}
```

Listing 2.9. Receiving a sequence of bits (pseudo code).

2.2. Adaptation of the Exploit

In order to make the exploit function under the modified scheduler and to make it perform with higher transmission rates than envisioned for the original version, while still guaranteeing to a certain degree a memoryless channel, the exploit has to be adapted. As a result, large parts of the exploit code have been rewritten and only the channel's basic concept (see page 11) was retained. Due to its complexity, the full code for transmitting a bit sequence will only be shown in the appendix (Section B.1) in favor of simplified pseudo code versions at this point. Nevertheless, the full code is interesting as it reveals some nontrivial details about timing issues, especially in the receiver.

2.2.1. Fixed-Quantum Compatibility

The basic difficulty in establishing an IRCC is making a sender task initiate an operation whose result – an interrupt caused by some hardware device – happens while the receiver task is executing. To achieve this, the sender of the original exploit yielded the CPU right after generating the interrupt. The receiver could then, if no other tasks demand for CPU time, immediately be scheduled and detect the interruption occurring during its runtime. This idea however assumed a default scheduler. With the scheduler modifications, a receiver task is blocked until the end of the sender's full timeslice and, thus, the interrupt generated by the sender near the beginning of its timeslice is likely to occur while the sender's timeslice has

not yet expired. To achieve that even under the changed conditions the receiver is executing when the interrupt occurs, the sender must ensure that the generation of the interrupt and the following yielding of the CPU happens close to the end of the respective timeslice.

On modern operating systems, scheduling happens transparently to the running processes, i.e. a task is not explicitly notified that it has just been scheduled or when it gets withdrawn the CPU next time. While for almost all applications this is a desired behavior, it is harmful for the exploit's sender. Fortunately, the sender can use a function like sched_yield (): On the one hand, an invocation of this function yields the CPU and thereby gives away the remainder of the current timeslice, but on the other hand it guarantees that when the caller next time resumes its execution, it does so in a newly begun timeslice. If the sender knows the timeslice length (timeslice_length), then by taking the time immediately at the beginning of the new timeslice, it can also calculate when the timeslice is expected to end. Correspondingly, it is able to delay the generation of the interrupt until shortly before this end of the timeslice via wait_until () (see below on how to realize such a delay). Right after generating the interrupt, the sender can then again call sched_yield () to ensure that also the next beginning of a timeslice is known.

2.2.2. Higher Transmission Rates

The original exploit was programmed to support transmission rates of a single bit in at least three or four seconds. For this purpose, timers have enough accuracy and a short gap of, say, one second suffices to not account an interruption to the wrong bit, which would violate the assumption of a memoryless channel.

For a proper analysis of the exploit, it should support higher transmission rates, up to a point where only a single interrupt's generation and reception is possible during a bit interval, i.e. where bit intervals are "atomic". Assuming a timeslice duration of 100 ms and both parts of the exploit consuming always full timeslices, this atomicity would be given at a bit interval length of 200 ms. For intervals of this length or shorter, the normal timers (as provided by the C library functions setitimer () and, especially, alarm()) do not provide enough accuracy and, thus, timers have been removed from the exploit code completely.

When only waiting until a certain point in time, the timer can be replaced by a "busy clock querying." Where code is executed until a timer fires, the timer has to be replaced by sporadic clock queries. Details about how this has concretely been done for the receiver can be found in Section B.1.2.

Retaining a memoryless channel even without long gaps and at higher transmission rates requires for a different transmission framework. Instead of sending and receiving single bits sequentially with functions like in Listings 2.1 and 2.2, the transmission interval for *all* bits is now taken as a whole. Then during the full interval, sending and receiving is done without any explicit bit interval boundaries in the code. The sender tries to "send" every single timeslice it is executing, but whether it has to send a **0** (do nothing) or a **1** (generate an interrupt) is decided shortly before the end of the respective timeslice by (a) taking the time, (b) computing the current bit index, and (c) finally obtaining the bit value. Analogously, the receiver permanently receives (measures potential interruptions) and only in the case of an interruption inside the relevant duration range, the receiver (a) takes the time, (b) computes the current bit index, and (c) finally increases the interruption counter for the respective bit.

2.2.3. Parameters

The implementation of the exploit determines the basic characteristics of the IRCC transmission. For instance, sender and receiver are assumed to run alternatingly, with the sender triggering either exactly one or none interrupt, depending on its input, for every timeslice. The receiver during its runtime repeatedly tries to capture interruptions and counts their number for every bit interval. Nevertheless this fundamental decision does not determine every aspect of the exploit. Parameters at several places in the exploit can be almost freely chosen without changing the communication paradigm, but having a huge influence on the realization and the quality of the IRCC.

In the following, the identified parameters of the implemented exploit are grouped and explained. Changes of the implementation however, even if they do not alter the basic channel ideas, may introduce new parameters, modify the meaning and influence of present parameters or even make parameters obsolete.

Timing. First and foremost, sender and receiver have to know, *when* the transmission of *which* bit takes place. Therefore, both applications have parameters for a transmission starting point, a bit duration and the number of bits to be transmitted. Since bits are sent consecutively, this determines the temporal position of every bit interval belonging to the transmission. Obviously the number of bits cannot be as freely chosen as both other parameters, but is rather predetermined by the amount of data to be transmitted. Similarly also another timing parameter, the timeslice length, needs to be known to the sender but is also predetermined – in this case by the operating system.

The implementation of the receiver shows another parameter related to timing: the `skew`. Its value determines how much the bit intervals in the receiver are moved into the future, relative to the bit intervals in the sender. If, for example, the skew is set to 1 ms, then for every bit index i, this i-th bit in the receiver starts and ends 1 ms later than the same bit starts and ends in the sender. The reason why such a parameter is necessary or, at least, useful will become clear later, when the adjustment of parameter values is examined. A related parameter, which is not shown in the pseudo code of the channel, is the *gap*. This gap is a short amount of time around bit interval boundaries, during which the receiver has to ignore detected interruptions. It will also be further motivated and described in the analysis part of the thesis.

Generation of interrupts. In order to trigger the later occurrence of an interrupt, the sender in general has to call a system function, which usually requires for a list of parameters to be specified. In case of hard disk interrupts generated via `read()` or `write()`, this would be a file descriptor representing an opened file and its current offset, and a number of bytes to be read or written. For NIC interrupts and the `sendto()` function, the parameters are a (UDP) socket, a destination address and port, the payload length[11] and potentially also a combination of flags.

Besides the decision of *how* the interrupt is generated it can also be configured *when*, if at all, the generation takes place within the respective timeslice. This is controlled by the parameter `epsilon` from Listing 2.8, which determines how early before the end of the

[11] The payload itself is not considered as it does not change the characteristics of the generated interrupt or its generation itself.

timeslice the sender calls the interrupt generating function. Its value must be chosen large enough that there is enough time to call the function and subsequently yield the CPU. It has to be small enough that the generated interrupts do not occur already within the sender's timeslice. Section 3.3 takes a closer look on this parameter's influence of the channel performance.

Detection of interrupts. Listing 2.9 shows that the receiver passes the iterations parameter to function measure_duration(), which has not been changed since Listing 2.2. This parameter indirectly controls the duration base (in Listing 2.2) of a measuring phase. Hidden in the not shown pseudocode function in_relevant_range () there are also parameters used for the decision of whether an interruption is relevant for the channel or not. In the concrete implementation, a lower and an upper bound for the interruption duration are used to do this task.

3. Analysis

The previous chapter shows one possibility of how an IRCC exploit can be implemented and how an operating system can be modified such that it provides a setting in which IRCCs might be used for circumventing security mechanisms. Based on the implementations and their underlying concepts, the now following chapter theoretically and experimentally analyzes, which bandwidths *can* be achieved in principle, and which bandwidths *are* achieved by the exploit.

It will be shown that not all sender-generated interrupts are actually detected by the receiver process and that this may be caused not only by the activities of the system environment but also by a disadvantageous configuration of the exploit parameters. For a better understanding of this last-mentioned aspect, a generic framework will be developed that captures the influence of parameter values on the channel bandwidth, depending on the properties of the system. On the basis of the testing environment, as introduced in Section 1.5, it will be shown how the framework can be applied to a real system to obtain suggestions for parameter values that are expected to lead to high detection rates and bandwidths. Finally, at the end of the chapter, it will be considered that not only may sender-initiated interrupts remain undetected but also could other interrupts be falsely considered as originating from the sender. It will be shown how relatively small modifications to the exploit can remarkably reduce the likelihood of such *false positives*.

3.1. Basic Channel Model

For the whole analysis, the implemented IRCC is modeled as a discrete memoryless channel (I, O, P), as introduced in Section 1.4. The input alphabet I and the output alphabet O result from the chosen implementation of the channel and, hence, do not give much leeway to the model. By contrast, the transition matrix P is the core of the channel model. It could be rather simple and assume a perfect, undisturbed transmission (see Section 3.2.1), but might as well include the possibility of interrupt losses (see Section 3.2.2) or even of arbitrarily complex forms of noise (compare Section 3.4).

Input alphabet. In the given implementation, the sender process of the channel discriminates between two states for every bit: generating no interrupt at all and generating one interrupt every timeslice in the bit's interval. Hence, independent of the concrete bit interval length, the input alphabet is binary, represented by $I = \{\mathbf{0}, \mathbf{1}\}$.

Output alphabet. The receiver's output for a bit is the number of counted interruptions within the respective bit's interval and must therefore be a nonnegative integral number. Thus, for the output alphabet $O = \mathbb{N}$ is chosen. This avoids an early commitment to a specific upper bound on the number of counted interruptions, which would not even have an advantage for the model. For the sake of completeness it has to be mentioned that one

could, if it is desired, for example establish an upper bound based on the maximum number of measurings during a bit interval.[1]

In principle it would also have been possible to transform the counted numbers of interruptions back into binary values, with the intention that the output of the channel should ideally equal the input anyway. While this is innocuous for the output alphabet itself, problems arise when the transition matrix is concerned: Then it becomes important *how* the transformation is realized, i.e. which numbers of interruptions are to be considered a **0** and which a **1**. A bad mapping also leads to a bad channel capacity, as the example of a "**0**-only" transformation points out. However, the data-processing inequality (see [CT06, Theorem 2.8.1]) shows that even the best transformation possible can never improve the content of the output, but may lose information. Thus, neither for the implemented receiver nor for the model of the channel, an explicit binary output has any advantages to the choice made above.

Channel properties. The choice of a finite input alphabet I and a countably infinite output alphabet O makes the channel discrete by definition (see Section 1.4). What hence remains to be discussed is the memorylessness of the implemented channel, requiring that the probability $p(y|x)$ of an output $y \in O$ depends only on the current input $x \in I$ and, thus, not on earlier inputs and/or outputs. Deliberately, the exploit does not make use of earlier outputs,[2] but the exploit cannot fully exclude the possibility that involuntarily an interrupt belonging to one bit interval is received in the respective succeeding bit interval – or vice versa. Suchlike artifacts will be referred to as *misaccountings* and may be due to

- the sender being interrupted between determining which bit value is to be sent and the actual interrupt generation (see Listing 2.8 on page 23),
- the receiver being interrupted between the end of the interruption measuring and the determination of the bit index (see Listing 2.9 on page 23),
- the interrupt taking an unexpectedly short or long time from its generation until its occurrence, e.g., as a result of an in-kernel or hardware delay,
- the arrival of unexpected response packets, causing additional interrupts after a longer delay (this is specific to a NIC-based channel).

The causes named in the first two bullets are possible but unlikely, since the time window for such interruptions is rather small. The likelihood of the third bullet depends on what times the exploit "expects" and, thereby, on the configuration of the exploit (especially the `skew` parameter). Practical experiments showed, that if exploit parameters are setup correctly (see Section 3.3), then misaccountings are very rare – at least on the author's hardware. At least for the following section it will therefore be assumed that the implemented channel is practically memoryless.

[1] Using symbols to be introduced later in this thesis, this upper bound would be $\lfloor t_b/(t_m + t_e) \rfloor$.

[2] It could, for example, relate the current output to the average of the previous outputs to filter noise.

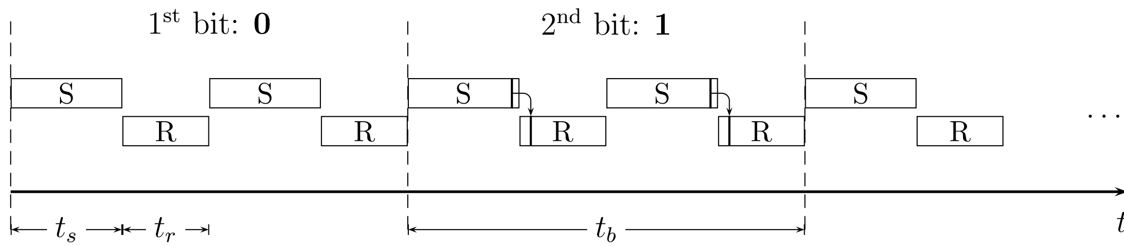


Figure 3.1. Schematic transmission from sender (S) to receiver (R) with bit interval length $t_b = 2(t_s + t_r)$.

3.2. Transmission Models

Based on the exploit implementation and the knowledge about the system's scheduling behavior, a model of IRCC transmissions can be constructed. Initially, to compute an upper bound on the transmission rate, this model will completely ignore any kind of distortion. Afterwards the model is extended by the possibility of interrupt losses.

Scheduling and synchronization. The exploit's sender and receiver task have to run concurrently during a transmission and, thus, require for synchronization of the single bits. Therefore, it is first assumed that both tasks have access to the same clock having a sufficing accuracy.[3] Using this common clock, sender and receiver task agree on a starting time for the whole transmission as well as on the bit interval length. As a result, both tasks have the same view on when bit transmissions start and end, which is depicted in Figure 3.1. For the sake of simplicity, a bit interval skew has been omitted.

The figure also illustrates how the two processes are assumed to be scheduled: Strictly alternating and each with a constant timeslice length. Firstly, the simplifying assumption of no disturbances also implies that no other tasks appear in the figure, next to sender and receiver. Secondly, the scheduler's round-robin policy ensures that right after one task has stopped its execution, the respective other task gets scheduled, forcing the processes to take turns. Thirdly, the scheduler dictates the timeslice length for the sender, say t_s, which the latter can neither shorten nor lengthen. Finally, the receiver in the given implementation does never voluntarily yield the CPU or execute a blocking system function and, thus, always uses the maximum available timeslice length, say t_r. In contrast to t_s, t_r does not have to be made constant for avoiding quantum-time channels, but the modified Linux scheduler guarantees a fixed t_r to simplify the analysis (see Section 2.1.2). Although the modified scheduler implements $t_s = t_r = 100\,\mathrm{ms}$, the model allows arbitrary other $t_s \neq t_r$ as well.

The second bit in Figure 3.1 demonstrates how interrupts are "sent" and "received": Short before the end of its timeslice, the sender calls some function which causes a hardware interrupt to occur sometime at the beginning of the following receiver's timeslice. This causality is indicated by an arrow. It can be seen that only one interrupt is triggered every sender/receiver timeslice pair with duration $t_s + t_r$. In order to avoid a drift between timeslices and bit intervals, the bit interval length t_b is assumed to be a positive integral multiple of $t_s + t_r$. As a consequence, the number of possible interrupts per bit interval does

[3] The question of how accurate the clock has to be will later be touched in Section 3.3.2.

not depend on the offset that the sender/receiver timeslices have, relative to the respective bit's starting time.

3.2.1. Lossless Transmission

In order to obtain an upper bound on the transmission rate possible with the implemented exploit, it is now assumed that sender and receiver are scheduled as depicted in Figure 3.1 and that all interrupts generated by the sender "arrive" at the receiver in the respective same bit interval. Furthermore it is assumed that noise is nonexistent so that the receiver never counts more interrupts than the sender has generated.[4]

The bit interval length is fixed to $t_b = N \cdot (t_s + t_r)$, for some number $N \in \mathbb{N} \setminus \{0\}$ of sender/receiver timeslice pairs per bit interval. In order to transmit a **0**, the sender does not generate any interrupts and, thus, the output of the receiver must be 0. On the other hand, if a **1** is transmitted, then the sender generates one interrupt every timeslice pair, summing up to a total of exactly N interrupts per bit interval which are also counted by the receiver. The output alphabet of the channel can in this case therefore be restricted to $O = \{0, N\}$ with a probability matrix

$$P = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \end{array} \begin{array}{c} \begin{array}{cc} 0 & N \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{array}, \tag{3.1}$$

where rows represent inputs and columns represent outputs.

Transmission rate. Independent of the value of N, the capacity of this channel is 1, since the output completely reveals the input. But due to the fact that the bit duration t_b is proportional to N, it is apparent to choose the smallest possible, $N = 1$, for a maximal transfer rate of 1 bit per time interval $t_s + t_r$.

Given $t_s = t_r = 100\,\text{ms}$, as implemented in the modified scheduler, this results in a bandwidth of $B = 5\,\text{bit/s}$. The receiver of the exploit, however, is not necessarily forced to consume full timeslices like it is the case for the sender. As a consequence the receiver could, in principle, yield the CPU at an offset within its timeslices, where no interrupts coming from the sender are expected to occur afterwards (see Section 3.4.2). With very early interrupts and t_r approaching – though obviously not reaching – $0\,\text{ms}$, the possible bandwidth B comes close to $10\,\text{bit/s}$, if $t_s = 100\,\text{ms}$ remains fixed.

These results can be compared to the $11.5\,\text{bit/s}$ determined in [MS07] and the $50\,\text{bit/s}$ of [MS08, Example 3], which both assume $t_s = t_r = 100\,\text{ms}$ but "transmit" multiple interrupts during every sender/receiver timeslice pair. The former article, though indirectly, also uses the number of interruptions as output; the latter additionally considers the times and durations of interrupt occurrences.

3.2.2. Lossy Transmission

In reality, the exploit is not perfect and especially not the only tasks on the system. Consequently an IRCC transmission does not proceed as perfect as shown in Figure 3.1 and, hence, not at the maximum possible transmission rate. This section therefore refines the model of the previous section by a parameter $\lambda \in [0, 1)$ which is the probability for each of a

[4] A practical justification for this assumption can be found in Section 3.4.

bit interval's N possible interrupts to be either not generated or not detected. The case that *all* interrupts are lost, i.e. $\lambda = 1$, is explicitly excluded from the model, because it would already intuitively disable any IRCC transmission (see Lemma A.1) and is therefore of no practical interest. It is furthermore assumed that the losses of interrupts are independent of each other, not only across bit interval boundaries so that the channel is memoryless, but also within bit intervals.

In the following, some of the reasons for interrupt losses will be introduced and the assumption of stochastic independence will be discussed. Afterwards the refined model is presented and further analyzed.

Other tasks. Even on a very minimalistic system there are kernel threads or, for example, the `init` process that wake up from time to time to do their work. If such a foreign task gets scheduled between a sender/receiver timeslice pair, then the interrupt generated by the sender is likely to interrupt this foreign task instead of the receiver and, thus, the interrupt is lost. Even if the foreign tasks gets scheduled at another position, the CPU time assigned to it reduces the time available for sender and receiver, potentially lowering the number of transmittable interrupts in the respective bit interval.

Whether the supposition of stochastically independent interrupt loss probabilities can be sustained depends on the pattern with which foreign tasks execute. The following counter-examples shall give an impression:

- A task waking up periodically with a fixed frequency makes the losses of different interrupts dependent of each other. This dependency becomes apparent especially if the wake-up interval is shorter than or not much longer than t_b.

- If two or more tasks share the same wake-up frequency, even if it is low, then it depends on the offset between them whether they amplify themselves within a single bit or distribute the losses more uniformly between multiple bits.

  This behavior was discovered with the `pdflush` kernel task and the `init` process, which by default both wake up every 5 seconds. For the testing environment, the former has therefore been configured to use a longer interval and the wake-up of the latter has been completely turned off (see Section 1.5).

- Tasks waking up sporadically but for a longer period of time, thereby interfering with the transmission of multiple interrupts in a row.

As a consequence it is assumed, that foreign tasks do not (a) wake up in (relatively) short intervals, (b) share the same wake-up timer frequency, or (c) perform extensive operations once they get scheduled. The satisfaction of these conditions can be examined on a given system by adding logging functionality to the scheduler and evaluating its output. This is also how the behavior of `pdflush` and `init` was spotted.

Imperfection of the receiver. An observation of the receiver implementation – even in the pseudo code of Listing 2.9 on page 23 – shows that the receiver is not all the time able to detect interruptions, but only while it is executing the measure_duration() function. An interruption during the evaluation of the measurement, for example, would not even be recognized. Assuming that interrupts occur independent of whether the receiver currently

measures or not,[5] the probability of a loss corresponds to the fraction of time which the receiver is executing outside of the measuring code.

Another reason for interrupt losses are collisions with other events, like other interrupts or other tasks waking up within one of the receiver's timeslices. Suchlike collisions would, from the perspective of the receiver process, look like a single, longer interruption, which the receiver may no longer consider as being caused by the sender and therefore ignore.

Both, the fraction of time used for evaluation and also the collisions will be subject to a closer inspection in Section 3.3. Until then, the loss probability λ is simply treated as a given constant that allows for a formal specification of the channel.

Transition matrix. If a $\mathbf{0}$ is sent through the implemented channel, the sender does not generate any interrupts during the full bit interval. Consequently, the loss probability has no influence on this case. Since furthermore by assumption there is no noise that could falsely be counted by the receiver, the input $\mathbf{0}$ always produces the output 0.

By contrast, a $\mathbf{1}$ may now produce outputs from 0 (all interrupts are lost) to N (no interrupts lost), including all of the intermediate values. Concerning the whole bit interval, the transmission of a $\mathbf{1}$ can be seen as a sequence of N independent yes/no (detection/loss) experiments – a *Bernoulli trial*. The number of positive results ("successes") of such a trial is distributed according to the *binomial distribution* (compare e.g. [Hoe66, 5.2.3]). In the concrete case, the only parameter of the distribution is $1 - \lambda$, the success probability for the detection of every single interrupt.

Putting both cases, $\mathbf{0}$ and $\mathbf{1}$, together, the transition matrix $P = (p(y|x))_{x \in I, y \in O}$ results, with $p(y|x)$ according to

$$\begin{aligned} p(y|X{=}\mathbf{0}) &= \begin{cases} 1 & \text{if } y = 0, \\ 0 & \text{otherwise,} \end{cases} \\ p(y|X{=}\mathbf{1}) &= \begin{cases} \binom{N}{y}(1-\lambda)^y \lambda^{N-y} & \text{if } 0 \le y \le N, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{3.2}$$

Theoretical capacities. Under the assumption that the loss rate λ is known, the channel capacity can be computed according to Section 1.4. The only nontrivial part of the computation is handled by the following theorem.

Theorem 3.1. *Let (I, O, P) be a discrete memoryless channel with $I = \{\mathbf{0}, \mathbf{1}\}$, $O = \mathbb{N}$ and P according to Equation* (3.2)*, for some $\lambda \in [0, 1)$ and $N \in \mathbb{N}$. Let $\rho := p(Y{=}0|X{=}\mathbf{1}) = \lambda^N$. Then the* maximal mutual information *between input and output is achieved by the input being distributed as*

$$p(X{=}\mathbf{1}) = \left(1 - \rho + \rho^{-\rho/(1-\rho)}\right)^{-1}$$

and $p(X{=}\mathbf{0}) = 1 - p(X{=}\mathbf{1})$.

Based on Theorem 3.1, the computation of the channel capacity *Cap* is as simple as computing the mutual information for the input being distributed according to the theorem.[6]

[5] This assumption may sound obvious but could be false if the receiver was implemented, e.g., in a way that a new receiver timeslice always starts at a fixed code position, like it is the case for the sender.

[6] The full formula is not shown here due to its length, but can be found in Equation (A.1) on page 67.

N	λ 0.1%	1.0%	10.0%	50.0%
1	0.994	0.960	0.763	0.322
2	1.000	0.999	0.960	0.558
5	1.000	1.000	1.000	0.900
10	1.000	1.000	1.000	0.994

(a) Capacity (bit)

N	λ 0.1%	1.0%	10.0%	50.0%
1	4.972	4.799	3.814	1.610
2	2.500	2.498	2.399	1.396
5	1.000	1.000	1.000	0.900
10	0.500	0.500	0.500	0.497

(b) Bandwidth (bit/s)

Table 3.1. Capacity and bandwidth under lossy transmission ($t_s = t_r = 100\,\text{ms}$).

The channel bandwidth also immediately resolves to $B = Cap\,/t_b$. For some chosen values of λ and N the results have been compiled into Table 3.1.

Shannon has shown in his famous paper [Sha48] that every transmission rate up to the channel capacity is possible at an arbitrarily low error rate – provided an appropriate channel encoding is used. Thus, it has to be considered that an attacker, who is able to implement and use such an encoding, would choose $N = 1$ and would then be able to almost reach the bandwidths shown in Table 3.1b.

Determining $\boldsymbol{\lambda}$. Finding out the concrete value of λ for a given system can in principle be performed theoretically or empirically. The latter method naturally is only possible if the system is at one's disposal, which might not necessarily be the case for an attacker. However, the alternative of determining λ theoretically can easily become cumbersome even on a small system, since it would involve modeling for example (a) the behavior of all running tasks and their interaction with the exploit (through the scheduler), and (b) the temporal positions of the receiver's evaluation phases. Thus, to determine λ and check the validity of the "lossy transmission" model, practical experiments are performed.

In order to show that both $\mathbf{0}$ and $\mathbf{1}$ are received according to the matrix described in Equation (3.2), at first the absence of additional noise is ensured – for example by transmitting a sequence of $\mathbf{0}$'s and verifying that the output is always 0. Afterwards, an alternating sequence of zeroes and ones, $(\mathbf{10})^k$, is transmitted through the channel. Transmitting the different symbols alternatingly instead of separately (e.g. via $\mathbf{1}^k\mathbf{0}^k$) is primarily intended to show one kind of memorylessness violations: Misaccountings. If an interrupt is accounted to a wrong bit interval, then it is almost certain that this interval belongs to a $\mathbf{0}$, which is the immediate predecessor and successor of each $\mathbf{1}$. Since a $\mathbf{0}$ is already known to always transform into a 0, misaccountings are immediately visible where $\mathbf{0}$'s transform to nonzero outputs.

Once the memorylessness is witnessed, λ can be determined with the help of the *maximum-likelihood estimator* (MLE, compare e.g. [Hoe66, 3.3]) for the binomial distribution: $\lambda = 1 - \bar{y}/N$, where $\bar{y}$ is the mean observed output for input $\mathbf{1}$ (see Theorem A.1). A *G-test* (see e.g. [Wik08]) as one example of a goodness of fit test (compare e.g. [Hoe66, Chapter 10]) can finally be applied to test the hypothesis of whether the numbers of counted interrupts are indeed binomially distributed. For more information about these tests, the interested reader is referred to the above literature.

Table 3.2 shows the experimentally determined values for λ under different parameter

Setting	t_m	λ	Cap (bit)	B (bit/s)	G
1	0.183 μs	14.1%	0.698	3.492	0.05
		14.3%	0.696	3.478	0.05
2	1.823 μs	1.80%	0.935	4.676	5.28
		2.11%	0.927	4.633	8.33
3	181.2 μs	0.10%	0.994	4.972	0.02
		0.34%	0.984	4.918	23.3
4	724.8 μs	0.31%	0.985	4.924	43.8
		0.32%	0.984	4.922	42.8
5	1105 μs	1.02%	0.959	4.796	0.03
		3.57%	0.889	4.447	0.15
6	1275 μs	10.9%	0.748	3.740	30.1
		15.4%	0.680	3.399	31.8

Table 3.2. Experimentally determined λ for different parameter settings.

settings.[7] For each setting, two transmissions of the bit sequence $(\mathbf{10})^{8000}$ were conducted with $N = 2$ and the default timeslice lengths $t_s = t_r = 100\,\text{ms}$. Every single transmission, thus, took roughly 1¾ hours. What the table does not show is that for each experiment there were no misaccounted interrupts. Capacities and bandwidths are based on the respective value of λ only (not on the practically determined values for $p(y|X = \mathbf{1})$) and use $N = 1$. Without going into detail about the meaning of the settings, which will later be the subject of Section 3.3, it can be seen that for instance Settings 3 and 4 yield interestingly low loss rates and corresponding high capacities and bandwidths, while the outcomes of Settings 1 and 6 are significantly worse.

Finally, the G values of the table's last row shall be compared to the values of the χ^2 distribution for 1 degree of freedom.[8] In case of a Type I error (the probability of rejecting a true hypothesis) of 5%, one obtains $\chi^2 \approx 3.841$ (see [Hoe66, p. 401]). Since this value is exceeded by most of the G values, especially in Settings 4 and 6, the hypothesis of a binomial distribution and, with it, also the assumption of stochastically independent interrupt losses, should at least be considered questionable. Despite the results of the G-test, the assumption of a binomial distribution is sustained for three reasons: (a) The test did not reject the hypothesis in all cases but instead for some experiments yields even very low values. (b) The outputs even in case of large G values were not completely dissimilar to the binomial distribution but showed only comparatively large values of $p(Y{=}0|X{=}\mathbf{1})$. (c) It was expected in advance that the binomial distribution is only an approximation (compare page 31).

It is unexplained why in most cases the computed G values belonging to one parameter setting are almost equal. The transmission logs belonging to the experiments, which show the concrete numbers of generated and detected interruptions for each bit, did not give any hints.

[7] More about the parameter settings used for the experiments can be found on pages 49 and 53.

[8] The degree should be $k - 1 - l$ for k observed values and l distribution parameters [Hoe66, 10.4]. Here $k = N + 1 = |\{0, \ldots, N\}|$ and $l = 1$ for only λ.

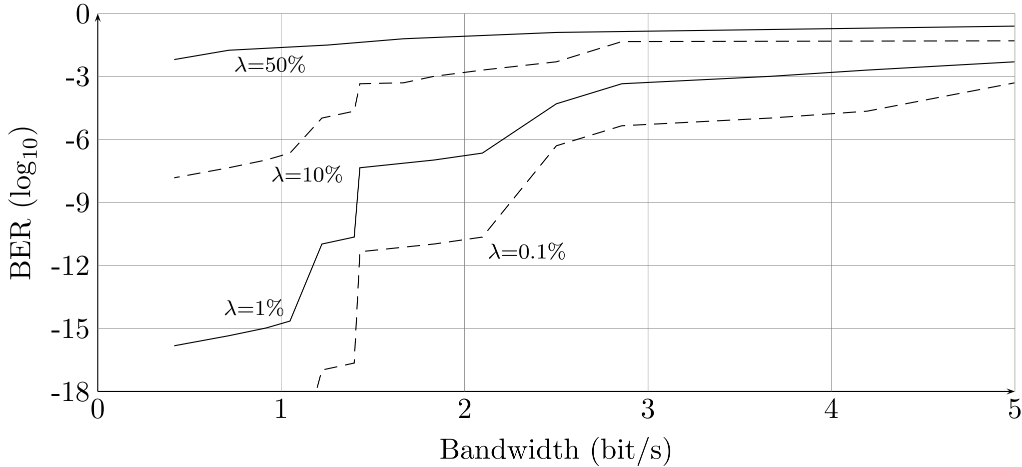


Figure 3.2. Bandwidth and bit error rates ($t_s = t_r = 100\,\text{ms}$, $N \leq 4$, $l \leq 31$).

Simple encodings. If, for some reason, an attacker is not able or willing to implement a sophisticated encoding that approximates the channel capacity, he/she might fall back to a simpler method: *Thresholds*. To determine which bit has potentially been sent, the receiver compares the counted number y of interrupts within a bit interval against a certain threshold value τ. For $y \geq \tau$, the output y is then decided to be a $\mathbf{1}$, for $y < \tau$ it would be a $\mathbf{0}$.

Theorem 3.2. *Let $\mathfrak{T}_\tau : O \to I$ be a function transforming channel outputs into expected channel inputs according to threshold $\tau \in \mathbb{N}$: $\mathfrak{T}_\tau(y) = \mathbf{1}$ iff $y \geq \tau$. Let P_τ be the transition matrix resulting from P when applying $\mathfrak{T}_\tau$ to the outputs of (I, O, P). Then*

$$Cap(I, I, P_1) = \max_{\tau \in \mathbb{N}} Cap(I, I, P_\tau) = Cap(I, O, P).$$

The capacity of the lossy channel is thus not reduced by choosing a threshold of $\tau = 1$. Furthermore, all other threshold values might only lead to lower capacities but never to an increased one. A similar result can also be found for the *bit error rate* (BER), i.e. the probability that the output (after applying the transformation function) differs from the input.

Theorem 3.3. *Let, for $\tau \in \mathbb{N}$, $\mathfrak{T}_\tau$ and P_τ be defined as in Theorem 3.2. Then for all $\tau \in \mathbb{N}$ with $Cap(I, I, P_\tau) > 0$ it holds that $p(\mathfrak{T}_1(Y) \neq X) \leq p(\mathfrak{T}_\tau(Y) \neq X)$.*

The optimal threshold $\tau = 1$ of Theorems 3.2 and 3.3 is rather intuitive: All $\mathbf{0}$ inputs are “guessed” correctly and only those $\mathbf{1}$ inputs wrongly for which no interrupts are detected during the respective bit interval. Larger threshold values would only identify more $\mathbf{1}$ inputs falsely as a $\mathbf{0}$. The only possible lower threshold, $\tau = 0$, would identify all $\mathbf{0}$ inputs falsely.

If a threshold of $\tau = 1$ is used for the encoding, then the most simple approach for lowering the error probability is to increase N, since counting not even a single interrupt in a bit's interval is less likely the more interrupts have (potentially) been generated.[9] Obviously, longer bit intervals also decrease the bandwidth, so there is a tradeoff between speed and reliability.

[9]Formally, for $\lambda \in (0, 1)$, $p(Y{=}0 | X{=}\mathbf{1}) = \lambda^N$ is strictly monotonic decreasing with increasing N.

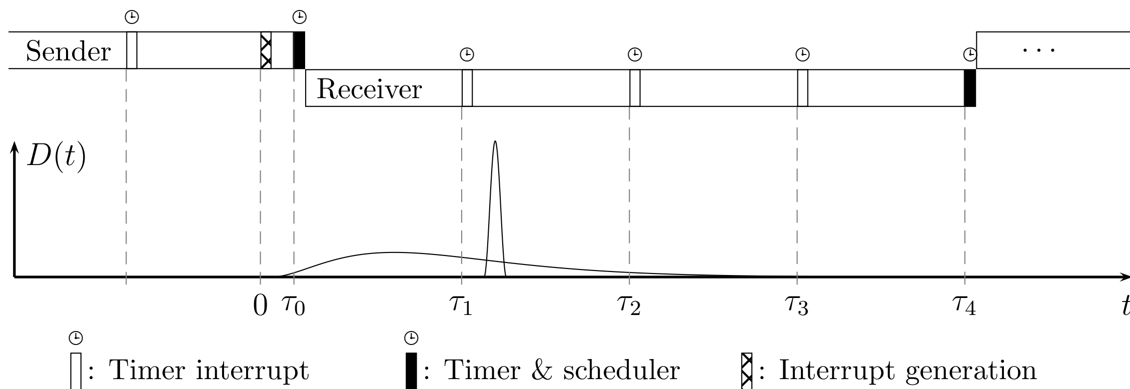


Figure 3.3. Refined model including timers and variable interrupt delay.

Next to varying N, also error correcting codes can be applied to reduce the error rate. For an example, the simple and popular *Hamming codes* have been chosen, which introduce the block length l as a new parameter and are able to detect and correct a single erroneous bit inside a block. Figure 3.2 shows that even at a relatively high loss probability of $\lambda = 10\%$ the error rate can easily be reduced below one in a million while at the same time staying above 1 bit/s.[10] The figure is based on the assumption that $p(X{=}\mathbf{0}) = p(X{=}\mathbf{1}) = 0.5$ and shows the discrete computed values connected for a better readability.

3.3. Adjusting Exploit Parameters

The present model focuses on the very basic idea of the implemented IRCC exploit: Sender and receiver are scheduled alternatingly and at most one interrupt is "transmitted" every timeslice pair. Based on Theorem 3.1 attackers as well as potential victims are able to estimate the value or, respectively, threat to a certain target system resulting from the given exploit. The potential victim could simply be pessimistic, assume a low loss rate λ and, from the computed channel capacity, evaluate whether and which countermeasures against this kind of channel are needed. By contrast, an attacker is not only interested in the possible transmission rate itself but has to configure his/her exploit instance properly to make it achieve optimal or at least close to optimal transmission rates when run on a target system. Table 3.2 on page 34 has already shown that the value of even a single exploit parameter can have a significant influence on the performance of the exploit. Therefore, in the following section a generic framework is developed that describes the interdependencies between (a) the exploit and its configuration, (b) the system, and (c) the loss rate and the possibility of misaccountings. This framework thereby on the one hand deepens the understanding about IRCCs, and on the other hand allows for finding suitable or even optimal configurations. The benefit of this framework will be demonstrated by applying it to a concrete system, namely the testing environment of Section 1.5.

In order to make propositions about good choices for the values of exploit parameters at all, the system model has to be refined so that it also considers events on a time scale below the length of timeslices. This refinement is illustrated in Figure 3.3, which focuses on a single receiver timeslice from the second bit – a **1** – of Figure 3.1. The figure adds

[10]The bit error rates for Hamming codes have been computed according to Section A.3.

two new, fine-grained aspects to the IRCC transmission: Timer interrupts and a variable delay from the generation of an interrupt in the sender until its later occurrence. In this example, there are four timer interrupts during the receiver's timeslice, of which the last triggers a task switch, here back to the sender. The variable interrupt delay is indicated by two imaginary probability density distributions at the bottom of the figure. To begin with, both new aspects and reasons for their relevance to the exploit will be taken a closer look at. Afterwards also a third, resultant aspect is highlighted.

Timers. Modern, multitasking operating systems require for timer interrupts in order to gain control about the system from time to time and then do routine work like, e.g., scheduling another task. Current Linux kernels like the one used for this thesis allow the timer frequency being set to 100 Hz, 250 Hz, 300 Hz or 1000 Hz.[11] Assuming a timeslice length of 100 ms, this means 10 to 100 timer interrupts within a single timeslice and, thus, the mere possibility of a "collision" of timer and sender-initiated interrupts cannot generally be neglected. In this context, the *collision of interrupts* denotes the occurrence of one interrupt during the execution of another interrupt's handler.

An important detail to note is the times at which the switch from the sender to the receiver process happens: Immediately after a timer interrupt. This is the result of the fixed quantum scheduling (precisely, the code of Listing 2.6d on page 21), which treats the sender like a CPU-hog consuming up the full timeslice at its disposal. Since the sender generates its interrupts relative to the end of its timeslice, i.e. relative to the timer interrupt responsible for the task switch, the occurrence of these interrupts is *not* temporally independent of the occurrence of timer interrupts. In other words, timer interrupts always occur at rather fixed positions compared to the occurrence distribution of the sender-initiated interrupts.

Interrupt delay. Usually, an interrupt is raised by a device to indicate the completion of an action to the operating system. In case of the exploit, the action is initiated by the sender calling a system function. Depending on the device (NIC, hard disk, etc.) and its current status (position of mechanical parts, sleep state, etc.), carrying out the action may take more or less time. Hence, for the same action demanded from a device, the delay until the occurrence of the completion interrupt is not a constant but is subject to a probability distribution. Such a distribution might resemble one of the two imaginary examples of Figure 3.3 but may as well be more complex, containing multiple maxima or being rather discontinuous.

A necessary condition for an IRCC to be possible is that at least a nonzero fraction of all sender-initiated interrupts occurs while the receiver is executing, since the remainder likely cannot be detected by the receiver. Nevertheless not only the receivable fraction but also its delay distribution is of importance, because it has an influence on how often the sender-initiated interrupts collide with or occur close to timer interrupts. Given a very narrow delay distribution it could be that either the sender's interrupts *always* collide with a timer interrupt, or they *always* do not. Figure 3.3 depicts an example of such a distribution with a single, narrow peak shortly after a timer interrupt. In this case, collisions would never occur. But if the peak was positioned slightly earlier in the figure, then its majority would overlap with the timer and, thus, result a high likelihood of collisions. In contrast, the broader

[11] "Dynamic ticks", which are available since Linux 2.6.21, are not considered in this work.

distribution shown in the figure could be moved along the time axis without much changes in the probability of collisions between timer interrupts and sender-initiated interrupts.

It has to be noted that the delay distribution is not fixed with the choice of the interrupt source but may also depend on parameters that are – directly or indirectly – passed to the interrupt generating system function. In case of the NIC as the interrupt source, sending larger packets tends to cause longer delays; in case of a hard disk, one might try to alter between reading data blocks from inner and outer cylinders to elongate the delay.

Interrupt duration. After an interrupt request is signaled to the CPU, the currently running task is interrupted and execution is switched to an interrupt handler, which then tries to find out the reason for the interruption, handles it and afterwards switches back to the initially interrupted task.[12] The duration of the interruption is therefore determined by the amount of kernel and device driver code being executed. The receiver should know the range of possible durations of the chosen interrupt source as good as possible in order to make its decision of whether an interruption is sender-initiated or not as reliable as possible. Nevertheless it may in principle happen that the durations of other interruptions, like e.g. a timer interrupt, intersect with the durations of sender-initiated interrupts. Since however the current model does not allow for this situation (it would result in $p(Y{>}0|X{=}\mathbf{0}) \neq 0$), the framework does also not include the purpose of optimizing the configuration of the receiver's interrupt classification parameters. At the same time this avoids the requirement of knowing duration distributions for noise which in turn are expected to be harder to determine than the durations of the sender-generated interrupt. Section 3.4 will give a practically oriented justification for this choice.

Skew and gaps. Both exploit processes query the clock to determine the current bit index temporally close to the generation and, respectively, reception of an interrupt. Listings 2.8 and 2.9 on page 23 indicate how this is realized in the concrete implementation. Due to the interrupts' delay and duration, the detection happens some time after the generation, but nevertheless both events should be accounted to the same bit index. To achieve this, bit intervals in the receiver are shifted into the future by some amount which will be referred to as the *bit interval skew*.

If the interrupt delay always was a constant, then a correctly chosen skew would suffice for preventing misaccountings. But as explained above, this is not necessarily the case and therefore, no matter what bit interval skew has been chosen, interrupts may still be accounted to the wrong bit index. If an interrupt is generated shortly *before* the end of one (sender's) bit interval and the delay by coincidence is longer than the chosen skew, then the interrupt is detected within the (receiver's) following bit's interval. On the contrary, if the interrupt is generated shortly *after* the beginning of a new bit interval and the interrupt delay is shorter than the skew, then the interruption is accounted to the preceding bit index. This leads to a situation where the input belonging to one bit index influences the output belonging to another – the immediately succeeding or preceding – bit index. Hence, such a channel must not be considered memoryless unless misaccounting is very rare. The chosen solution to the problem is a *gap* in the receiver between every pair of subsequent bit intervals,

[12]The operating system may as well decide to schedule another task based on potentially changed conditions resulting from the interrupt, like some high priority task previously waiting for data being unblocked.

such that all interruptions detected during this gap are simply ignored. Obviously the gap has to be chosen long enough to "absorb" the variation of the interrupt delay.

Instead of ignoring interrupts that were detected during a gap, the receiver task could as well store this information separately in order to not lose as much information. In the later evaluation of the output, the attacker might then be able to infer[13] or guess to which bit interval the interrupt should have belonged.

The above considerations depend very much on *how short* the temporal distance between the interrupt generation (which is preceded by a clock query) and the bit interval boundaries can become. Shorter distances potentially increase the likelihood of misaccountings and therefore demand for longer gaps and a precisely setup skew. Figure 3.1 on page 29 simplifyingly omits a bit interval skew and shows all bit interval boundaries at times where a task switch from the receiver to the sender happens. As a consequence of the latter, the sender queries the clock at least approximately t_s after a bit interval boundary, and at least approximately t_r before. Misaccountings in this model could therefore almost be excluded, since the variation of the interrupt delay is not expected to exceed the timeslice lengths t_r and t_s. In practice, however, it is, if possible at all, not trivial to guarantee that bit intervals and timeslices adhere to suchlike synchrony. Even if bit interval boundaries are placed at a switch from the sender to the receiver at the beginning of the transmission, the distance between the bit interval boundaries and the sender's clock queries may vary and, in particular, shrink during the transmission. Two apparent reasons[14] to be examined below are (a) other tasks which wake up during the transmission, and (b) a bit interval length which is not an exact integral multiple of $t_s + t_r$.

From a realistic setting, other tasks than the two exploit processes cannot be excluded. There are always at least also kernel tasks and system processes (like `init`), which usually are not all asleep for the whole time. Thus, once such a task wakes up, it will finally get scheduled and thereby shift all following timeslices of the exploit processes into the future by some amount of time. By construction, the bit intervals remain unaffected, as they are fixed in time and *not* in the number of passed sender/receiver timeslice pairs.[15]

The previously described effect is only able to cause shifts which are integral multiples of the timer interval, because timeslice lengths are not determined by polling the clock but by counting timer interrupts[16] which Figure 3.3 on page 36 illustrates. A sophisticated exploit implementation could use this property of the scheduler and guarantee that bit interval boundaries keep at least a certain minimum distance (e.g., half a timer interval in both directions) to the sender's clock queries, for instance by determining when timer interrupts occur already prior to the transmission and placing bit interval boundaries accordingly. Thus, even though the execution of other tasks reduces the minimum distance from the scale of timeslice lengths to the length of timer intervals, a reduction arbitrarily close to zero is impossible, independent of the execution times of these tasks.

This minimum distance may completely vanish if bit interval lengths are not exact integral

[13]If N interrupts were already detected for one bit, then another interrupt occurring during the preceding or succeeding gap must belong to the previous or, respectively, next bit interval.

[14]Further causes may be possible but are expected to be at least less significant.

[15]An implementation based on timeslice pairs would be harder to realize (the timeslice notion would have to be implemented into the receiver) and potentially more susceptible to misaccountings due to disturbances by other tasks.

[16]This is the result of a design choice of the (original) scheduler, which could instead also setup timers such that they occur only at times when they are known to be needed.

multiples of t_s+t_r, because then a permanent drift between timeslices and bit intervals takes place. Over the course of a transmission, the single drifts of every bit, even if they are small, quickly accumulate and destroy the minimum distance of the last paragraph. Even small deviations from the actual timeslice pair length $t_s + t_r$ of, say, 0.01% add up to 10 ms (i.e. one timer interval at 100 Hz timer frequency) within a transmission of only 100 s. Thereby the bit interval boundaries might be taken almost arbitrarily close to the sender's clock queries. A guarantee like the one proposed in the previous paragraph would under these circumstances simply be impossible.

Earlier in this chapter, $t_b = N(t_s + t_r)$ for $N \in \mathbb{N}$ was introduced as a requirement for the formal model and, thus, one might think that the mentioned drifts cannot occur as long as the exploit adheres to this requirement. In practice the problem can nevertheless arise since the exploit might only be able to determine $t_s + t_r$ approximately and would then be unable to deliberately choose t_b as an integral multiple. The later study of the concrete instance (in Section 3.3.2) will demonstrate that accurately determining the actual timeslice lengths is everything else but trivial.

3.3.1. A Generic Framework

For an attacker, the right choice of exploit parameter values is important as it influences the likelihood of interrupt losses as well as of interrupt misaccountings. Thus, these both aspects of the IRCC are the subject of the generic framework: Interrupt losses are to be minimized and interrupt misaccountings are to be avoided at all. Although both aspects are related – longer inter-bit gaps for instance may increase the loss rate – they are examined in isolation in separate parts of the framework for the sole purpose of an easier understanding. The existing interdependency of both parts must however be kept in mind when the framework is applied to a concrete system.

Both parts first identify a set of relevant system parameters, like e.g. the interrupt delay or the timer frequency. However, due to the complexity of the overall system, including hardware and software, no claim is raised that the set of parameters is complete. Other influences may still exist but are at least expected to have only a minor impact on the choice of exploit parameter values.

Often these parameters are not constants but probability distributions (over time), which for the sake of a simple formalism are then assumed to be continuous and integrable.[17] Furthermore the framework relies on the assumption that these system parameters do not change over time, which implies also independence from the duration of channel usage (a requirement for memorylessness).

Besides the system parameters, each part of the framework is concerned with a set of exploit parameters whose values need to be determined for running the exploit. The distinction between system and exploit parameters is not always unambiguous. For example the interrupt delay (a putative system parameter) may strongly depend on the parameters to the interrupt-initiating function (exploit parameters). In case of a NIC-based channel, this especially applies to the packet length, which determines the time required to send out the packet over the cable before the completion interrupt is raised.

[17] Wherever a distribution turns out to be discrete and has to be treated as such, necessary changes to the framework are expected to be only of rather symbolic nature (like substituting "$\int$" by "$\sum$").

Interrupt Losses

The reason for an interrupt loss that probably comes to mind first is that another process than the receiver is executing while the interrupt occurs. This problem however is hard to influence from within the exploit, as the kernel is in principle free to choose whether other ready tasks are scheduled between sender and receiver. More interesting for the configuration of the exploit are those interrupts that go astray *even though* they occur while the receiver is executing. Their likelihood is determined by the configuration of the exploit and is, thus, subject of the framework.

Before coming back to the explanation of why the receiver may lose interrupts, first the underlying system parameters and their symbols are introduced, beginning with those related to timers. As a convention, the call of the interrupt-initiating operation happens at time $t = 0$. The first ensuing timer interrupt of interest is the one that causes the task switch from the sender to the receiver, at the yet unspecified time $\tau_0 > 0$. From then on, all following timer interrupts take place with constant frequency f at points in time $\tau_i = \tau_0 + i \cdot T$, where $T := f^{-1}$ is the *timer interval*. In addition to a starting time, every timer interrupt with index i also entails the execution of an interrupt handler with a duration c_i. The distinction between the different *timer interrupt durations* c_i is primarily introduced because some interrupts may be known to take longer or shorter than others. For example, c_0 includes the overhead for the task switch, while all other c_i do not.[18]

In favor of a more concise notation, values like c_i or τ_i are treated as constants instead of results of stochastic processes. This is based on the assumption that even if the values should vary, then only with a very small deviation from a certain mean. The same also applies to many of the values to be introduced below. Nevertheless, this is not to be considered a limitation of the framework, since the framework is more a conceptional than a formal guide. Thus, the user of the framework can finally choose whether constants are appropriate or whether it is necessary to work with discrete or continuous distributions for obtaining accurate results.

The second group of system parameters contains those related to the interrupts that are generated by the sender. At first, the *interrupt generation* itself takes some time t_g, consisting of the runtime of kernel code and the time to communicate the respective request to the device. After the device has carried out the request, it raises the desired interrupt. The duration from the beginning of the interrupt generation until the occurrence of the interrupt, the *interrupt delay*, is referred to as D. In contrast to all other variables in this part of the framework, this delay is explicitly treated as a probability distribution (i.e. $D(t) \geq 0$ for all t, and $\int_0^\infty D(t)\,dt = 1$). This choice has been made since, depending on the chosen interrupt source for the channel, interrupt delays might strongly vary (as already mentioned in the discussion corresponding to Figure 3.3 on page 36). Finally, the *duration* d of the interrupt handler completes the list of the sender-initiated interrupt's properties. All three of these interrupt-related system parameters may actually be influenced by varying the parameters to the interrupt-generating system call, but this relation is not explicitly further covered by the framework.

Yet another category of parameters are those that depend on the implementation of the exploit but cannot be freely chosen. For this part of the framework this is only the *evaluation duration* t_e which corresponds to the (uninterrupted) execution times of `in_relevant_range (dt)`

[18]The task switch from the receiver back to the sender is considered to be temporally too far away from the interrupt generation to be relevant.

f: Timer interrupt frequency.
T: Timer interrupt interval.
τ_i: Time of the i-th timer interrupt, relative to the interrupt generation.
c_i: Duration of the i-th timer interrupt.

t_g: Duration of the interrupt-initiating operation.
D: Delay from generation to occurrence of sender-initiated interrupts.
d: Duration of sender-initiated interrupts.

t_e: Runtime of the evaluation of a measuring in the receiver.

ε: Temporal pre-position of the interrupt generation wrt. the end of the sender's timeslice.
t_m: Runtime of an uninterrupted measuring in the receiver.

Table 3.3. Framework parameters related to interrupt losses.

and `transmission_finished ()` in Listing 2.9 on page 23. More precisely, t_e represents only the evaluation in case of a negative result, as otherwise it would already be known that the interrupt is *not* lost. Thus, especially the determination of the bit index and the logging of the detected interrupt in the positive case are not included.

Finally there are the actual exploit parameters. In case of the sender this is the amount of time that the generation of the interrupts is started before the end of the respective timeslice, which, in absence of another concise term, is called the *(temporal) pre-position* ε. Listing 2.8 on page 23 shows this parameter as `epsilon`. An attacker should choose ε such that $\varepsilon \geq t_g$ as this ensures that the interrupt generation completely fits into the timeslice which it is started in. Otherwise, if the condition was violated, the interrupt-initiating request might be delivered to the hardware device at the beginning of a new timeslice, during which then in all likelihood also the completion interrupt would occur. Parameter ε determines the temporal coordinates of the timer interrupts through $\tau_0 = \varepsilon$, since at τ_0 the sender's timeslice ends and the task switch to the receiver is performed.

The receiver comprises a parameter that influences the loss rate, too. This is the *measuring duration* t_m, which corresponds to the (uninterrupted) execution time of `waste( iterations )` in Listing 2.2 on page 13. The choice of `iterations` fixes the number of times that a small portion of code is repeated in a loop and therefore indirectly determines the value of t_m.[19]

Table 3.3 presents an overview with a short summary for all the above system and exploit parameters of the framework. The interrupt loss probability can now be expressed as

$$\lambda = \int D(t) \cdot l(t)\, dt, \tag{3.3}$$

where $l(t)$ is the yet unknown probability of losing the interrupt if it occurs at time t, relative to the time of interrupt generation. For the further analysis of $l(t)$, the temporal axis is divided into five disjoint ranges, (i) to (v), as illustrated in Figure 3.4. These correspond to different causes for the interrupt not being detected by the receiver. The figure visualizes how the ranges result from the various system and exploit parameters. Ranges (iii) to (v) consist of several subranges, which repeat for every timer interval the receiver's timeslice, as

[19] Note that, especially for small `iterations`, the relation between t_m and `iterations` is not necessarily linear as one might expect.

Figure 3.4. Ranges of different collision and loss probabilities.

the "..." in the figure indicates. The following enumeration presents the formal definition of the different ranges and explains how these ranges influence the loss probability $l(t)$ at that point in time.

(i) $0 \leq t < \tau_0 - d$. The sender-initiated interrupt occurs during the sender's timeslice and early enough to not collide with the following timer interrupt.

It is expected that interrupt occurrences during this range are hard or even impossible to detect. At least for the given exploit implementation, this range implies $l(t) = 1$ for all times t within the range.

(ii) $\tau_0 - d \leq t < \tau_0 + c_0$. The sender-initiated interrupt collides with the timer interrupt responsible for the task switch to the receiver.

Due to the collision, the receiver cannot detect the interruption resulting from the sender-initiated interrupt by itself. It may however be able to detect the induced elongation of the time span between the last switch away from the receiver back to the receiver again. If not, then $l(t) = 1$.

(iii) $\tau_i - d \leq t \leq \tau_i + c_i$ for some $i > 0$. This range corresponds to (ii) but for a timer interrupt fully within the receiver's timeslice.

The detection of an interrupt occurring within this range is potentially simpler than it is within the previous range, since here the interruption consists only of two interrupts (with approximate duration $d + c_i$) but no task switch. Again, if the receiver is unable (or unwilling[20]) to detect this kind of collisions, $l(t) = 1$ holds.

(iv) $\tau_i + c_i < t < \tau_i + c_i + t_m$ or $\tau_{i+1} - t_m < t + d < \tau_{i+1}$ for $i \geq 0$. A collision between sender-initiated and timer interrupts is not possible but the receiver may nevertheless measure both interrupts within the same measuring phase (if no evaluation phase falls between both interrupts). For the receiver this would then resemble a collision, but will here be referred to as a *compound measuring*.

In the very first subrange ($\tau_0 + c_0 < t < \tau_0 + c_0 + t_m$), a compound measuring would be as hard to identify as in range (ii). Analogously, compound measurings in all other subranges are related to range (iii).

[20]The receiver could deliberately ignore interruption durations $d + c_i$ e.g if there are other events resulting in interruptions of this duration.

(v) Otherwise the sender-initiated interrupt does neither collide with a timer interrupt or task switch, nor will it be measured together with one. Thus, unless the receiver explicitly favors collisions or compound measurings, the exploit should be configured that sender-initiated interrupts occur within this range.

Even though the intention of the above ranges is to facilitate the determination of loss probabilities $l(t)$, concrete values or formulas have intentionally been omitted in their explanations. This is because unless it is known that the receiver cannot detect interrupts occurring within a certain range (i.e. $l(t) = 1$), the loss probability is determined by (a) the probability that the interrupt occurs during an evaluation phase, (b) the likelihood of a collision or compound measuring with other events, like sporadic interrupts or other tasks awaken after such an interrupt, and (c) the probability for a sender-initiated interrupt to occur during an inter-bit gap. All these influences are not explicitly modeled here but instead only discussed. The first cause for interrupt losses alone might lead to a loss probability of $t_e/(t_m + t_e)$, if occurrences are uniformly distributed on the measuring/evaluation cycles of length $t_m + t_e$. Other distributions could as well lead to interrupts either being never or being always lost due to occurrences within an evaluation phase. The distribution of the "other events" in the second cause for interrupt losses is similarly decisive, but potentially more complex. A simple example will be presented later in the concrete instance. Finally, the gaps as the last listed cause for interrupt losses form the connection between this first part of the framework with the part dealing with interrupt misaccountings.

The aim of the attacker should now be to select values of ε and t_m that make all or at least most of the sender-initiated interrupts occur during range (v) within a measuring phase.

Interrupt Misaccountings

Earlier explanations already showed why interrupts may be misaccounted and what can be incorporated into the exploit (and hence the model) to avoid this: A *bit interval skew* σ and an *inter-bit gap* γ. The skew denotes the temporal distance between the sender's and the receiver's bit interval boundaries. For instance, $\sigma = 1\,\text{ms}$ means that the bit intervals as seen by the receiver begin and end 1 ms after they begin and, respectively, end in the notion of the sender. By convention, one half of the gap takes place before every receiver's bit interval boundary and the other half afterwards. Thus, for every a interval boundary at time t there is a gap during the interval $[t - \gamma/2, t + \gamma/2]$.

This part of the framework again introduces additional system parameters. To begin with, there is the temporal distance between the sender's clock queries (shortly before the interrupt is generated if the bit is determined to be a **1**) and the succeeding bit interval boundary. During long transmissions, this distance may vary for reasons that have already been discussed, even though not completely independent from prior distances. Nevertheless, the distance is modeled as a probability distribution B, defined on interval $[0, t_b)$ (i.e. $\int_0^{t_b} B(t)\,dt = 1$), under the assumption of a reasonably long transmission.[21] The intention of this distribution is to constitute a generalized notion of the minimum temporal distance, which has already been motivated in the introduction to the framework. Its shape is partly determined by the system, through the scheduling of other tasks or variations of the timer

[21] This is again a situation where a more sophisticated exploit implementation could explicitly note prior distances and continuously adapt the value of σ based on this knowledge.

B: Probability distribution for the temporal distance between clock queries of the sender and bit interval boundaries during a long transmission.
M: Probability distribution for the temporal distance between the occurrence of a sender-initiated interrupt and the end of the measuring phase it occurs in.
Z: Random variable representing the delay from the interrupt generation by the sender until its detection in the receiver.
σ: Temporal skew between sender's and receiver's bit interval notions.
γ: Duration of the gap between bit intervals in the receiver.

Table 3.4. Framework parameters related to interrupt misaccountings.

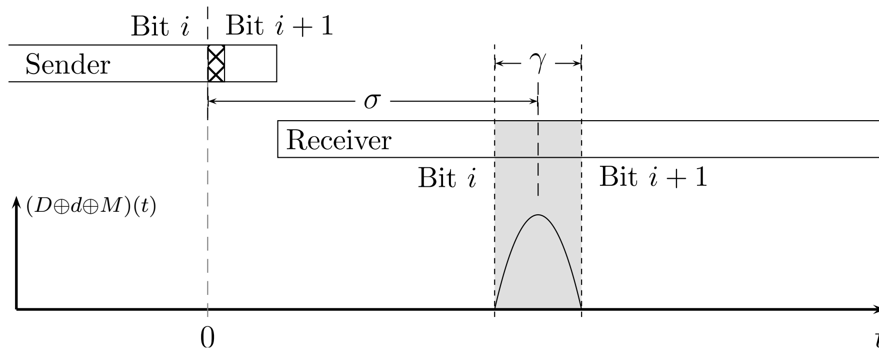


Figure 3.5. Bit intervals from the perspective of sender and receiver, if sending may happen arbitrarily close to bit interval boundaries.

frequency, but also determined by the exploit and, for example, its accuracy in the choice of t_b as an integral multiple of $t_s + t_r$.

The second and last system parameter for this part of the framework is the probability distribution M, defined on the interval $[0, t_m)$ (i.e. $\int_0^{t_m} M(t)\,dt = 1$). Every $M(t)$ gives the (conditional) probability density of an interrupt occurring with temporal distance t to the end of the surrounding measuring phase (given that it occurs during a measuring phase at all[22]). In other words, such a distance t represents the amount of time that passes from the occurrence of the interrupt until the receiver starts evaluating whether there was an interruption. This distribution is of interest because longer measuring durations t_m potentially increase the uncertainty about when inside the measuring interval the interrupt actually occurred. A higher uncertainty in turn may require for a longer gap to avoid misaccounted interrupts. Also the skew may have to be chosen higher due to the increased average delay until the evaluation. It has to be remarked that M depends not only on t_m but also on D, if the measuring intervals have fixed offsets within the receiver's timeslices.

An overview of the additional parameters introduced for capturing interrupt misaccountings is given in Table 3.4.

The aims of the adjustment of σ and γ are (a) making misaccounted interrupts impossible or at least sufficiently unlikely, and (b) minimizing the probability of detecting interrupts during the gap (i.e. avoiding unnecessarily large gaps). Therefore, let Z be a random variable indicating the duration from the time of interrupt generation, until a sender-initiated interrupt is detected, if it is detected at all. It is distributed according to the convolution

[22] Lost interrupts cannot be misaccounted!

of D, d and M, written here as $D \oplus d \oplus M$.[23] Unlike in the first part of the framework, the interrupt duration d is here formally treated as a probability density function for the sole reason that it does not complicate the notation.

Figure 3.5 illustrates the problem of interrupt misaccountings and visualizes the relevant parameters (except for B). At the very beginning, on the left of the figure, the sender is executing and is still in its i-th bit interval. Close to the end of its timeslice the sender has to query the clock, compute the bit index at the obtained time and, depending on the bit's value, immediately afterwards either generate an interrupt (as it is shown in the figure) or not. The figure now depicts the situation where the transition to the $(i+1)$-th bit interval coincides with the time at which the sender queries the clock for the bit index, such that it is unclear to which bit interval the generated interrupt belongs. Now the problem is how the receiver should treat such a borderline interrupt. The figure shows an imaginary distribution for Z, which indicates when and with which probability (density) the receiver is expected to detect the interrupt. Thus, the grayed range corresponds to an interval during which sender-initiated interrupts can occur, which the receiver cannot surely assign to either bit index, i or $i+1$. This range should therefore be declared as the gap. All interrupts occurring before can correctly be attributed to bit i, all interrupts occurring afterwards must belong to bit $i+1$. To achieve this, the skew σ should be chosen to shift the receiver's bit intervals into the future by the right amount, as the figure shows.

The figure actually considers the "worst case" where $B(0) > 0$ and sender's clock queries and bit interval boundaries may thus coincide. Nevertheless the width of the support of Z's distribution, i.e. the variability of the interrupt detection time, might demand for a gap also if the minimal temporal distance between the clock queries and interval boundaries is nonzero but small enough.

Given Z, the requirement of *not* misaccounting an interrupt can be formulated as

$$\int_0^{t_b} B(\delta) \cdot p(-(t_b - \delta) + \sigma - \gamma/2 \leq Z \leq \delta + \sigma + \gamma/2)\, d\delta \stackrel{!}{=} 1, \tag{3.4}$$

where the lower boundary for Z is the beginning of the previous gap and the upper bound is the end of the following gap. The variable of integration, δ, ranges over all possible distances to bit interval boundaries. Analogously, the probability of a correct reception *between* (excluding) the gaps can be expressed as

$$\xi := \int_0^{t_b} B(\delta) \cdot p(-(t_b - \delta) + \sigma + \gamma/2 < Z < \delta + \sigma - \gamma/2)\, d\delta. \tag{3.5}$$

Combining losses and misaccountings. If a gap is determined to be necessary for satisfying Equation (3.4), then this gap must also be considered in the first part of the framework. Since any detected interrupt that occurred during a gap has to be ignored by the receiver, this situation effectively equals an interrupt loss. Its probability $1 - \xi$ has to be considered when computing the loss rates $l(t)$. The interdependency between both parts can be illustrated by how different terms influence ("→") each other:

$$t_m \quad \rightarrow \quad M \quad \rightarrow \quad \gamma \quad \rightarrow \quad \xi \quad \rightarrow \quad l(t) \quad \rightarrow \quad t_m.$$

[23] If the receiver successfully detects the desired interrupts even in case of collisions with other events, the distribution becomes more complicated.

		Samples	Mean	Std.dev.	Minimum	Maximum
RTL8029	Generation	20000	165.05 μs	1.042 μs	164.22 μs	218.25 μs
	Delay	12543	1326.02 μs	0.694 μs	1325.22 μs	1379.19 μs
	Duration	12543	15.19 μs	0.196 μs	15.02 μs	19.20 μs
RTL8139	Generation	20000	7.73 μs	1.097 μs	7.25 μs	81.05 μs
	Delay	12525	125.61 μs	0.767 μs	125.30 μs	199.01 μs
	Duration	12525	8.28 μs	0.194 μs	8.10 μs	10.34 μs

Table 3.5. RTL8029 (10 Mbit/s NIC controller) and RTL8139 (100 Mbit/s NIC controller), sending broadcast UDP packets with 1400 bytes payload.

Task switch		Samples	Mean	Std.dev.	Minimum	Maximum
No	Kernel	12000	5.977 μs	0.332 μs	5.676 μs	19.323 μs
	Userspace	25000	6.602 μs	0.302 μs	6.312 μs	7.781 μs
Yes	Userspace	27614	7.694 μs	0.201 μs	7.218 μs	8.887 μs

Table 3.6. Timer interrupt durations with and without task switch.

3.3.2. A Concrete Instance

In order to demonstrate how the generic framework can be used to determine a configuration tailored to a given concrete target system, it is applied to the author's testing environment as introduced in Section 1.5. Even though the computations that will follow are based on assumptions about the target system, large parts have been kept as generic as possible. This has the intention that these parts can easily be adopted also for other targets with similar properties.

Firstly the exploit is implemented to use the NIC as the source device for interrupts by sending broadcast UDP packets. Table 3.5 shows the empirically determined properties of the distributions of t_g, D and d for two NIC chips and 1400 bytes of payload. The amount of payload for the experiments was chosen relatively close to the maximum transmittable with a single Ethernet frame. A larger choice would finally have led to multiple packets being sent and multiple interrupts being generated and would thereby violate the model.[24] Smaller choices intuitively tend to reduce the interrupt delay (less has to be sent) and would thereby only restrict the sender to perform the interrupt generation closer to the end of the timeslices.

The generation time t_g was measured in userspace by taking the time immediately before and after the call to the interrupt-initiating function. The first of these times was also used for the collection of the delay D, whose end time however was measured in the kernel at the beginning of the interrupt handler. Finally, the interrupt duration d was completely measured in the kernel at both ends of the interrupt handler. In general, kernel-measured interruption durations tend to be lower bounds, since they do not embrace the switch from user to kernel mode as well as assembly code to enter and leave the interrupt handler. On the other hand, userspace-measured durations of interruptions are upper bounds, because a task cannot take the time exactly at the point in time when an interruption occurs or when an interrupt handler finishes.

That it is not necessary to detect and measure interrupts from within the kernel is shown

[24] Compare Section 3.4.3 for a discussion of this idea.

for the timer interrupts in Table 3.6. The approximate durations of timer interrupts were first manually identified in a histogram of interruption durations (which was easily collected by a slightly modified receiver application). Then a specialized program, also akin to the exploit's receiver, was used to conduct more precise measurings of only the timer interrupts. For the duration of task switches, two processes were started to collect "interruptions" of the length of a timeslice, caused by the respective other process. The outcomes of both processes were then combined to obtain the wanted values (as the difference between the beginning of an interruption of one process and the succeeding ending of an interruption of the other process).

The clock frequency was set in the kernel configuration to $f = 100\,\mathrm{Hz}$, giving $T = 10\,\mathrm{ms}$. It will later be discovered that this value is slightly larger than the actual timer frequency, but for now the approximate value suffices.

Finally there is the evaluation time t_e, which corresponds to the runtime of a very small amount of code to evaluate the outcome of the measuring phase. This particularly includes the code for the comparison of the measured duration against the lower and upper bound of the measuring range. However, also function calls or other conditional and unconditional jumps may add up to the runtime. In order to at least get an approximation of t_e, the core of the evaluation code was copied to a dedicated measuring program and complemented with runtime measuring code. Although the result of $t_e = 27\,\mathrm{ns}$ was measured with only a very small standard deviation, it is unclear what impact the extraction of the evaluation code into a separate program and the addition of time measuring code have on the runtime to be determined. The above value of t_e should therefore only be seen as a rough guide for the dimension of the evaluation duration.

Interrupt Losses

Due to the mostly small deviations from the mean, all interrupt properties are simplified to constants assuming the respective mean value: $t_g = 165.05\,\mu\mathrm{s}$, $d = 15.19\,\mu\mathrm{s}$ and $D = 1326.02\,\mu\mathrm{s}$ for the NIC interrupts;[25] $c_0 = 7.709\,\mu\mathrm{s}$ and $c_i = 6.602\,\mu\mathrm{s}$ for $i > 0$ as timer interrupts are concerned. This spares the reader the introduction of, e.g., support intervals and especially simplifies Equation (3.3) of page 42 to $\lambda = l(D)$, where $l(D)$ is determined in the following. At the end, when concrete suggestions for the exploit parameter values have been found, it has to be taken into account that slight corrections may be necessary to respect also the deviations of system parameters from their mean values.

The primary goal here is not to determine λ, but to minimize it by adjusting exploit parameters t_m and ε. The interrupt delay distribution has only a single, narrow peak, which can be moved relative to the timer interrupts with the parameter ε. If possible, then it should be set to a value that makes the interrupt not fall into the collision ranges (i), (ii) or (iii), since these either disallow the detection at all or at least complicate it. Range (i) would require $D < \varepsilon - d$, so $\varepsilon \leq D + d$ suffices to exclude the possibility of a sender-initiated interrupt occurring within this range. Similarly, range (ii) can be eliminated by reducing ε's upper bound further to $D - c_0$. Range (iii), i.e. a collision with a timer interrupt after the task switch, is excluded already for the first one due to the short interrupt delay D (since $\varepsilon \leq D + d - T = -8658.79\,\mu\mathrm{s}$ cannot be satisfied).

The short interrupt delay D effectuates all sender-initiated interrupts occurring before τ_1,

[25] Formally one might want to define $D(t) = \delta(t - D)$ where δ is the Dirac delta function.

independent of how ε is chosen. Therefore, all later subranges of (iv) and (v) can henceforth be excluded from the considerations. However, their remaining, earlier subranges (before τ_1) still depend on t_m, which therefore is considered next.

The measuring duration t_m influences the probability of compound measurings as well as the likelihood of interrupt losses due to the receiver evaluating instead of measuring. If it is assumed that (a) the interrupt can occur only during the ranges (iv) and (v), i.e. $\varepsilon \le D - c_0$, and (b) occurrences of sender-initiated interrupts are uniformly distributed on the measuring/evaluation phases, then the probability $\bar{\lambda} := 1 - l(D)$ of successfully detecting an occurring interrupt can be computed as follows:

$$\bar{\lambda}(t_m, \varepsilon) = \begin{cases} \dfrac{t_m}{t_m + t_e} & \text{if } t_m \le D - c_0 - \varepsilon, \\ \dfrac{D - c_0 - \varepsilon}{t_m + t_e} & \text{if } D - c_0 - \varepsilon < t_m \le T + \varepsilon - D - d, \\ \dfrac{T - c_0 - d - t_m}{t_m + t_e} & \text{if } T + \varepsilon - D - d < t_m \le T - c_0 - d, \\ 0 & \text{otherwise } (t_m > T - c_0 - d). \end{cases} \tag{3.6}$$

This case distinction only holds if $D - c_0 - \varepsilon < T + \varepsilon - D - d$, which expresses that the interrupt occurs in the first half of the interval between the two timers (the inverse case would be analogous). The first case corresponds to collision range (v), where compound measurings are impossible and $\bar{\lambda}$ therefore equals the probability of the receiver executing the measuring code. All remaining cases belong to collision range (iv): In the second case, t_m is small enough to only allow for compound measurings with the previous task switch; case 3 handles those cases where compound measurings are additionally possible together with the following timer interrupt; and in the last case, t_m is large enough to always produce compound measurings.

One can easily satisfy oneself that $\bar{\lambda}$ is continuous with respect to t_m. For every fixed ε within its permitted interval, the maximum of $\bar{\lambda}$ with respect to t_m is located at $t_m = D - c_0 - \varepsilon$, since each case's corresponding function is monotonic but only the first one is not monotonically decreasing. Another way of writing the maximum is $t_m + \varepsilon = D - c_0$, which shows that the sum of both adjustable parameters (t_m and ε) is constant. To further maximize $\bar{\lambda}$ over all allowed values of ε, the strict increasing monotonicity of $t_m/(t_m + t_e)$ with respect to t_m suggests to maximize t_m or, equivalently, minimize ε.

Concluding, to minimize the interrupt loss probability, ε should be chosen minimal with $\varepsilon \approx t_g$ in order to make the interrupt occur as close to the middle of the inter-timer interval as possible, and for t_m the maximal time should be chosen that does not yet cause compound measurings. Hence $\varepsilon \approx 165\,\mu$s and $t_m \approx 1153\,\mu$s would be the recommendation obtained with the framework. Table 3.2 on page 34 shows some interesting experimental results obtained with different values of t_m. The temporal pre-position for all experiments was chosen as $\varepsilon = t_g + 40\,\mu$s to be a bit more robust against the variations in the interrupt distributions and inaccuracies of the timing. The increase of ε leads to a decrease of t_m by the same amount, as the sum of both has to be constant for the maximum of $\bar{\lambda}$. Thus, Setting 5 of the table would (approximately) correspond to the framework's suggestion. Nevertheless the table shows that this t_m does not yield the best results (though also not the worst) and should rather be chosen smaller, like in Setting 3 or 4. In the following, one possible cause for the suboptimal predictions will be factored into the computation:

Additional interruptions. Before, however, it is demonstrated that the framework is, for instance, able to explain the results of Settings 1 and 6 of the experiments.

In Setting 1, the measuring duration $t_m \approx 0.183\,\mu$s was chosen, which clearly belongs to the first case of Equation (3.6). Given t_e as above, one easily obtains $1-\bar{\lambda} = t_e/(t_e+t_m) \approx 12.9\%$ – close to the experimentally determined 14.1% and 14.3%. Similarly, for Setting 6 with $t_m \approx 1275\,\mu$s one obtains, when applying the second case of Equation (3.6), that $1-\bar{\lambda} \approx 12.7\%$ – right in the middle of the experimental results.

Additional interrupts. In Equation (3.6), $\bar{\lambda}$ considers only simple losses of the sender-initiated interrupts and compound measurings with timers or task switches. However one might also think of additional other, randomly occurring interruptions that are responsible for compound measurings: Another process may wake up for a short time[26] or a key press by a user could trigger an interrupt. Firstly, to stay within the model, it is assumed that all the additional interruptions cannot be confused with the sender-initiated interrupts, for instance because their durations differ.[27] This preserves $p(Y{>}0|X{=}\mathbf{0}) = 0$ as in Equation (3.2) on page 32. Secondly, the simplifying assumption is made that the occurrences of the additional interruptions are seldom with respect to T (in the sense that more than one occurrence within a time interval of length T is unlikely) and can be modeled by a *Poisson distribution* (compare e.g. [Hoe66, 5.2.4]). The choice of this distribution is neither based on empirical results, nor on theoretical considerations about the exact nature of these events, but on the assumed rarity of the additional interruptions and also the distribution's simplicity.

So let φ be the average number of Poisson-distributed other interruptions per time unit (e.g., $\varphi = 1/\,$s). The probability of no such interruption during a measuring phase is then $e^{-t_m\varphi}$, so that the total detection probability becomes

$$\tilde{\lambda}(t_m) = \frac{t_m}{t_m + t_e} \cdot e^{-t_m\varphi}.$$

Theorem 3.4. *Given $c, \varphi > 0$, the function $g(t) = \frac{t}{t+c}e^{-t\varphi}$ assumes its only maximum on $\mathbb{R}^+$ at*

$$t^* = -c/2 + \sqrt{(c/2)^2 + c/\varphi}$$

and is strictly monotonic increasing on $[0, t^*)$.

The maximum given by Theorem 3.4 can of course only be assumed if there exists a valid ε such that $t_m = D - c_0 - \varepsilon$ as before. The monotonicity of $\tilde{\lambda}$ therefore suggests to choose

$$t_m = \min\left(D - c_0 - t_g, -t_e/2 + \sqrt{(t_e/2)^2 + t_e/\varphi}\right),$$

i.e. either result of the theorem or, if this value is too large, the maximal value that still has a corresponding valid ε.

Table 3.7 shows the suggested measuring duration t_m and the resulting detection probability $\tilde{\lambda}$ for some chosen values of φ, from no additional interruptions at all to an average of 50 per second. The larger the value of φ, the more inaccurate $\tilde{\lambda}$ is expected to be, since then also other tasks are likely to be active and disturb the transmission. The suggested t_m

[26] This situation only applies if not all processes are subject to the fixed timeslice lengths.

[27] In the testing environment, for example interrupts resulting from mouse moves turned out to have durations in the interval $[10.4\,\mu\text{s}, 11.7\,\mu\text{s}]$.

φ	t_m	$\tilde{\lambda}(t_m)$	σ	γ
0 /s	1153.00 μs	0.002%	1945.82 μs	1211.15 μs
1 /s	164.30 μs	0.033%	1451.47 μs	222.45 μs
10 /s	51.95 μs	0.104%	1395.29 μs	110.10 μs
50 /s	23.22 μs	0.232%	1380.93 μs	81.37 μs

Table 3.7. Suggested measuring durations t_m and their influence on the loss rate, the suggested skew and the suggested size of the gap.

for $\varphi = 1$ /s is close to the t_m of Setting 3 in Table 3.2, page 34, which led to the low loss rates. One might thus argue that in this environment, disturbances leading to an interrupt loss in the receiver occur at most once per second.

For the sake of completeness it has to be remarked that the values computed for $\tilde{\lambda}$ are lower than the experimentally determined values of λ since the former does not capture interrupts that are either not generated by the sender at all, or occur while the receiver is executing.

Interrupt Misaccountings

From the four new parameters introduced for the adjustment of the bit interval skew and the gaps between bit intervals, M is considered first. A practical evaluation of M is expected to be error-prone, as additional time measurements and their recordings in user- and kernel-space almost necessarily tamper with the results. Therefore the pessimistic assumption is made that M is distributed uniformly on $[0, t_m)$, so that the uncertainty about the outcomes is maximal (compare [CT06, Example 12.2.4]). However, for the following computations it is only of interest that the support interval of M is the full $[0, t_m)$.

Now that M has been fixed, it is possible to advance to the distribution of the random variable Z introduced in the generic framework. But instead of computing or only approximating the convolution, the distribution is only reduced to its support interval $[Z^-, Z^+]$, which is assumed to be contiguous (otherwise also a contiguous extension of the support is sufficient). Formally it is required that $p(Z^- \leq Z \leq Z^+) = 1$ and $p(z^- \leq Z \leq z^+) < 1$ for all intervals $[z^-, z^+] \subsetneq [Z^-, Z^+]$. For convenience, the width of the interval is referred to as $\Delta Z := Z^+ - Z^-$.

Regarding distribution B, two options will be exercised. In the first, the concrete shape of the distribution is ignored and it is only assumed that distances to bit interval boundaries may become arbitrarily small (i.e. $\int_0^t B(\delta)\, d\delta > 0$ and $\int_{t_b - t}^{t_b} B(\delta)\, \delta > 0$ for all $t > 0$). Then it follows from Equation (3.4) on page 46 that, in order to prevent from misaccountings, $p(\sigma - \gamma/2 \leq Z \leq \sigma + \gamma/2) = 1$ has to hold. This in turn is equivalent to the conditions $\sigma - \gamma/2 \leq Z^-$ and $Z^+ \leq \sigma + \gamma/2$. An immediate consequence of these inequalities is $\gamma \geq Z^+ - Z^- = \Delta Z$, i.e. the gap must be at least as large as the variation of the delay until the detection of the interrupt, as intuition suggests. The choice of the smallest gap $\gamma = \Delta Z$ finally leads to a skew amount of $\sigma = (Z^+ + Z^-)/2$. For some possible choices of t_m, the suggestions for σ and γ can be obtained from Table 3.7.

A gap implemented into the receiver has the disadvantage that some interrupts may have to be ignored. This is similar to a higher loss rate, even if it cannot generally affect all interrupts of a bit interval at the same time (unless $N = 1$). Thus, one usually wants to avoid gaps in the receiver, for example by ensuring that the sender's clock queries keep a

	Kernel	Userspace
Measurings:	5	5
Samples/measuring:	≈ 18500	25000
Duration/measuring:	$\approx 1\,\mathrm{h}$	$\approx 10\,\mathrm{min}$
Average timer interval:	9999669.43218 ns	9999669.43188 ns
	± 0.0035 ns	± 0.00024 ns
Standard deviation:	14 ns to 27 ns	293.4 ns $\pm$ 0.8 ns

Table 3.8. Results of timer interval length measurements.

certain minimum distance from the bit interval boundaries. The following therefore assumes $\gamma = 0$ and finds out, which distance distribution B is necessary to make this possible.

With $\gamma = 0$, from Equation (3.4) on page 46 one obtains that $p(-(t_b - \delta) + \sigma \leq Z \leq \delta + \sigma) = 1$ has to hold for all $\delta \in [0, t_b)$ satisfying $B(\delta) > 0$. Similar to the above calculations, this is equivalent to $t_b - \delta + \sigma \leq Z^-$ and $Z^+ \leq \delta + \sigma$, for the same set of δ. Let $[b^+, t_b - b^-]$ be an interval fulfilling $\int_{b^+}^{t_b - b^-} B(\delta)\, d\delta = 1$, i.e. the support of B is a subset of $[b^+, t_b - b^-]$. Then a sufficient condition for no misaccountings is the conjunction of the above inequalities for all $\delta \in [b^+, t_b - b^-]$. Few transformations of this conjunction leads to $Z^+ - b^+ \leq \sigma \leq Z^- + b^-$, which only has a solution for σ if $\Delta Z \leq b^+ + b^-$. Informally this simply states that the minimum distance to the "left" (b^-) an the "right" (b^+) of the sender's clock query must together be larger than the gap, that the receiver would otherwise have to implement (compare γ above). For the following it is assumed that $b^+ + b^- = \Delta Z$.

Until now it looks like the occurrences of bit interval boundaries have to be limited to an interval of duration $t_b - b^- - b^+ = t_b - \Delta Z$. Earlier explanations however showed that other tasks waking up may introduce timeslice shifts of arbitrary integral multiples of the timer interval length T. Therefore, the bit interval boundaries only have a tolerance of $T - \Delta Z$, which corresponds to the maximum accumulated drift between timeslices (in the form of $N \cdot (t_s + t_r)$) and bit intervals (t_b) during a transmission. Assuming a transmission duration of one hour and $T = 10\,\mathrm{ms}$, then the deviation of t_b from $N \cdot (t_s + t_r)$ must be below 3 ppm (parts per million) to stay within the tolerance![28]

Determining $t_s + t_r$ from within the exploit processes is not trivial, especially if the value has to be highly accurate. Surprisingly it turned out, that even the knowledge of the putative values of t_s and t_r as they are setup in the kernel does not necessarily suffice. In the concrete example, both t_s and t_r were set to 100 ms, which the kernel treats as 10 timer interrupts at 100 Hz. However the actual timer interval length was not 10 ms, as the 100 Hz would suggest, but about 33 ppm shorter (compare Table 3.8).[29] Consequently, the timeslice lengths are shorter than expected as well.

Besides the actual results, Table 3.8 also reveals that measuring the timer interval length in userspace is possible. By using linear regression on the collected times of timer interrupts, even a much higher accuracy was possible than the used time function, `gettimeofday()`, supports on its own (it returns seconds and microseconds). The collection of timer interrupts was done similarly to the detection of sender-initiated interrupts in the receiver of the exploit.

[28] This comes from $(10\,\mathrm{ms} - \Delta Z)/1\text{ hour} \leq 2.78 \cdot 10^{-6}$ for $\Delta Z \geq 0$.

[29] The cause for this discrepancy are different clock sources, like in this case the local APIC being used for timer generation but the TSC for the current time.

If sender and receiver of the exploit are able to use the same bit interval length, for instance because the receiver determined it and was able to communicate it to the sender, then a transmission could last for more than 11 years (!) at a stretch until the potential inaccuracy of the measured bit interval length could accumulate to more than 8788.5 μs (corresponding to $T - \gamma$ for the userspace-measured mean T and the largest γ of Table 3.7). On the other hand, if both exploit processes have to determine the bit interval length on their own, then their results may differ and, hence, result in a drift between their notions of the bit interval positions. This drift in turn requires for a gap which prevents from misaccountings and has to be longer with increasing transmission durations. In the concrete case, if the transmission takes no longer than 39 days, then the potentially accumulated drift remains below the smallest gap of Table 3.7, $\gamma = 81.37\,\mu$s.

Of course the decision whether the exploit processes should precisely measure the timer interval length ought to consider also that this measurement requires for time that – depending on the implementation[30] – cannot be used for the transmission itself. The transmission time that is lost due to the timer detection therefore has to be traded off against the lost transmission time (and higher losses) resulting from gaps.

For the experiments of Table 3.2 on page 34, no gap was used ($\gamma = 0$) and the skew has been set to $\sigma = 1.5$ ms. The timeslice lengths on whose basis the bit interval length has been set, were $t_s = t_r = 99.99669431$ ms. This precision for the timeslice lengths was the reason why the skew has not been changed along with the measuring duration, contrary to the suggestion of Table 3.7 on page 51.

3.3.3. Applicability & Limitations

Applicability. Concluding, the application of the generic framework to the given concrete instance demonstrates the usability and the flexibility of the framework. At the same time it gives an advice regarding what steps have to be taken in order to get to the desired results, a suggested exploit configuration. It is expected that most of these steps can be copied for other target systems if the exploit still uses NIC interrupts or other interrupts with a narrow delay distribution. Only the characteristic values for the NIC and timer interrupts would have to be changed.

Due to its generality and its abstraction from the actual implementation, the framework should also be applicable to other implementations that exploit IRCCs, though potentially with a different set of exploit parameters.

Although the framework was originally constructed only for dealing with IRCCs, it may well be possible to confer it to similar kinds of covert channels, like e.g. quantum-time channels. For their implementation, configuration and analysis it might, as for IRCCs, be of interest how often a receiver process does not detect the execution of the sender, or how often the receiver cannot definitely assign a detected execution of the sender to a certain bit interval.

Limitations. The performed experiments show that the framework did not yield an optimal set of parameter values for the exploit in the concrete instance. This does not necessarily have to be a problem of the framework but could as well be the result of an inaccurate

[30] It may be possible to detect and compute the timer interval during the transmission, both in the sender and in the receiver.

instantiation that, e.g., did not model the duration of clock queries or ignored the duration of the "other interrupts."

The framework also cannot explain why different experiments with the same exploit parameters and an unchanged testing environment may lead to different loss rates (compare especially Setting 6 of Table 3.2 on page 34).

3.4. False Positives

So far, the channel model and, particularly, its transition matrix P assumes that sender-initiated interrupts belonging to a bit are either detected by the receiver or may also get lost for reasons that have been explained earlier. However it does not comprise the possibility that the receiver could in principle also regard arbitrary other interruptions as coming from the sender. Examples for such interruptions range from interrupts coming from the same hardware device but triggered by another process than the sender, over resembling interrupts coming from different hardware devices, to interruptions even by other tasks. Ignoring the possibility of such *false positives* in the receiver has basically two consequences: (a) The computed theoretical capacities and bandwidths may be higher than what is actually possible on a real system. For an attacker this could be disadvantageous since transmissions turn out to be slower than expected. (b) The chosen channel encoding might not work at all. For example, the threshold encoding with threshold $\tau = 1$ would, independently of the sender's input, return a sequence of **1**'s if there should be at least one false positive within every bit interval.

Obviously, this deficiency has to be remedied. The first and possibly most intuitive approach for doing so would be to refine the model by adding false positives to the transition matrix. For input **0**, the outcome would then be distributed based on the frequency of false positives only, while the output corresponding to input **1** would be the result of a sum (convolution) of the false positives and the correctly detected interrupts coming from the sender. Given the resulting transition matrix, the computation of the channel's capacity and bandwidth is possible and a suitable channel encoding can be chosen. The problem with this approach is that the distribution of false positives may be complex and varying over time, and may not be known even approximately to a remote attacker. Examples that cause such complex distributions could be Ethernet traffic (compare e.g. [LTWW94]) or direct human interaction with devices like a keyboard or a mouse. Of course this argument also applies to the binomial distribution and its parameter λ of the previous model, but is less crucial there since only the value $p(Y{=}0|X{=}\mathbf{1})$ – the probability of *false negatives* – of this distribution is relevant for the analysis.

Instead of refining the model and choosing example distributions for false positives, another approach is taken: Ensuring that false positives either do not occur at all or at least seldomly enough to be negligible. Therefore, the receiver must, potentially in cooperation with the sender, be able to distinguish interrupts caused by the sender from other, initially resembling interruptions and ignore the latter. The aim of this section is to introduce techniques that can be implemented into the exploit and then allow the receiver to discriminate interrupts also based on other attributes than only their duration.

In the following, at first the potential noise is subject to a closer look. Afterwards two techniques are described that may be used for filtering noise, possibly also in combination. In both cases it is also possible to use the underlying ideas for increasing the upper bounds of

the implemented channel's bandwidth. A short discussion of these alternatives is therefore presented at the end of each technique.

3.4.1. Classification of Noise

In the given implementation, the receiver tries to detect interrupts by measuring the elongation of the runtime of a certain piece of code. Thus, everything happening in a computer that is able to elongate the execution time of a simple loop with a floating-point computation, may in principle be confused with sender-initiated interrupts. Besides hardware interrupts, this very broad definition also permits rather exotic influences as, for instance, hardware that wants to access the memory bus at the same time as the receiver or disadvantageous branch predictions in the CPU. It is expected, though not experimentally verified, that these latter effects cause only elongations that are by far shorter than those of hardware interrupts. The following will therefore ignore the potential exotic influences and concentrate only on hardware interrupts.

From the receiver's perspective, a hardware interrupt lasts from the time of the actual interrupt signal (by the hardware to the CPU), which stops the execution of the receiver, until the time that the receiver task is resumed. In the meantime at first the interrupt is handled, for example by driver code. However this kernel code might decide that, after the actual interrupt handler has finished, not the originally interrupted task should be resumed but instead another task should be scheduled (compare e.g. [BC05, p. 273] for timer interrupts or [CRKH05, p. 270] for device drivers). The consequence of rescheduling before returning to the interrupted task is a longer and potentially more varying interruption duration.

Another important aspect of hardware interrupts is *how* they are caused, which can be grouped into three categories: (a) Externally caused interrupts, like those triggered by a keyboard or a mouse, or the NIC for an incoming packet; (b) internally caused interrupts, for example indicating the completion of a HD read or write, or the sending of a network packet; and (c) automatically caused interrupts like timers. Timers do not need any further explanation. They occur at a high but constant frequency and may effect a rescheduling. Externally caused interrupts in contrast can occur at any time and with an almost arbitrary frequency, while internally caused ones occur within a certain period of time after their generation.

Regarding internally caused interrupts, there are basically two options for their origin: Blocking or non-blocking system calls. The former effectuate an immediate yield of the CPU such that the interrupt is likely to occur while another task is executing (and then might cause a rescheduling back to the originator of the interrupt). Furthermore, due to the immediate yield, the time-frame during which such an interrupt would occur in the IRCC receiver, if it was scheduled after the yield, would be determined by the delay distribution of the respective interrupt. For interrupts generated with a non-blocking system call, this property is not given unless the respective task immediately afterwards voluntarily yields the CPU (like the sender of the exploit does). It may happen that in the non-blocking case the resulting interrupt always or almost always interrupts the generating task itself (if it does not yield or block until then) or imply a relatively long time-frame during which the succeeding task (the receiver) is interrupted.

3.4.2. Incorporating Interruption Times

In its initial form, the implemented exploit considers an interruption to be relevant only based on its duration. If it falls into a certain range, then the interruption is considered to be caused by the sender, otherwise it is ignored. Furthermore, the receiver also measured for interruptions during a full timeslice, here 100 ms, even though it was known that the sender-initiated interrupts in the testing environment should occur only during a relatively small fraction of every timeslice (compare e.g. the variation of the delay in Table 3.5, page 47).

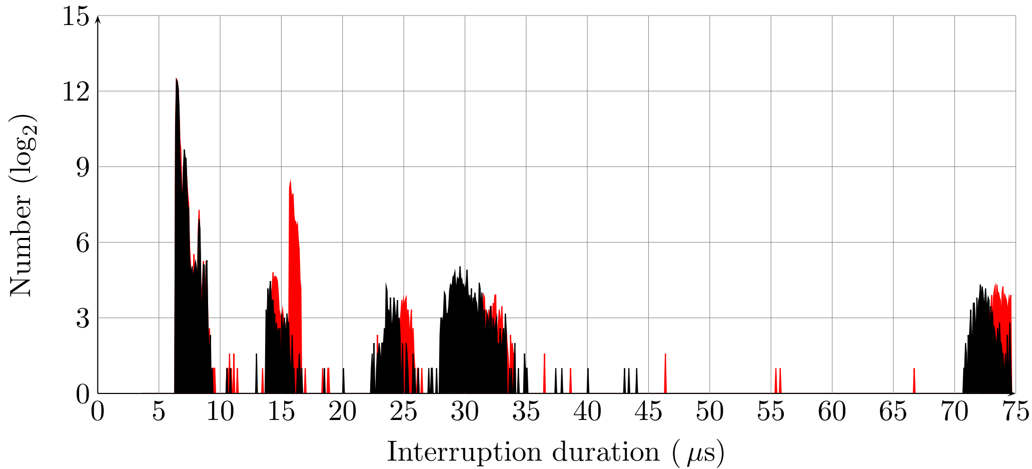
To make use of more information about the times of interrupt occurrences, the receiver thus needs a notion of timeslices. Once it knows when the respective current timeslice has begun, it can easily determine at which offset in the timeslice an interrupt occurs. For the implementation of the sender, the system function sched_yield () has been successfully used to add the notion of timeslice, so it is apparent to proceed similarly also for the receiver. However, in case of the receiver two problems have to be solved first:

(a) The receiver somehow has to find out the right time for yielding the CPU, if possible without permanently querying the CPU and thereby increasing the evaluation duration t_e. For setting up the bit interval length suitably (avoiding drifts that finally require for gaps), it would additionally be helping if the CPU is yielded always at the same offset in the timeslice so that t_r remains constant as before (this latter problem vanishes if the operating system enforces fixed timeslice lengths not only for the sender but also for the receiver).

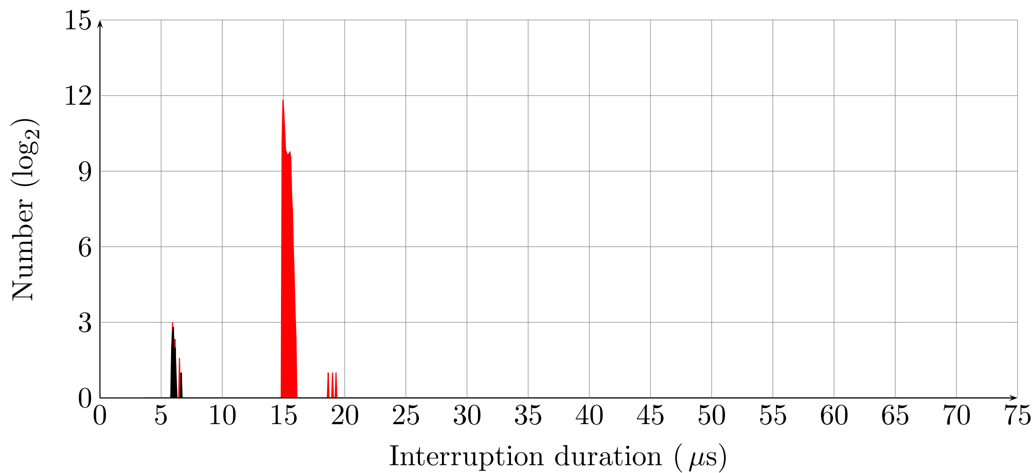
(b) Through yielding the CPU before the end of its timeslice the receiver should not negatively influence the sender. However if it yields the CPU, say, in the middle of two timer interrupts, then an immediately following sender timeslice will not have the full expected length t_s but rather approximately half a timer interval less.

For solving the second problem it was decided to yield the CPU shortly after a timer interrupt, so that if the sender is scheduled afterwards then its timeslice length is reduced by only a minimal amount of time. The number of the timer interrupt, after which the CPU is yielded, should of course be chosen such that afterwards no further sender-initiated interrupts are expected. Regarding the implementation to solve the first of the above problems, it was chosen to use (software) timers via the setitimer () system function. Since the operating system delivers the SIGALRM signal only after timer interrupts, the receiver does not even have to know an exact time for when the timer should expire. Alternatively, the receiver could also try to detect the timer interrupts itself, just like it also detects the sender-initiated interrupts, but this is expected to complicate the implementation much more than the setitimer () solution did.

In the testing environment the receiver can already yield the CPU after the first timer interrupt of its timeslice, resulting in a shortened receiver "timeslice" length $t_r = 10\,\text{ms}$ (a single timer interval). This does not only increase the channel bandwidth, but also reduces the reception time during which the receiver would otherwise only have been able to collect false positives. Figure 3.6 illustrates the huge impact of this modified implementation in the rather noisy extended testing environment. For the figure, all interruptions with a duration of up to $75\,\mu\text{s}$ were collected with a receiver process and compiled into a histogram with a bin width of approximately 71 ns. The peaks at about $15\,\mu\text{s}$ in both red histograms



(a) Receiving for 655 s with full receiver timeslices ($t_r = 100$ ms).



(b) Receiving for 3600 s with shortened receiver timeslices ($t_r = 10$ ms).

Figure 3.6. Histograms of interruption durations in the extended testing environment, with (red) and without (black) sender-generated interrupts.

show the interruptions of sender-initiated NIC interrupts (compare the duration in Table 3.5, page 47).[31]

It is assumed that most of the noise shown in the two histograms of Figure 3.6a is the result of the kernel scheduling other tasks after a timer interrupt. Externally caused interrupt can be excluded, since keyboard and mouse were not touched throughout all experiments and no other hosts could have triggered NIC interrupts as none were connected to the testing environment's network. The only suspect for internally caused interrupts would probably be the hard disk, but since no programs were deliberately started and no data was deliberately read from or written to a disk, their number should be fairly small compared to the amount of noise in the histogram.

Although these results are already impressive, the receiver still left a time-frame of roughly 10 ms for the sender-initiated interrupts and, thus, also for noise. The variation of the NIC interrupt delay, however, is much shorter (compare again Table 3.5 on page 47) and should allow for an even stricter filtering.

To verify this hypothesis, the receiver was enhanced to determine the current time at the beginning of every timeslice and again shortly *after* a sender-initiated interrupt has been

[31] The "NIC-peak" in Figure 3.6b is higher than in 3.6a since for both figures not the number of generated interrupts but the reception time was kept constant to keep the amount of noise comparable.

detected. The differences between both times have been collected during a transmission of the sequence $(\mathbf{10})^{8000}$ and compiled into Table 3.9. For the measuring, a short $t_m \approx 0.91\,\mu$s was chosen to increase the accuracy of the detection times.[32] In order to accept all sender-initiated interrupts it would thus have sufficed to leave a time-frame of about $33\,\mu$s, which is even shorter than the delay variation computed in Table 3.5 ($54\,\mu$s). The low standard deviation of the values from the mean indicates that this time-frame could even further be reduced with risking only few false negatives. In addition, the time-frame may also be moved in time by changing the value of ε in the sender.

The approach of allowing only short time-frames for the sender-generated interrupts has the potential of filtering out timer interrupts and internally caused interrupts completely, and of reducing externally caused interrupts. The latter kind can in principle occur at any time, so the probability of such interrupts occurring in a short time-frame is reduced but may still be nonzero. Timer interrupts and internally caused interrupts can be excluded as a cause for false positives if the time-frame for sender-initiated interrupts can be moved to an offset in the receiver's timeslice where it is distinct from the foreign interrupts' time-frames. However, the longer the time-frames of sender-initiated and/or foreign interrupts are, the larger is also the remaining probability for false positives.

Samples:	7027
Mean:	$1180.1\,\mu$s
Std.dev.:	$0.9167\,\mu$s
Min:	$1158.4\,\mu$s
Max:	$1191.5\,\mu$s

Table 3.9

The reason why this modified receiver implementation has not been used for the whole thesis is that its interrupt loss rate was significantly higher (10% and more) than it was with the unmodified version. The cause for this deficiency however could not be identified.

Alternatives. Instead of using the time of an interruption as a property for filtering noise, it could also be used for an extension of the input alphabet. Regarding the implementation of the sender, the constant temporal pre-position ε could be replaced by a variable whose value depends on the respective bit of data to be sent. Analogously the receiver could assign a certain value to an interruption depending on when within the receiver's timeslice it has occurred. Based on a variation of $33\,\mu$s for the detection time of an interrupt, and a possible interval for ε of $[t_g, D + d] \approx [165\,\mu\text{s}, 1341\,\mu\text{s}]$ (see Section 3.3.2), this would allow for $35 = \lfloor(1341 - 165)/33\rfloor$ distinct values for ε. Including the possibility of not generating an interrupt, the input alphabet could thereby extended to 36 elements that might be distinguished by a receiver. Without any disturbances the capacity would then increase from 1 to $\log(36) \approx 5.2$ bits per "bit" interval.

3.4.3. Generating Multiple Interrupts

Depending on the interrupt source chosen for the channel it may be possible to generate multiple interrupts with a single call to a system function. In case of the NIC it was discovered that sending a payload that, together with all packet/frame headers, would exceed the maximum transmission unit (MTU) leads to multiple packets being sent consecutively and one interrupt being raised for *each* sent packet. The typical MTU value for Ethernet is 1500 B. Table 3.10 shows the characteristics of the four interrupts resulting from sending 5800 B. The standard deviations of the interrupts' delays and durations are small compared to the respective means, which qualifies both attributes for being used as filtering criteria.

[32]Due to the modified receiver, the measuring phases now have rather fixed offsets within the timeslice, so

		Samples	Mean	Std.dev.	Min	Max
Generation		20000	338.65 μs	7.718 μs	336.68 μs	1088.30 μs
#1	Delay	6384	1394.99 μs	4.604 μs	1393.68 μs	1752.59 μs
	Duration	6384	182.03 μs	0.246 μs	181.50 μs	185.45 μs
#2	Delay	5472	2638.96 μs	4.987 μs	2637.50 μs	2997.24 μs
	Duration	5472	170.52 μs	0.282 μs	170.13 μs	171.48 μs
#3	Delay	5472	3882.77 μs	4.998 μs	3881.31 μs	4241.82 μs
	Duration	5472	24.29 μs	0.228 μs	23.90 μs	24.99 μs
#4	Delay	5472	5036.99 μs	4.999 μs	5034.71 μs	5396.05 μs
	Duration	5472	14.84 μs	0.227 μs	14.57 μs	15.46 μs

Table 3.10. Generating 4 interrupts by sending 5800 B of payload (values for the RTL8029 NIC controller).

In principle, multiple interrupts may also be generated by explicitly calling the generating system function multiple times. This approach would allow for a more fine-grained control, for example in terms of the possibility of choosing different interrupt sources or influencing the temporal distance between the occurring interrupts. In the latter case it however has to be considered that the distance between the interrupt-generating calls may have no influence on the distance between the actual interrupts: Requests to the respective device might be buffered by the kernel until the completion of the first request, indicated by the first interrupt. A potential disadvantage of the approach with regard to filtering noise is a higher variation in at least the generation duration, but potentially also the interrupt delay.

The receiver could use multiple interrupts to reduce the probability of false positives in various ways:

- If the individual interrupts differ in their duration, as it is the case in Table 3.10, then the receiver may choose to consider only those of the interrupts whose duration ranges are not subject to noise.
- If the receiver has a notion of timeslices, as suggested in the previous section, then it may choose to count only those of the interrupts whose range of temporal offsets in the receiver's timeslice does not comprise noise.
- The receiver may try to detect all of the generated interrupts and increase the output counter for the respective bit only if at least some minimum number of them is found in a timeslice.
- In addition to the previous bullet, the receiver may require that the temporal distances between the putative sender-generated interrupts are correct.

Generating multiple interrupts instead of just a single one adds redundant information, which the receiver in principle can use to filter any kinds of foreign interrupts (timers, internally and externally caused, with and without subsequent rescheduling). Certainly, false positives are still possible, but the probability of multiple interrupts occurring during a single receiver timeslice, all with the right offset and duration, should be small enough to be negligible.

that a larger t_m could result in a detection always within the same evaluation phase.

Though multiple interrupts are, in case of the NIC, simple to generate, they have not been used for producing the main results of this thesis, as the resulting channel encoding would have differed from the one of the original exploit implementation (see the beginning of Chapter 2).

Alternatives. Similar to the encodings presented in [MS07] and [MS08], multiple interrupts can be used to extend the input alphabet. If k is the maximum number of interrupts that can be induced with a single generating function call, then one might define $I := \{0, \ldots, k\}$ and then for every input $x \in I$ generate exactly x interrupts. In the absence of any noise or losses and under the assumption that all k different interrupts can be distinguished by the receiver (based on their durations and/or times), this would increase the capacity from 1 to $\log(k + 1)$ (corresponding to the entropy of the input).

Even if the input alphabet is not enlarged, multiple interrupts could also be used for reducing the loss rate λ since, for instance, four interrupts are less likely to be all undetected than just a single one. To transmit a **1**, the sender might then always generate k interrupts.

4. Conclusion

4.1. Novel Countermeasures

In [MS07] the authors already propose and evaluate a set of countermeasures against IRCCs, which were already implemented in some operating systems but for other reasons than mitigating IRCCs. During the work on this thesis, especially on the implementation of the exploit and the scheduler patch, two additional countermeasures were found. Both can in all likelihood be implemented into a scheduler like Linux's O(1) or FreeBSD's ULE scheduler without much effort and at the same time do not have a huge impact on the system's overall performance.

Inter-timeslice gaps. Interrupt-related covert channels, as they are covered by this thesis, are only possible if one process – the sender – is able to generate an interrupt that occurs while another process – the receiver – is executing. Since it usually does not take a long time from the generation of an interrupt until its occurrence (little more than a 1 ms in case of the NIC of the testing environment), sender and receiver process have to execute in relatively quick succession. The operating system could therefore mitigate IRCCs by ensuring a minimum temporal distance between the execution of two tasks, of which the first must not transmit information to the second. If a distance is guaranteed that is longer than the maximum delay of all interrupts available to a sender process on the respective system, then IRCCs may even be suppressed completely. During the gaps, the CPU does not necessarily have to idle. Instead, other runnable tasks could be scheduled between a switch from a sender to a receiver, as long as they do not belong to the class of senders themselves.

With only little modification, the scheduler patch developed for this thesis is already able to establish this countermeasure: Instead of storing the remaining time of the sender's timeslice into `fq_wait_ticks` in `fixed_quantum_update_leaving()` (Listing 2.7, page 22), an arbitrary amount may simply be added to ensure a gap.

As mentioned, the length of the gap and, thereby, the impact of this countermeasure on the system performance depends on the maximum delay of interrupts that a sender process can generate. One might imagine situations where this maximum delay can become too long for the gaps to be feasible any more. An example could be a sender process that deliberately sends requests to a server that responds very late or with a large number of consecutively arriving packets. The corresponding receiver would in this scenario then try to detect the NIC's *reception interrupts*. Even under these circumstances, the proposed gaps might still be useful for absorbing interrupts with short delays, but could then only mitigate instead of eliminate IRCCs.

Randomized timeslice lengths. In Section 2.1.2, the O(1) scheduler is modified to eliminate quantum-time channels from sender tasks to receiver tasks by ensuring a notion of

fixed timeslice lengths for senders. Consequently, the sender of the exploit is implemented to generate its interrupts shortly before the end of its timeslices, such that the interrupts can occur while the next task – the receiver – is executing. Fixed timeslice lengths, however, are not a necessary condition for eliminating quantum-time channels. It is only required that the sender cannot, for instance by yielding the CPU, influence when the receiver is scheduled next time. To achieve this, the scheduler could also implement randomized timeslice lengths, distributed on some reasonable interval.

Regarding the elimination of quantum-time channels there should be no difference between fixed and randomized timeslice lengths. However, the IRCC sender process implemented for this thesis would only be able to make a small fraction of its generated interrupts be detectable by the receiver. All others would occur while the sender itself is still executing. An adapted sender that generates an interrupt shortly before all, most or at least multiple possible ends of its timeslice – e.g. shortly before each timer interrupt –, may be able to circumvent the countermeasure, but this approach (a) may be possible only to a limited amount, e.g. in case of interrupt rate limiting, (b) may not be possible at all, e.g. if the system calls generating the interrupt always block the sender, and finally (c) would make the sender behave even more suspicious.

In order to implement this countermeasure, the scheduler would only have to refill a (sender) task's timeslice with a (pseudo) random number of ticks instead of a constant one. Few changes to Listing 2.4 on page 19 should suffice to achieve this.

4.2. Surprises

Fragility. During the work in this thesis, a lot of transmissions with the exploit were performed in order to improve the implementation, get rid of bugs, and discover the effect of parameter or environment changes. Often it was discovered that even innocent looking modifications suddenly resulted in significantly higher loss rates or, sometimes, disabled the channel completely. Though annoying, these incidences deepened the understanding of IRCCs and are therefore not excluded from the thesis.

- At the very beginning of the work, the Linux kernel configuration had to be set up. Initially, the USB stack had been enabled, but since no USB devices were attached at that time and the kernel was decided to be reduced to the minimum, USB was finally removed. The consequence was that the NIC interrupts suddenly led to shorter interruptions of the exploit's receiver – even though, as said, no USB devices were attached before, and also the NIC did not share its IRQ number with the USB controller.

- In the beginning of the work on the exploit, the measuring duration (t_m) – more precisely the value of `iterations` – was hardcoded into the receiver. When this value was then changed it often happened that the putative durations of sender-initiated interrupts also altered. Longer measuring durations thus seemed to cause longer interrupt durations, though of course the interrupts itself did not change.

  This effect vanished once the measuring duration was no longer hardcoded but instead given as a command line parameter. It is thus presumed that the discovered behavior was the result of compiler optimizations, but the exact culprit is not known. However, one might imagine that even an already finished interrupt handler has an influence

on the subsequently executed instructions, for instance because the context switches flushed the CPU's information (e.g., branch prediction) about how to arrange the execution of these instructions. Longer measuring durations in this context would result in more, potentially slower executed instructions after the interruption and, thereby, longer putative interruption durations.

The consequence of these observations is that an actual attacker may have to know more about the target system than intuitively necessary and must be careful that small, seemingly insignificant changes of the exploit code or the compilation conditions do not disable the transmission completely.

Availability. The exploit presented in this thesis was implemented with NIC interrupts in mind, even though it does not explicitly exclude the possibility of being used together with other interrupt sources. The results presented in Table 3.2 on page 34 show that IRCCs on the basis of NIC interrupts are possible. Furthermore, Table 3.5 on page 47 indicates that NIC interrupts may even be a very good choice due to their very small variances, especially regarding their delay and duration. Nevertheless this latter table also shows a tendency: Interrupt delays (especially after subtracting the generation times) are approximately inversely proportional to the network bandwidth. While the delay with a 100 Mbit/s NIC should still be sufficient for establishing an IRCC, a Gigabit Ethernet NIC could already be sending out packets too fast for letting the completion interrupt occur during a receiver timeslice. To retain the possibility of a NIC-based IRCC, the sender might then try to resort to sending multiple packets, as described in Section 3.4.3, or "jumbo frames" (larger Ethernet frames), if they are available.

4.3. Related Work

The focus of this thesis is on the implementation and analysis of an exploit of interrupt-related covert channels, which have first been introduced in [MS07] and are a specific class of covert channels. A different approach of exploiting hardware interrupts for malicious purposes is presented in [Tro98], where the keyboard input is eavesdropped by detecting the resulting interrupts. This idea however differs from the work in this thesis, in that is does not constitute a communication channel with a deliberate sender.

Covert channels in general have been introduced by Lampson in [Lam73]. The research on this kind of channels can be categorized into (a) *identification*, (b) *modeling* and *analysis*, in particular of the bandwidth, (c) *mitigation* or *elimination*, and (d) *implementation*. An overview of the former three aspects can be found in [Gli93] and [McH95].

Identification. In [Gli93], three approaches for identifying covert channels are observed: (a) Syntactic and semantic information-flow analysis (compare, e.g., [Den76]), which identifies covert information flow between "storage objects"; (b) the shared resource matrix method (see [Kem83, Kem02]), which identifies the possibility of a covert channel with a matrix specifying which primitive (operation) may reference and/or modify which resource attribute; and (c) noninterference analysis (see, e.g., [GM82, Rus92]), which is based on a finite-state representation of the system under investigation (more precisely, its trusted

computing base, TCB) and a formal analysis of the set of possible executions ("runs") of the system.

Besides IRCCs as introduced in [MS07], also the quantum-time and interquantum-time channels of [Hus78] exploit the sharing of the CPU between processes. Other covert channels have been found, which are based on, e.g., shared disks in [KW91], a shared system bus in [Hu91], or a shared network in [Gir87].

Modeling and analysis. IRCCs are modeled and analyzed in an information-theoretic framework in [MS07, MS08]. The information-theoretic approach is formally based on [Sha48] and has also been applied for the analysis of covert channels, e.g., in [Mil89, Gra93, MM94].

As an example for a different approach of modeling and analyzing covert channels, the Markov model of [TG88] shall be mentioned.

Mitigation. A formal analysis of several countermeasures against IRCCs, including, e.g., a reduced clock resolution, polling and interrupt-rate limiting, is presented in [MS07]. The approaches that are not specific to interrupts can also be found for other covert channels, like, e.g., the fuzzy time for the bus-contention channel in [Hu91]. For covert channels exploiting other shared resources, however, also different specific countermeasures are suggested (compare, e.g., [KW91]).

Implementation. Explicit implementations of exploits for covert channels can hardly be found. Often, only more or less concrete ideas of actual or potential implementations are described; sometimes only a hint about the existence of an exploit is given, as, for example, in [Hu91].

Several implementations exist for different *network-based covert channels*. In [CBS04], the authors describe their implementation of a "covert network timing channel" that transmits data via the reception or, respectively, absence of network packets. Related to this channel are the "JitterBugs" of [SMB06], which transmit information with varying packet times and have been implemented in hardware. A channel based on packet ordering/sorting is described in detail in [Ahs00, AK02], but without referencing the existence of an exploit implementation.

Approaches that can also found under the term of covert channels, transmit data (covertly) over overt channels, for instance in unused protocol header fields [Rut04, ML05].

4.4. Summary

This thesis approached interrupt-related covert channels from the perspective of an attacker. Therefore it showed how an attacker could implement an exploit that is able to perform its task even on a system enhanced with countermeasures against a similar kind of covert channel. This exploit was afterwards modeled and analyzed theoretically, to obtain upper bounds on the exploit's bandwidth. Practical experiments performed in a given environment complemented the results with lower bounds. The free parameters of the exploit were subject to a framework, that allowed for obtaining a configuration tailored to a given target system. Potential problems resulting from noise were discussed and techniques for avoiding or reducing them were proposed.

To sum up, this thesis demonstrates the threat that IRCCs pose to secrets stored on a computer system.

Implementation. The scheduler modification of Section 2.1 showed a possible solution for prohibiting quantum-time channels in the Linux operating system. With the usage of a simple information flow policy, it degrades the system's performance only when necessary.

More important than the modified scheduler is the exploit implementation that has been presented in Chapter 2. It consists of a sender process that, depending on the bit value to transmit, generates one interrupt or none, and a receiver process that steadily measures for detecting the occurrence of an interrupt. Both processes are intended to execute concurrently and alternatingly to establish the channel. The only functionality that the operating system has to offer to the processes is a function to query an accurate clock and a function to yield the CPU.

Analysis. In Chapter 3, the implemented exploit was at first modeled as a discrete memoryless channel. It was then computed that the upper bound of the channel bandwidth is 5 bit/s, if default timeslice lengths of 100 ms are taken as a basis. Transmission rates of almost 10 bit/s were determined to be possible without changing the channel model, if only the receiver does not consume full timeslices.

Practical experiments based on a given testing environment showed not only that transmissions through an IRCC are possible, but that transmission rates of up to 4.9 bit/s could be achieved, if exploit parameters are setup well and optimal encodings were used. Even simple encodings however turned out to permit very low error rates while the transmission rate still stays above 1 bit/s.

The influence of exploit parameters on the bandwidth has been dealt with by a generic framework that identifies interdependencies between the properties of a target system, the configuration of the exploit, and its performance. This allowed for adjusting exploit parameters towards their optimal values, as it was demonstrated by applying the framework to a concrete instance of a computer system.

The channel model that does not include the possibility of falsely positive interrupt detections was justified by analyzing noise and presenting techniques for reducing or eliminating it.

4.5. Outlook

The practical implementation of covert channels and, in particular, interrupt-related covert channels offers interesting questions to be solved, as this thesis should have shown. This section lists some aspects that could be subject of further research on the topic of IRCCs.

Autoconfiguration. The list of parameters introduced for the generic framework in Section 3.3.1 and the remarks of Section 4.2 already suggest that an attacker might have to know very much about the target system in order to configure an IRCC exploit properly. In cases where the attacker has no or only very limited access to this system it would therefore be desirable to have sender and receiver configure themselves as soon as they are run on the target. For few of the exploit parameters this is already done (namely only t_g in the sender and `base` of Listing 2.2 on page 13 in the receiver). It would be very interesting

to see how the system parameters d and c_i can be determined without human interaction, especially if the system is rather noisy. Considering e.g. Figure 3.6a on page 57, this might finally lead to the application of pattern recognition algorithms. Once the values for the aforementioned parameters are known, it is rather simple to automatically determine the remaining framework parameters like D or f. An implementation of the generic framework, potentially reduced to some relevant special cases, could then finally compute the actual exploit parameters.

Inconspicuousness. Sender and receiver of the exploit presented in Chapter 2 are CPU hogs. The former most of the time repeatedly queries the clock while waiting for the end of its timeslice, the latter repeatedly executes some portion of code to see whether its runtime has been elongated. The resulting permanently high CPU utilization and/or a faster rotating CPU fan, including the induced acoustic noise level, might thus quickly reveal the presence of the exploit. It would therefore be interesting to see whether and how the implementation of a less conspicuous IRCC exploit is possible.

Channel improvements. The ideas discussed as alternatives in Sections 3.4.2 and 3.4.3 might be more challenging to implement but potentially result in remarkably higher bandwidths. In particular, it would be interesting to find out why the receiver implementation using shortened timeslice resulted in a significantly increased interrupt loss rate.

Tickless kernel. Current Linux kernels[1] offer a “tickless kernel” whose intention is to reduce the power consumption by avoiding unnecessary timer interrupts. However, timer interrupts are an important part of the generic framework and are also expected to be the culprit for the differences between the histograms of Figure 3.6 on page 57. It would, thus, be interesting to see whether a tickless kernel results in higher or even lower bandwidths, and how the framework would have to be adapted.

Countermeasures. The novel countermeasures proposed in Section 4.1 should probably deserve further attention in terms of a theoretical and practical evaluation. For the first countermeasure, inter-timeslice gaps, the maximal interrupt delay of contemporary hardware should be determined, to obtain an estimation of the required gap between the tasks’ timeslices. It would also be interesting to see whether and how an operating system could automatically determine the necessity and duration of a gap based on the system call history.

For the second countermeasure, randomized timeslice lengths, it would be interesting to evaluate the impact on the exploit implementation presented here and devise improved exploits if possible.

[1] At the time of writing, this is version 2.6.27.

A. Proofs

A.1. Lossy Noiseless Transmission

Proof of Theorem 3.1, page 32. Let $\pi := p(X{=}\mathbf{1})$. Then $p(X{=}\mathbf{0}) = 1-\pi$ and applying the definition of mutual information gives

$$\begin{aligned} I(X;Y) &= \sum_{x,y} p(x,y) \cdot \log \frac{p(x,y)}{p(x)p(y)} = \sum_{x,y} p(y|x)p(x) \cdot \log \frac{p(y|x)}{p(y)} \\ &= (1-\pi) \sum_{y} p(y|X{=}\mathbf{0}) \log \frac{p(y|X{=}\mathbf{0})}{p(y)} + \pi \sum_{y} p(y|X{=}\mathbf{1}) \log \frac{p(y|X{=}\mathbf{1})}{p(y)}. \end{aligned}$$

Since for $X{=}\mathbf{0}$ there is only the output $y = 0$ with a positive probability, the first sum collapses to a single summand, which can even be further simplified due to $p(Y{=}0|X{=}\mathbf{0}) = 1$. This yields

$$I(X;Y) = -(1-\pi)\log p(Y{=}0) + \pi \sum_{y} p(y|X{=}\mathbf{1}) \log \frac{p(y|X{=}\mathbf{1})}{p(y)}.$$

The second sum can now be splitted into the cases $y = 0$ and $0 < y \leq N$ (for $y > N$, $p(y|X{=}\mathbf{1}) = 0$ anyway). Recall from the theorem that $\rho := p(Y{=}0|X{=}\mathbf{1})$. Then

$$\begin{aligned} I(X;Y) &= -(1-\pi)\log p(Y{=}0) + \pi\rho \log \frac{\rho}{p(Y{=}0)} + \pi \sum_{0<y\leq N} p(y|X{=}\mathbf{1}) \log \frac{p(y|X{=}\mathbf{1})}{p(y)} \\ &= -(1-\pi+\pi\rho)\log p(Y{=}0) + \pi\rho\log\rho + \pi \sum_{0<y\leq N} p(y|X{=}\mathbf{1}) \log \frac{p(y|X{=}\mathbf{1})}{p(y)}. \end{aligned}$$

From $p(y) = \sum_x p(y|x)p(x)$ (according to the law of total probability) one can easily obtain $p(Y{=}0) = 1 - \pi(1-\rho)$ and, for $y > 0$, $p(y) = \pi p(y|X{=}\mathbf{1})$. This leads to

$$\begin{aligned} I(X;Y) &= -(1-\pi(1-\rho))\log(1-\pi(1-\rho)) + \pi\rho\log\rho - \pi\log\pi \sum_{0<y\leq N} p(y|X{=}\mathbf{1}) \\ &= -(1-\pi(1-\rho))\log(1-\pi(1-\rho)) + \pi\rho\log\rho - (1-\rho)\pi\log\pi. \qquad \text{(A.1)} \end{aligned}$$

The capacity $Cap(I,O,P)$ is defined as the maximum of the mutual information $I(X;Y)$ between input and output, with respect to the input distribution. This input distribution is represented by the single parameter $\pi = p(X{=}\mathbf{1})$ (obviously, the choice of $p(X{=}\mathbf{0})$ would also have been valid). The maximum[1] can be computed by solving $\frac{d}{d\pi} I(X;Y) = 0$ for π.

[1] The extremal value must be a maximum, since $I(X;Y)$ is concave [CT06, Theorem 2.7.4].

Since π appears in $I(X;Y)$ twice in the form $f(\pi) \cdot \log f(\pi)$, this expression is first generally derived, where $f'(\pi) := \frac{d}{d\pi} f(\pi)$:

$$\begin{aligned}\frac{d}{d\pi} f(\pi) \log_2 f(\pi) &= f'(\pi) \log_2 f(\pi) + f(\pi) \cdot \frac{f'(\pi)}{f(\pi) \ln(2)} \\ &= f'(\pi) \cdot (\log_2 f(\pi) + \log_2(e)) = f'(\pi) \cdot \log_2(e f(\pi)).\end{aligned}$$

Then by utilizing the above result for the derivation one obtains

$$\begin{aligned}\frac{d}{d\pi} I(X;Y) &= (1-\rho) \log(e(1-\pi(1-\rho))) + \rho \log \rho - (1-\rho) \log(e\pi) \\ &= \rho \log \rho + (1-\rho) \log \frac{1-\pi(1-\rho)}{\pi} \\ &= \rho \log \rho + (1-\rho) \log\left(1/\pi - (1-\rho)\right) \stackrel{!}{=} 0.\end{aligned}$$

Now it shows that $\rho \neq 1$ or, equivalently, $\lambda \neq 1$ must hold for π to remain in the equation. This is intuitive, since $\lambda = 1$ means all interrupts are lost and thereby no information would be transmitted, i.e. $I(X;Y) \equiv 0$.

If $\rho < 1$, then solving for π is easy as it appears only once in the equation:

$$\pi = \left(1 - \rho + 2^{-\frac{\rho \log \rho}{1-\rho}}\right)^{-1} = \left(1 - \rho + \rho^{-\rho/(1-\rho)}\right)^{-1}. \qquad \square$$

Lemma A.1. *Given a channel as in Theorem 3.1, a loss rate of $\lambda = 1$ results in $Cap = 0$.*

Proof. If $\lambda = 1$, then $\rho := p(Y{=}0|X{=}\mathbf{1}) = 1$ (as of Equation (3.2)). Applying this value to Equation (A.1) immediately leads to $I(X;Y) = 0$, independent of the input distribution. By the definition of Cap as the maximum mutual information over all input distributions, also $Cap = 0$ follows. $\square$

Theorem A.1. *Let $N \in \mathbb{N}$ be a fixed integer and, for $\lambda \in [0,1]$ and $y \in \{0,\ldots,N\}$, let $f(y,\lambda) = \binom{N}{y} \lambda^{N-y} (1-\lambda)^y$. Define $L(y_1,\ldots,y_n;\lambda) := \prod_{i=1}^n f(y_i,\lambda)$. For a fixed sequence $y_1,\ldots,y_n \in \{0,\ldots,N\}$ define $\bar{y} := \frac{1}{n}\sum_{i=1}^n y_i$.*

Then $\lambda = 1 - \bar{y}/N$ maximizes $L(y_1,\ldots,y_n;\lambda)$ for fixed $y_1,\ldots,y_n$.

Proof. For the following let the (nonempty) sequence $y_1,\ldots,y_n \in \{0,\ldots,N\}$ for $n \geq 1$ be fixed and define the mean of the sequence as $\bar{y} := \frac{1}{n}\sum_{i=1}^n y_i$. Then

$$\begin{aligned}L(y_1,\ldots,y_n;\lambda) &= \prod_{i=1}^n f(y_i,\lambda) = \prod_{i=1}^n \binom{N}{y_i} \lambda^{N-y_i} (1-\lambda)^{y_i} \\ &= \lambda^{n(N-\bar{y})} \cdot (1-\lambda)^{n\bar{y}} \cdot \prod_{i=1}^n \binom{N}{y_i}\end{aligned}$$

At first the special cases $\bar{y} = 0$ (possible only if all $y_i = 0$) and $\bar{y} = N$ (possible only if all $y_i = N$) are considered. In the first case, L simplifies to

$$L(y_1,\ldots,y_n;\lambda) = \lambda^{nN},$$

which is obviously maximized by $\lambda = 1$, the maximal value allowed for λ. Similarly, the second special case leads to

$$L(y_1, \ldots, y_n; \lambda) = (1 - \lambda)^{nN},$$

which is maximized by $\lambda = 0$, the smallest value allowed for λ. Both results agree with the theorem, claiming that $\lambda = 1 - \bar{y}/N$ maximizes L.

Finally the "normal" case $0 < \bar{y} < N$ has to be looked at. Therefore it can firstly be observed that $L(y_1, \ldots, y_n; \lambda) = 0$ if and only if either $\lambda = 0$ and $\bar{y} \neq N$, or $\lambda = 1$ and $\bar{y} \neq 0$. Thus, for $\bar{y} \notin \{0, N\}$ a value of $\lambda = 0$ or $\lambda = 1$ cannot result in a maximum of L (all $\lambda \in (0, 1)$ would lead to a $L(y_1, \ldots, y_n; \lambda) > 0$). In the following it is therefore assumed that $\lambda \in (0, 1)$.

A necessary condition for a maximum of L is $\frac{\partial}{\partial\lambda} L(y_1, \ldots, y_n; \lambda) = 0$:

$$\begin{aligned}\frac{\partial}{\partial\lambda} L(y_1, \ldots, y_n; \lambda) &= n\lambda^{n(N-\bar{y})}(1-\lambda)^{n\bar{y}} \left(\frac{N-\bar{y}}{\lambda} - \frac{\bar{y}}{1-\lambda}\right) \cdot \prod_{i=1}^{n} \binom{N}{y_i} \\ &= n \underbrace{\left(\frac{N-\bar{y}}{\lambda} - \frac{\bar{y}}{1-\lambda}\right)}_{=:\alpha(\bar{y},\lambda)} \cdot L(y_1, \ldots, y_n; \lambda)\end{aligned}$$

Here, $\alpha(\bar{y}, \lambda) = 0$ is a necessary and sufficient condition for $\frac{\partial L}{\partial\lambda} = 0$, as all other factors are strictly positive. One can easily convince oneself that $\alpha(\bar{y}, \lambda) = 0$ is equivalent to the claim of the theorem, $\lambda = 1 - \bar{y}/N$.

A sufficient condition for this solution to be a maximum of L is $\frac{\partial^2}{\partial\lambda^2} L(y_1, \ldots, y_n; \lambda) < 0$. Therefore the partial derivation of α is easily computed:

$$\frac{\partial}{\partial\lambda} \alpha(\bar{y}, \lambda) = -\frac{N-\bar{y}}{\lambda^2} - \frac{\bar{y}}{(1-\lambda)^2}.$$

This then almost immediately leads to

$$\frac{\partial^2}{\partial\lambda^2} L(y_1, \ldots, y_n; \lambda) = nL(y_1, \ldots, y_n; \lambda) \left(n\alpha^2(\bar{y}, \lambda) + \frac{\partial}{\partial\lambda} \alpha(\bar{y}, \lambda)\right)$$

For $\lambda = 1 - \bar{y}/N$ it is known from before that $\alpha(\bar{y}, \lambda) = 0$. Furthermore, $n > 0$ and $L(y_1, \ldots, y_n; \lambda) > 0$ and only $\frac{\partial}{\partial\lambda} \alpha(\bar{y}, \lambda) < 0$. Thus, for the extremal value $\lambda = 1 - \bar{y}/N$ the second partial derivation of L is indeed negative, confirming the maximum. □

A.2. Thresholds

Proof of Theorem 3.2, page 35. Firstly, for each $\tau \in \mathbb{N}$ let $Y_\tau = \mathfrak{T}_\tau(Y)$ be the random variable modeling the output of the channel (I, I, P_τ). Since Y_τ is thus independent of X, once Y is given, the data-processing inequality [CT06, Theorem 2.8.1] can be applied. This gives

$$I(X; Y) \geq I(X; \mathfrak{T}_\tau(Y)) = I(X; Y_\tau)$$

for all $\tau \in \mathbb{N}$ (and all input distributions $p(X)$). Consequently also

$$Cap(I, I, P_\tau) = \max_{p(X)} I(X; Y_\tau) \leq \max_{p(X)} I(X; Y) = Cap(I, O, P)$$

by the definition of channel capacity. Since this last inequality holds true for all $\tau \in \mathbb{N}$, it follows that

$$Cap(I,O,P) \geq \max_{\tau\in\mathbb{N}} Cap(I,I,P_\tau). \tag{A.2}$$

What now remains to be shown is that $Cap(I,O,P) = Cap(I,I,P_1)$. Therefore it will first be demonstrated that $I(X;\mathfrak{T}_1(Y)) = I(X;Y)$:

$$\begin{aligned} I(X;\mathfrak{T}_1(Y)) &= \sum_{x,y\in I} p(\mathfrak{T}_1(Y){=}y|x)p(x)\log\frac{p(\mathfrak{T}_1(Y){=}y|x)}{p(\mathfrak{T}_1(Y){=}y)} \\ &= \sum_{x\in I} p(\mathfrak{T}_1(Y){=}\mathbf{0}|x)p(x)\log\frac{p(\mathfrak{T}_1(Y){=}\mathbf{0}|x)}{p(\mathfrak{T}_1(Y){=}\mathbf{0})} \\ &\quad+ \sum_{x\in I} p(\mathfrak{T}_1(Y){=}\mathbf{1}|x)p(x)\log\frac{p(\mathfrak{T}_1(Y){=}\mathbf{1}|x)}{p(\mathfrak{T}_1(Y){=}\mathbf{1})}. \end{aligned}$$

For the first sum, input $X = \mathbf{0}$ leads to $p(\mathfrak{T}_1(Y){=}\mathbf{0}|X{=}\mathbf{0}) = p(Y{=}0|X{=}\mathbf{0}) = 1$ and the term for input $X = \mathbf{1}$ is abbreviated using $p(\mathfrak{T}_1(Y){=}\mathbf{0}|X{=}\mathbf{1}) = p(Y{=}0|X{=}\mathbf{1}) =: \rho$. For the second sum, $p(\mathfrak{T}_1(Y){=}\mathbf{1}|X{=}\mathbf{0}) = p(Y{>}0|X{=}\mathbf{0}) = 0$ (which makes the summand for $x = \mathbf{0}$ vanish) and $p(\mathfrak{T}_1(Y){=}\mathbf{1}|X{=}\mathbf{1}) = p(Y{>}0|X{=}\mathbf{1}) = 1-\rho$. It then follows

$$\begin{aligned} I(X;\mathfrak{T}_1(Y)) &= -(1-\pi)\log p(Y{=}0) + \pi\rho\log\frac{\rho}{p(Y{=}0)} + \pi(1-\rho)\log\frac{1-\rho}{P(Y{>}0)} \\ &= -(1-\pi+\pi\rho)\log p(Y{=}0) + \pi\rho\log\rho + \pi(1-\rho)\log\frac{1-\rho}{P(Y{>}0)}. \end{aligned}$$

From the proof of Theorem 3.1 it is known that $p(Y{=}0) = 1-\pi(1-\rho)$ and, hence, $p(Y{>}0) = \pi(1-\rho)$. This leads to

$$I(X;\mathfrak{T}_1(Y)) = -(1-\pi(1-\rho))\log(1-\pi(1-\rho)) + \pi\rho\log\rho - (1-\rho)\pi\log\pi,$$

which equals $I(X;Y)$ as of Equation (A.1) on page 67. This hence for all π (i.e. for all input distributions) proves $I(X;\mathfrak{T}_1(Y)) = I(X;Y)$. An immediate result of the identical mutual information is the identical capacity:

$$Cap(I,I,P_1) = \max_{p(X)} I(X;\mathfrak{T}_1(Y)) = \max_{p(X)} I(X;Y) = Cap(I,O,P). \tag{A.3}$$

Since $Cap(I,I,P_1) \leq \max_{\tau\in\mathbb{N}} Cap(I,I,P_\tau)$ has to hold due to the definition of the maximum, the previous results can be combined to yield first

$$Cap(I,I,P_1) \leq \max_{\tau\in\mathbb{N}} Cap(I,I,P_\tau) \overset{\text{(A.2)}}{\leq} Cap(I,O,P) \overset{\text{(A.3)}}{=} Cap(I,I,P_1)$$

and, since the first and the last term of the inequality are the same, finally also gives the claimed equality of the theorem. □

Proof of Theorem 3.3, page 35. Firstly it will be shown that $Cap(I,I,P_0) = 0$, again via the mutual information: By definition $I(X;\mathfrak{T}_0(Y)) = H(\mathfrak{T}_0(Y)) - H(\mathfrak{T}_0(Y)|X)$ and, since entropies are always nonnegative, $I(X;\mathfrak{T}_0(Y)) \leq H(\mathfrak{T}_0(Y))$. In this case, as $\mathfrak{T}_0(y) = 1$ for all $y \in O$, the entropy must be zero and hence so is the mutual information and as well the capacity.

Secondly it is shown that $p(\mathfrak{T}_1(Y) \neq X) \leq p(\mathfrak{T}_\tau(Y) \neq X)$ holds for all $\tau > 0$ ($\tau = 0$ can be excluded because it has already been shown that $Cap(I, I, P_0) = 0$ and, thus, this τ does not satisfy the requirements of the theorem).

$$\begin{aligned} p(\mathfrak{T}_\tau(Y){\neq}X) &= \sum_x p(x)p(\mathfrak{T}_\tau(Y) \neq x|x) \\ &= p(X{=}\mathbf{0})p(\mathfrak{T}_\tau(Y){\neq}0|X{=}\mathbf{0}) + p(X{=}\mathbf{1})p(\mathfrak{T}_\tau{\neq}1|X{=}\mathbf{1}) \\ &= (1-\pi)p(Y{\geq}\tau|X{=}\mathbf{0}) + \pi p(Y{<}\tau|X{=}\mathbf{1}) \end{aligned}$$

Since $p(Y{>}0|X{=}\mathbf{0}) = 0$ and, hence, especially $p(Y{\geq}\tau|X{=}\mathbf{0}) = 0$ for all $\tau > 0$, the equation can be simplified to

$$p(\mathfrak{T}_\tau(Y){\neq}X) = \pi p(Y{<}\tau|X{=}\mathbf{1}) = \pi \sum_{y=0}^{\tau-1} p(y|X{=}\mathbf{1}).$$

All probabilities $p(y|X{=}\mathbf{1})$ are nonnegative and, thus, $p(\mathfrak{T}_\tau(Y){\neq}X)$ is monotonically increasing with τ. Therefore it has to hold that $p(\mathfrak{T}_1(Y){\neq}X) \leq p(\mathfrak{T}_\tau(Y){\neq}X)$ for all $\tau > 0$. □

A.3. Hamming Bit Error Rate

This section is concerned with Hamming codes as introduced, e.g., in [CT06, Section 7.11]. These codes can detect and correct single-bit errors within a block of l bits. For a block length of $l = 2^k - 1$, at least k bits must be "parity bits" which contain redundant information. The goal of this section is to compute the bit error rate $E_k(\varphi)$ for the $l-k$ data bits in a block of length l, if for *all* bits the error probability is φ. The results can then be compared, for instance, against the error rate of a transmission without any error correction. In will in the following be assumed that the single bit errors are pairwise stochastically independent.

Capturing bit corrections in an exact closed formula seems to be cumbersome and would go beyond the scope of this thesis. Therefore, a simpler approach is taken: For a given block length $l = 2^k - 1$, it is assumed that the input is a sequence of $l - k$ zeros. All possible outputs are iterated through and are applied the error correction. The data bits of the resulting bit sequence are then compared with the all-zeros input and the number of differing bits is counted. As the probability of each enumerated output sequence can be computed (based on φ), the expected number of bit errors can be computed with this method. One can easily convince oneself that the results do not only apply to zero-only but also to arbitrary input sequences of length $l - k$.

The disadvantage of enumerating outputs is of course that it scales badly with the block length: For a block length of l bits, 2^l outputs must be generated and processed. The results of this section are therefore practically limited to $k \leq 5$ error bits.

Let

- k be the number of parity check bits of the Hamming code of interest, $l = 2^k - 1$ be the block length,
- $C_k : \mathbb{B}^l \to \mathbb{B}^l$ be the function which does the bit correction, where $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ is the Boolean domain and $\mathbb{B}^l$ thus is the set of Boolean vectors (bit vectors) of length l,
- $D_k : \mathbb{B}^l \times \mathbb{B}^l \to \{0, \ldots, l\}$ be the Hamming distance of two Boolean vectors, i.e.

$$D_k(x, y) = |\{i \in \{1, \ldots, l\} \mid x_i \neq y_i\}|,$$

where x_i is the i-th bit of the bit vector x.
Given a bit error probability φ for the transmission (i.e. *not* for the result after applying C_k), the probability of obtaining output $y \in \mathbb{B}^l$ from input $x \in \mathbb{B}^l$ only depends on the number of differing bits:

$$p(y|x) = \varphi^d(1-\varphi)^{l-d}, \qquad d := D_k(x,y).$$

Thus, it is possible to do the enumeration of all output values only once for every k, independent of φ, by computing

$$s^i_{k,d} := |\{y \in \mathbb{B}^l \mid D_k(x,y) = d \wedge C_k(y)_i \neq x_i\}| \qquad \text{for } x = \vec{0}.$$

This definition makes $s^i_{k,d}$ be the number of possible outputs whose Hamming distance to the input is exactly d and whose i-th bit (potentially among other bits) is wrong.

For the i-th bit in the block, the bit error rate is then determined by

$$\begin{aligned} E^i_k(\lambda) &= \sum_{\substack{y\in\mathbb{B}^l \\ C_k(y)_i\neq x_i}} p(y|x) = \sum_{d=0}^{l} \sum_{\substack{y\in\mathbb{B}^l \\ D_k(x,y)=d \\ C_k(y)_i\neq x_i}} p(y|x) = \sum_{d=0}^{l} \sum_{\substack{y\in\mathbb{B}^l \\ D_k(x,y)=d \\ C_k(y)_i\neq x_i}} \varphi^d(1-\varphi)^{l-d} \\ &= \sum_{d=0}^{l} \varphi^d(1-\varphi)^{l-d} \cdot s^i_{k,d} \end{aligned}$$

The first result of the enumeration is that all bits inside a block are equal, i.e. $s^i_{k,d} = s^j_{k,d}$ for all $i,j \in \{0,\dots,l\}$, as expected. Therefore, let $s_{k,d} := s^i_{k,d}$ for arbitrary i. Analogously $E_k(\varphi) := E^i_k(\varphi)$. Table A.1 shows the $s_{k,d}$ resulting from enumerating all outputs and the values of $E_k(\varphi)$ for some chosen φ.

A.4. Generic Framework

Proof of Theorem 3.4, page 50. Firstly, all extremal values of $g(t)$ are found by equating the first derivation with 0. Since only positive extremal values are of interest, $t \geq 0$ is implicitly assumed below.

$$g'(t) = -\varphi \cdot \frac{t}{t+c}e^{-t\varphi} + \frac{c}{(t+c)^2}e^{-t\varphi} = \underbrace{\frac{e^{-t\varphi}}{(t+c)^2}}_{\gneq 0} \cdot (c - t\varphi(t+c)).$$

It follows that $g'(t^*) = 0$ iff $t^*(t^*+c) = c/\varphi$, which immediately resolves to

$$t^* = -c/2 \pm \sqrt{(c/2)^2 + c/\varphi},$$

where only the claimed value is positive. Furthermore, $g'(t) > 0$ iff $t(t+c) < c/\varphi$, which easily implies $g'(t) > 0$ for $t \in [0, t^*)$ and therefore confirms the theorem's claimed monotonicity.

d	$k=2$	$k=3$	$k=4$	$k=5$
0	0	0	0	0
1	0	0	0	0
2	3	9	21	45
3	1	19	119	575
4		16	392	4,760
5		12	1,036	30,828
6		7	2,093	156,793
7		1	3,067	639,535
8			3,368	2,154,360
9			2,912	6,115,200
10			1,967	14,800,929
11			973	30,817,059
12			336	55,619,200
13			84	87,527,020
14			15	120,567,645
15			1	145,732,275
16				154,807,920
17				144,614,880
18				118,726,055
19				85,501,325
20				53,855,256
21				29,551,236
22				14,044,875
23				5,734,365
24				1,990,040
25				579,488
26				139,083
27				26,705
28				3,920
29				420
30				31
31				1

k	φ = 10%	φ = 1%
2	$1.73 \cdot 10^{-2}$	$1.52 \cdot 10^{-4}$
3	$4.59 \cdot 10^{-2}$	$4.51 \cdot 10^{-4}$
4	$7.92 \cdot 10^{-2}$	$10.16 \cdot 10^{-4}$
5	$9.99 \cdot 10^{-2}$	$20.16 \cdot 10^{-4}$

Table A.1. Distance-producing outputs $s_{k,d}$ (left) and bit error rate $E_k(\varphi)$ (right).

What remains to be verified is that $g(t)$ really assumes a maximum at the given value. Therefore, first $g'(t)$ is transformed, exploiting that $t^* \neq 0$ (for the division by t):

$$\begin{aligned} g'(t) &= \frac{e^{-t\varphi}}{(t+c)^2} \cdot (c - t\varphi(t+c)) = \frac{t \cdot e^{-t\varphi}}{t+c} \cdot \left(\frac{c}{t(t+c)} - \varphi\right) \\ &= g(t) \cdot \left(\frac{c}{t(t+c)} - \varphi\right) \end{aligned}$$

To confirm the maximum, the second derivation must be negative:

$$g''(t) = g'(t) \cdot \left(\frac{c}{t(t+c)} - \varphi\right) - \underbrace{g(t) \cdot \frac{c(2t+c)}{t^2(t+c)^2}}_{>0}$$

At the extremal value, $g'(t) = 0$ by construction, so that the first term of $g''(t)$ vanishes, leaving only a negative remainder. $\square$

B. Source Code

B.1. Adapted Exploit

The full sources of the adapted exploit consist of approximately 2000 lines of C++ code (including comments and empty lines for better readability) of which only those parts are shown here, which are relevant for understanding the implementation of the IRCC. This explicitly *excludes*:

Program initialization. Sender and receiver of the exploit are configured through command line arguments which have to be parsed at the beginning of the respective main() function.

Debugging parts. Some parts have been added to the exploit code to obtain further information from the IRCC, such as a histogram of interruption durations or an automatic computation of a channel matrix.

External utilities. Several tools have been programmed for collecting and evaluating data, like measuring timer interrupt and scheduling durations (c_i of the framework), NIC interrupt generation times (t_g) or the timer interrupt frequency (f). The code related to measuring interruptions often consists mainly of simplified receiver code (repeatedly executing the measure_duration() function). Evaluation code is mostly not more than computing mean values and standard deviations.

In the following, sender and receiver implementation are presented separately in this order, for each starting with the main function for sending and, respectively, receiving a bit sequence via an IRCC. For both programs, the listing of the main function is then followed by listings containing helper functions and classes.

For the beginning, Listing B.1 shows a small set of types used by sender and receiver that have been defined to simplify reading and understanding the exploit code.

```
typedef unsigned int uint;

typedef unsigned int usecs_t; /* Integral microseconds. */
typedef double fusecs_t; /* Floating-point microseconds. */

typedef unsigned long long timing_t;
```

Listing B.1. Shortcut types.

B.1.1. Sender Implementation

The sender implementation makes use of the helper classes Waiter (Listing B.4, page 79), Generator (Listing B.5, page 80) and DataContainer (Listing B.6, page 82), as well as the small helper function get_next_sync() (Listing B.3, page 79). The main sending function is shown in Listing B.2 on page 78.

Main function. Function send() in Listing B.2 is the concrete counterpart to Listing 2.8 on page 23. It can be divided into three parts: Initialization, transmission, and output. The initialization partly happens already before the function is executed, when the function's parameter objects are constructed (see below). In this function, the only notable initialization is the one performed by the get_next_sync() function (see Listing B.3). Depending on the sync parameter this function returns the time of the determined beginning of the transmission in a **struct** timeval, here T_begin. As the function's code shows, transmissions always begin when the number of seconds of the system clock next time reaches an integral multiple of sync.

The well commented transmission code consists of an endless loop which starts with a sched_yield () to make sure that the following code is executed in a new timeslice. Afterwards, the code waits until the expected end of this new timeslice and queries the clock to determine which bit index (bit_idx) is to be transmitted at exactly this point in time. If the bit index exceeds the number of bits to be sent (bit<0, see Listing B.6), then the transmission loop is left. Otherwise, if the bit is nonzero, then the sending statistics are updated and an interrupt is generated. If the bit is zero, then nothing is done and the transmission loop starts again at the beginning.

The output part of the function does nothing relevant for the transmission but simply shows how many timeslices have been counted and how many times the interrupt-generating function has been called.

Waiter. The intention of the Waiter class is to provide a simple way of waiting for a specified amount of time. Listing B.4 shows that the main function of the class has been realized by simply querying the clock in a busy loop (i.e. without yielding the CPU intermediately). The alternative of setting up a timer (e.g. via setitimer ()) seems to be ineligible due to the limited precision of normal timers. High precision timers (HPETs) have not been considered as they were not available on the testing system.

The waiter object is constructed with the number of microseconds that have to be waited when the object's main function is called. On initialization of the sender process, this duration is set to $t_s - t_g - \varepsilon'$, where t_s is taken from a command line argument, t_g is empirically determined by generating a small number of interruptions and, finally, $\varepsilon' > 0$ is an additional pre-position amount. This way was preferred to directly specifying ε as $\varepsilon \geq t_g$ was determined to be necessary anyway.

Generator. The generation of interrupts has been implemented with flexibility in mind. Therefore, the class for generating NIC interrupts has been derived from a generic interrupt generation class, named base (contained in the C++ namespace Generator). In principle, classes for arbitrary other interrupt sources can be added to the exploit this way.

The main function for generating NIC interrupts, NIC::**operator**()(), is nothing more than an encapsulation of the sendto() system function. All parameters to the function, like the

socket, the payload and the packet destination, have already been initialized by payload_init () and sock_init (). Besides the setup of network addresses and ports, the initialization code makes the socked nonblocking and turns the socket into a broadcast socket if the destination address seems to be a broadcast address. The advantages of broadcast addresses is that they can be used in a testing network without any other computers attached. In such a setting, normal IP addresses would lead to a failing Ethernet address lookup and, in consequence, no sent packets except for the ARP (address resolution protocol) packets.

The idea of generating multiple interrupts to transmit more than just binary symbols (see Section 3.4.3), is already implemented in the class, especially NIC::**operator**()(). That is to say, the latter function determines the length of the payload to be sent based on the input. However, this functionality does not have to be made use of.

Data container. The DataContainer (Listing B.6) manages a simple data structure that holds the sequence to be sent. For the elements of the sequence, decimal digits are allowed, but which values make sense depends on the used interrupt generator class and whether it supports more values than the Boolean ones. If the data passed to the constructor is shorter than the number nbits to be managed, then the data sequence is repeated sufficiently often. (This allows for a succinct way of telling the sender to transmit, e.g., the sequence $(\mathbf{10})^k$.) During the transmission, only the get() member function is used, which returns -1 if the index is out of range and the transmission should, thus, be terminated.

```
bool send(Generator::base &generator, DataContainer &data,
      const Waiter &waiter, uint sync, fusecs_t duration,
      int verbosity)
{
   unsigned int slices=0, ints=0, errs=0;
   struct timeval T, T_begin;
   std::vector<unsigned int> counts(data.get_bits(), 0);

   if (!get_next_sync(sync, T_begin)) return false;
   for (;;) {
      sched_yield(); // Ensure beginning of new timeslice
      waiter(); // Wait until close before end of timeslice
      gettimeofday(&T, NULL); // Get current time
      if (timercmp(&T, &T_begin, <))
         continue; // Transmission not started
      timersub(&T, &T_begin, &T); // T since begin of transmission
      size_t bit_idx = (size_t)(tv_micros(T) / duration);
      Generator::input_t bit = data.get(bit_idx);
      if (bit < 0) break; // End of transmission
      else if (bit > 0) { // Generate (an) interrupt(s)
         ++ints;
         ++counts[bit_idx];
         if (!generator(bit)) ++errs;
      }
      ++slices;
   }
   if (verbosity) { // Show collected information
      struct timeval T_end;
      gettimeofday(&T_end, NULL);
      std::cout << "End of sending: " << T_end
         << " (" << T << "s)\n";
      if (verbosity>1) {
         size_t n=data.get_bits();
         for (size_t idx=0; idx<n; ++idx)
            std::cout << idx << ": " << counts[idx] << '\n';
      }
      std::cout << slices << " slices, " <<
         ints << " potential interrupts, " <<
         errs << " errors.\n";
   }
   return true;
}
```

Listing B.2. Main function for sending a bit sequence.

```
/* Compute and return the next time at which the seconds of
 * gettimeofday() assumes an integer multiple of 'sync'. */
bool get_next_sync(unsigned int sync, struct timeval &tv_sync)
{
    if (gettimeofday(&tv_sync, NULL) != 0) return false;
    tv_sync.tv_usec = 0; // Wait until full second always!
    tv_sync.tv_sec = sync * (tv_sync.tv_sec / sync + 1);
    return true;
}
```

Listing B.3. Obtaining the next synchronization time.

```
#define USECS_PER_SECOND (1000000)

class Waiter {
    private:
        struct timeval tv;
    public:
        // Initialize to always wait <usecs> microseconds
        Waiter(usecs_t usecs) {
            tv.tv_sec  = usecs / USECS_PER_SECOND;
            tv.tv_usec = usecs % USECS_PER_SECOND;
        }

        // Wait for duration <tv> from the time of the call
        void operator()(void) const {
            struct timeval tv_end, tv_now;
            gettimeofday(&tv_end, NULL);
            timeradd(&tv_end, &tv, &tv_end);
            do {
                gettimeofday(&tv_now, NULL);
            } while (timercmp(&tv_now, &tv_end, <));
        }
};
```

Listing B.4. Waiting until a given point in time.

```
typedef char input_t;

class base {
  protected:
    bool ok;

  public:
    base() : ok(false) {};
    virtual ~base() {};

    virtual bool operator()(input_t input) = 0;
    inline operator bool(void) const { return ok; };

    usecs_t duration(input_t input);
};

class NIC : public base {
  private:
    int sock;
    struct sockaddr_in dest;
    char *payload;
    ssize_t input_unit;
    input_t max_input;
    int flags;

    bool payload_init();
    bool sock_init();
  public:
    NIC(const in_addr &addr, in_port_t port, ssize_t unit_len,
          input_t input_max);
    // params ::= <addr/dotted> ':' <port> ',' <unit-length>
    NIC(const std::string &params, input_t input_max);
    virtual ~NIC();

    virtual bool operator()(input_t input);
};

bool NIC::operator()(input_t input)
{
  if (!input) return true;
  // Input is not checked against max_input!
  ssize_t len = input_unit*(size_t)input;
  return sendto(sock, payload, len, flags, (struct sockaddr*)&dest,
        sizeof(dest)) == len;
}
```

Listing B.5. Interrupt generation (classes and main function).

```
#define BCAST_MASK htonl(0x000000ff)

bool NIC::sock_init() {
   struct sockaddr_in local;
   local.sin_family = dest.sin_family = AF_INET;
   local.sin_addr.s_addr = htonl(INADDR_ANY);
   local.sin_port = htons(0);
   if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) return false;
   // Broadcast destination? Simply check for .255 at end.
   if ((dest.sin_addr.s_addr & BCAST_MASK) == BCAST_MASK) {
      int on = 1;
      if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &on,
              sizeof(on)) != 0) {
         close(sock);
         return false;
      }
   }
   if (bind(sock, (struct sockaddr*)&local, sizeof(local)) < 0) {
      close(sock);
      return false;
   }
   int flags = fcntl(sock, F_GETFL, 0);
   if (flags < 0) return false;
   return fcntl(sock, F_SETFL, flags |= O_NONBLOCK) != -1;
}

bool NIC::payload_init() {
   if (!(payload = new char[input_unit*max_input])) return false;
   // The content itself has no meaning.
   memset(payload, '\0', input_unit*max_input);
   return true;
}

NIC::NIC(const in_addr &addr, in_port_t port,
      ssize_t unit_len, input_t input_max)
   : base(), input_unit(unit_len), max_input(input_max), flags(0) {
   dest.sin_addr   = addr;
   dest.sin_port   = port;
   ok = payload_init() && sock_init();
}

NIC::~NIC() {
   if (sock >= 0) close(sock);
   if (payload) delete [] payload;
}
```

Listing B.5. Interrupt generation (initialization and destruction).

```
class DataContainer {
   private:
      bool ok;
      Generator::input_t *bits, max;
      size_t len, n;
   public:
      DataContainer(const char* const data, size_t nbits);
      ~DataContainer() { if (bits) delete [] bits; }

      inline operator bool(void) const { return ok; }
      Generator::input_t get(size_t idx) const;

      size_t get_bits() const { return n; }
      Generator::input_t get_max() const { return max; }
};

DataContainer::DataContainer(const char* const data, size_t nbits)
   : ok(false), bits(NULL), max(0), len(0), n(nbits)
{
   for (const char* pos=data; *pos; ++pos,++len) {
      // Currently only [0..9]
      if (!(*pos >= '0' && *pos <= '9'))
         return; // invalid
   }
   bits = new Generator::input_t[len];
   if (!bits) return;
   for (unsigned i=0; i<len; ++i) {
      bits[i] = (data[i] - '0');
      if (bits[i] > max) max = bits[i];
   }
   if (n == 0) n = len;
   ok = true;
}

Generator::input_t DataContainer::get(size_t idx) const
{
   if (idx >= n) return -1; // Index out of range
   idx %= len; // Get index inside pattern
   return bits[idx];
}
```

Listing B.6. Storage for a (repeating) bit sequence.

B.1.2. Receiver Implementation

Like the sender implementation, also the receiver code is splitted up into several parts, the most important being a class for evaluating the measured interruptions (Count in Listing B.10, page 87) and a class for coping with time (TimeManager in Listing B.11, page 89).

Based on whether the preprocessor conditional SHORT_RECEIVER is defined or not, the receiver either shortens its timeslices or receives during its full timeslices (see Section 3.4.2). The only positions where this is relevant are Listings B.8 (page 86) and B.9 (page 86).

In the initialization of the receiver there is one part that is worth mentioning: The determination of what is called base in Listing 2.2 on page 13, i.e. the duration t_m of an uninterrupted measuring. To obtain this value, the measure_duration() function is executed repeatedly in a loop for a certain number of times and the minimum of all measured durations is taken. This is based on the assumption that when repeating the measuring often enough then some of the measurings will be uninterrupted (minimal).

Main function. Function receive () of Listing B.7 realizes the main function for the receiving part of an IRCC transmission. It corresponds to the homonymous pseudo code function in Listing 2.9 on page 23. Similar to the sender, it can also be separated into three parts, the initialization, the transmission loop and output code.

In the initialization phase, besides the initialization of variables, also make_short_timeslice () is called. This function is only relevant if shortened timeslices have been activated (at compile time) as otherwise it does not contain any code (see below and/or Listing B.9).

The endless loop for the reception of data from the sender can be split into the two phases that have already been mentioned several times in the main part of the thesis: Measuring and evaluating. The code for the measuring phase is depicted in Listing B.8 and is relatively unspectacular. Between two time measurings, function measure_duration() repeatedly performs a computation (1.5^k for some chosen $k \in \mathbb{N}$) whose result is ignored and whose only intention is to consume a constant amount of CPU time. More elaborate is the evaluation of the measuring's outcome. The aim here is to keep the evaluation phase as short as possible and, thus, to especially not pollute it with avoidable system calls. This led to the case distinction based on the evaluator's return value, which determines whether the transmission is over or not. If the evaluator found out that the measured duration is not within the expected range for sender-initiated interrupts, then it does not query the clock for the current time and does not even know whether the transmission should be over. Consequently it returns eUnchecked to signal to the main function that it might – after a certain number of iterations without the time being checked – want to query the clock manually.

Finally, the output code can basically be reduced to making the evaluator object print out its collected data. It will be described below.

Measuring interruptions. How interruptions are measured has already been shown slightly simplified in Listing 2.2 on page 13. Listing B.8 shows the real code, which does not differ much from the simplification. The code for the read_tsc () function that is passed to measure_duration() by receive () is omitted here. The interested reader is advised to look up how, e.g., the Linux kernel defines the function rdtscll () to see the assembly code for reading from this architecture-dependent CPU register.

Short reception. In order to shorten the receiver's timeslices, timers are set up via the setitimer () system function. For the experiments, the timer interval was set to only few microseconds in order to have the timer occur shortly after the first timer interrupt within the receiver's timeslice. Once the SIGALRM signal is raised, stop_measuring is set to **true** by sig_timer () to interrupt the measuring in measure_duration() and have it return **false**. This result is then handled by the main reception function and makes it call make_short_timeslice (). As Listing B.9 shows, this yields the CPU and sets up the timer again to shorten also the following timeslice.

Evaluation. Similar to the implementation of the interrupt generation, also the interruption evaluation has been made flexible. A base class (again named base, but inside the C++ namespace Evaluator) in Listing B.10 provides the interface for submitting measured interruption durations and for finishing a single bit or the whole transmission.

Derived from this base class is the Count class (same listing), which for each bit only counts the number of interruptions whose durations fall into a certain contiguous range. All values are stored in a vector and are printed out line by line in the finish_transmission () method. The code of this class is rather simple, so no further comments on its functionality are expected to be necessary. What however should be noted is that no bit index determination takes place if the measured interruption being evaluated is not within the specified range. This saves time and shortens the evaluation phase.

Time manager. The time and bit index handling has been swapped out to the TimeManager class of Listing B.11. Its function check_current_bit () is used in the main reception function as well as in the evaluation class described above. The intention of this function is to (a) check whether the transmission has already started, (b) determine the current bit index, (c) return the last handled bit index and also (d) return whether either the last and current bit differ, or the end of the transmission has been reached.

```
bool receive(Evaluator::base<timing_t> &evaluator, TimeManager &tm,
      long measure_iters, const struct itimerval &itv,
      int verbosity) {
   timing_t t, dt;
   unsigned int MU = MAX_UNINTERRUPTED(measure_iters);
   unsigned int n = MU;

   make_short_timeslice(itv);
   for (;;) {
      if (!measure_duration(read_tsc, measure_iters, t, dt)) {
         make_short_timeslice(itv);
         continue;
      }
      switch (evaluator(dt, tm, std::cout)) {
         case Evaluator::eUnchecked: // Time has not been checked
            if (--n == 0) { // Regularly check time "manually"
               n = MU;
               unsigned int old_bit;
               BitStatus bs = tm.check_current_bit(old_bit);
               if (bs == bUnchanged) break;
               evaluator.finish_bit(old_bit, std::cout);
               if (bs == bEOT) goto done;
            }
            break;
         case Evaluator::eNotEOT: // EOT is not reached
            n = MU;
            break;
         case Evaluator::eEOT: // end of transmission (EOT)
            goto done;
      }
   }
done:
   if (verbosity) {
      struct timeval T_end;
      gettimeofday(&T_end, NULL);
      std::cout << "End of reception: " << T_end << '\n';
   }
   evaluator.finish_transmission(std::cout, tm, verbosity>1);
   return true;
}
```

Listing B.7. Main function for receiving a bit sequence.

```
#ifdef SHORT_RECEIVER
// For interrupting measure_duration(), e.g. from signal handler
extern volatile bool stop_measuring;
#endif

#define EXPONENT 20UL

inline bool measure_duration(timing_fn gettime,
      long iterations, timing_t &t, timing_t &dt)
{
   t = gettime();
   unsigned long i,j;
   volatile double v = 1.0;
   for (dt=0,i=iterations; i>0; --i) {
#ifdef SHORT_RECEIVER
      if (stop_measuring) return false;
#endif
      for (j=EXPONENT; j>0; --j) v *= 1.5;
   }
   dt = gettime() - t;
   return true;
}
```

Listing B.8. Measuring interruptions and their duration.

```
#ifdef SHORT_RECEIVER
// Signal handler function for SIGALRM
void sig_timer(int signr) { stop_measuring = true; }
#endif

void make_short_timeslice(const struct itimerval &itv) {
#ifdef SHORT_RECEIVER
   // Yield and setup timer for next short timeslice
   sched_yield();
   stop_measuring = false;
   setitimer(ITIMER_REAL, &itv, NULL);
#endif
}
```

Listing B.9. Enabling shortened receiver timeslices.

```
enum EvaluationStatus { eUnchecked, eEOT, eNotEOT };

template <class value_t, class count_t=unsigned long>
class base {
  protected:
    value_t offset;
    uint bits;
    bool ok;
  public:
    base(value_t val_offset, uint nbits, bool is_ok=false)
      : offset(val_offset), bits(nbits), ok(is_ok) {};
    virtual ~base() {};
    virtual EvaluationStatus operator()(value_t v,
        TimeManager &tm, std::ostream &out) = 0;
    inline operator bool(void) const { return ok; };
    virtual void finish_bit(uint bit_idx, std::ostream &out) = 0;
    virtual void finish_transmission(std::ostream &out,
        const TimeManager &tm, bool verbose) = 0;
};

template <class value_t, class count_t=unsigned long>
class Count : public base<value_t,count_t> {
  private:
    count_t cur;
  protected:
    typedef std::vector<count_t> CountList;
    CountList counts;
    value_t min, max;
  public:
    Count(value_t offset, uint nbits, value_t vmin, value_t vmax)
      : base<value_t,count_t>(offset, nbits), cur(0),
      counts(nbits), min(vmin+offset), max(vmax+offset) {
      this->ok = (max >= min);
    }
    Count(value_t offset, uint nbits, const std::string &params);
    EvaluationStatus operator()(value_t v, TimeManager &tm,
      std::ostream &out);

    void finish_bit(uint bit_idx, std::ostream &out) {
      counts[bit_idx] = cur;
      cur = 0; // Reset counter for next bit
    }
    void finish_transmission(std::ostream &out,
        const TimeManager &tm, bool verbose);
};
```

Listing B.10. Evaluation of measured durations (class declaration).

```
/* Handle a measured value <v> (check if it is in the wanted
 * range [min,max] and, if it is, perform the right action
 * depending on the bit index). */
template <class value_t, class count_t>
EvaluationStatus Count<value_t,count_t>::operator()(value_t v,
        TimeManager &tm, std::ostream &out) {
    if (v < min || v > max) return eUnchecked;
    uint old_bit;
    switch (tm.check_current_bit(old_bit)) {
        case bBeforeTrans: return eNotEOT;
        case bUnchanged: break;
        case bEOT: finish_bit(old_bit, out); return eEOT;
        case bChanged: finish_bit(old_bit, out); break;
    }
    ++cur;
    return eNotEOT;
}

/* Print a list of all collected counts in the format
 * <bit index> ': ' <counted value> */
template <class value_t, class count_t>
void Count<value_t,count_t>::finish_transmission(std::ostream &out,
        const TimeManager &tm, bool verbose)
{
    typename CountList::const_iterator iter,end=counts.end();
    size_t idx=0;
    for (iter=counts.begin(); iter!=end; ++iter,++idx)
        out << idx << ": " << *iter << '\n';
}
```

Listing B.10. Evaluation of measured durations (cont.).

```
enum BitStatus { bBeforeTrans, bUnchanged, bChanged, bEOT };

class TimeManager {
  private:
    struct timeval T_begin;
    uint bits, cur_bit;
    fusecs_t duration, timeslice;
  public:
    TimeManager(uint sync, uint nbits, fusecs_t bit_duration,
          fusecs_t timeslice_length, usecs_t skew)
      : bits(nbits), cur_bit(0), duration(bit_duration),
        timeslice(timeslice_length)
    {
      get_next_sync(sync, T_begin);
      tv_advance(T_begin, skew);
    }

    BitStatus check_current_bit(uint &bit_idx) {
      bit_idx = cur_bit; // return the *old* index
      struct timeval T;
      gettimeofday(&T, NULL); // Get current time
      if (timercmp(&T, &T_begin, <)) // Too early?
        return bBeforeTrans;
      timersub(&T, &T_begin, &T); // T since transmission begin
      cur_bit = (uint)(tv_micros(T) / duration);
      if (cur_bit == bit_idx)
        return bUnchanged;
      if (cur_bit >= bits)
        return bEOT; // End-Of-Transmission
      return bChanged;
    }

    fusecs_t get_duration() const { return duration; }
    fusecs_t get_timeslice() const { return timeslice; }
};
```

Listing B.11. Time and bit index management.

B.1.3. Original Exploit

For the sake of completeness, the run_for() code that was used in the original exploit code and that is also referenced in the pseudo code of Listings 2.1 (page 12) and 2.2 (page 13), is shown in Listing B.12. It repeats a code block until a given amount of time has elapsed.

Syntactically, run_for() is intended to resemble an ordinary C/C++ control structure (like, e.g., **while**) and is therefore defined as a macro. It sets up a timer and repeats the following code block until the SIGALRM signal, indicating the expiration of the timer, occurred. In consequence, the may actually execute longer than the specified amount of time (either because the signal occurs too late or because the code to be repeated executes too long), but is never interrupted earlier than specified.

Before the run_for() macro is used, the function setup_alarmhandler() must have been called to setup the correct handler for the timer signal. (This overwrites the previous handler and therefore cannot and must not be used with other timers.)

```
/* SIGALRM handling for setitimer(). */
volatile bool awaiting_alarm; /* no optimizations! */
void handle_sigalarm(int sig_nr) { awaiting_alarm = false; }
bool setup_alarmhandler() {
    return signal(SIGALRM, handle_sigalarm) != SIG_ERR;
}

/* Setup an interval timer that fires after <duration> milliseconds
 * from the time the function is called. */
bool set_duration_timer(unsigned int duration)
{
    struct itimerval itv;
    memset(&itv, 0, sizeof(struct itimerval));
    itv.it_value.tv_sec  = duration/MSECS_PER_SECOND;
    itv.it_value.tv_usec = 1000*(duration%MSECS_PER_SECOND);
    return (setitimer(ITIMER_REAL, &itv, NULL) == 0);
}

/* Note: This construct is fragile. Do not use it in code like
 *     if (condition)
 *         run_for(duration) {
 *             stuff();
 *         }
 * as there are no parentheses around the lines of the macro. */
#define run_for(duration) \
    awaiting_alarm = true; \
    if (set_duration_timer(duration)) \
        while (awaiting_alarm)
```

Listing B.12. Repeating code (approximately) for a given amount of time.

```
/* Determine sender and receiver by their effective group ID */
#define IS_QUANTUM_TIME_SENDER(td)    ((td)->td_ucred->cr_gid == 2222)
#define IS_QUANTUM_TIME_RECEIVER(td) ((td)->td_ucred->cr_gid == 2223)
#define IS_QUANTUM_TIME_RELATED(td) \
        (IS_QUANTUM_TIME_SENDER(td) || IS_QUANTUM_TIME_RECEIVER(td))
```

Listing B.13. Determining sender and receiver tasks.

B.2. Modified FreeBSD Scheduler

Linux	FreeBSD
`kernel/sched.c`	`kern/sched_ule.c`
struct rq	**struct** tdq
schedule()	sched_switch(), tdq_choose()
scheduler_tick()	sched_clock()
sched_init()	tdq_setup()

Table B.1

This section shows the modifications made to the FreeBSD 7 scheduler, whose aim were the same as for the Linux scheduler but which were in the end not used for conducting experiments. All listings shown in this section are excerpts from the modified `sys/kern/sched_ule.c`. The original has CVS version 1.214.2.3 (dating April 19, 2008). Table B.1 juxtaposes the names of the affected file and the most relevant functions and data structures for the modification of Linux and FreeBSD.

Note that tdq_choose() in Listing B.14c has been reduced by two KASSERT statements (which result in a kernel panic if a passed condition is satisfied) to ease the understanding of the changes.

In FreeBSD, the timeslices for all tasks (threads) are of the same length of, by default, approximately 100 ms (the number of ticks for achieving this length is stored in the global variable sched_slice). Therefore, this section does not contain any code corresponding to Listing 2.4 on page 19.

(a) Data structure for inclusion into **struct** tdq.

```
/* This structure is added to each CPU's thread queue to achieve
 * a feeling of fixed quanta for a certain subset of all threads.
 */
struct fixed_quantum {
    /* Ticks to wait until next fixed-quantum thread may run. */
    u_int fq_wait_ticks;
    /* Statistics. */
    u_int fq_rejects;  /* Number of rejected fixed-quantum threads */
    u_int fq_max_wait; /* Current maximum value of fq_wait_ticks */
};
```

(b) Recording a leaving sender task in sched_switch().

```
    /* For quantum-time senders, save the number of remaining ticks.
     * Obviously only if the respective thread voluntarily released
     * the CPU or blocked before the slice expired. */
    if (IS_QUANTUM_TIME_SENDER(td))
        fixed_quantum_update_leaving(tdq, ts);
```

(c) Checking whether to block a receiver task with tdq_choose().

```
static struct td_sched *tdq_choose(struct tdq *tdq)
{
    struct td_sched *ts;

    TDQ_LOCK_ASSERT(tdq, MA_OWNED);
    ts = runq_choose(&tdq->tdq_realtime);
    if (ts != NULL && fixed_quantum_run_check(tdq, ts->ts_thread))
        return (ts);
    ts = runq_choose_from(&tdq->tdq_timeshare, tdq->tdq_ridx);
    if (ts != NULL && fixed_quantum_run_check(tdq, ts->ts_thread))
        return (ts);

    ts = runq_choose(&tdq->tdq_idle);
    return (ts);
}
```

(d) Updating data structure in sched_clock().

```
    /* Update the fixed-quantum wait value and
     * request a reschedule, if necessary. */
    if (fixed_quantum_clock_update(tdq))
        td->td_flags |= TDF_NEEDRESCHED;
```

Listing B.14. Steps in blocking receiver tasks.

```
static __inline void fixed_quantum_setup(struct fixed_quantum *fq) {
    /* Initially, a fixed-queue thread is allowed to start,
     * as no other one has already been run. */
    fq->fq_wait_ticks = 0;

    fq->fq_rejects = 0; /* No thread rejected so far. */
    fq->fq_max_wait = 0; /* Nobody waited yet. */
}

static __inline int fixed_quantum_clock_update(struct tdq *tdq) {
    /* Return TRUE iff fq_wait_ticks just changed to 0. */
    if ((tdq)->tdq_fq.fq_wait_ticks == 0) return FALSE;
    return (--(tdq)->tdq_fq.fq_wait_ticks == 0);
}

static __inline void fixed_quantum_update_leaving(struct tdq *tdq,
        struct td_sched *ts) {
    /* When a fixed-quantum thread is stopping its execution, the
     * number of its remaining slice ticks must be stored to avoid
     * scheduling another fixed-quantum thread for this many ticks. */
    struct fixed_quantum *fq = &tdq->tdq_fq;
    fq->fq_wait_ticks = ts->ts_slice;
    /* The following line triggers the 'ts->ts_slice = sched_slice'
     * line in tdq_add in order to make the thread get a full new
     * slice when it is executed next time. */
    ts->ts_slice = 0;
    fq->fq_max_wait = max(fq->fq_wait_ticks, fq->fq_max_wait);
}

static __inline int fixed_quantum_run_check(struct tdq *tdq,
        struct thread *td) {
    /* Non-quantum-time senders may always be scheduled. */
    if (!IS_QUANTUM_TIME_SENDER(td)) return TRUE;

    /* A fixed-quantum thread may only be scheduled if there are no
     * more wait ticks left (from a previous fixed-quantum thread
     * not using its whole ticks). */
    if (tdq->tdq_fq.fq_wait_ticks == 0) return TRUE;
    /* As the fixed-quantum thread will be rejected,
     * the counter is increased. */
    ++tdq->tdq_fq.fq_rejects;
    return FALSE;
}
```

Listing B.15. Helper functions for blocking receiver tasks.

Bibliography

[Aas05] J. Aas. Understanding the Linux 2.6.8.1 CPU scheduler. 2005.

[Ahs00] K. Ahsan. Covert channel analysis and data hiding in TCP/IP. Master's thesis, University of Toronto, 2000.

[AK02] K. Ahsan and D. Kundur. Practical data hiding in TCP/IP. In *Proceedings of the Workshop on Multimedia Security*, 2002.

[BC05] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, third edition, 2005.

[CBS04] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 178–187, 2004.

[CRKH05] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., third edition, 2005.

[CT06] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, second edition, 2006.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[Gir87] C. G. Girling. Covert channels in LAN's. *IEEE Transactions on Software Engineering*, 13(2):292–296, 1987.

[Gli93] V. Gligor. A guide to understanding covert channel analysis of trusted systems. CSC-TG-030, Rainbow Series (Light Pink Book), 1993.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[Gra93] J. W. Gray. On introducing noise into the bus-contention channel. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 90–98, 1993.

[Hoe66] P. G. Hoel. *Introduction to Mathematical Statistics*. John Wiley & Sons, Inc, third edition, 1966.

[Hu91] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[Hus78] J. C. Huskamp. *Covert Communication Channels in Timesharing Systems*. Technical report UCB-CS-78-02, University of California, 1978.

[Kem83] R. A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, 1983.

[Kem02] R. A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proceedings of the Annual Computer Security Applications Conference*, pages 109–118, 2002.

[KW91] P. A. Karger and J. C. Wray. Storage channels in disk arm optimization. In *IEEE Symposium on Security and Privacy*, pages 52–63, 1991.

[Lam73] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[LTWW94] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *Networking, IEEE/ACM Transactions on*, 2(1):1–15, 1994.

[McH95] J. McHugh. Chapter 8: Covert channel analysis from handbook for the computer security certification of trusted systems. Technical Memorandum 5540:080A, Naval Research Laboratory, 1995.

[Mil89] J. K. Millen. Finite-state noiseless covert channels. In *CSFW*, pages 81–86, 1989.

[ML05] S. J. Murdoch and S. Lewis. Embedding covert channels into TCP/IP. In *Information Hiding: 7th International Workshop, volume 3727 of LNCS*, pages 247–261, 2005.

[MM94] I. S. Moskowitz and A. R. Miller. Simple timing channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 56–64, 1994.

[Mol02] I. Molnar. Goals, design and implementation of the new ultra-scalable O(1) scheduler. `http://lxr.linux.no/linux+v2.6.22.9/Documentation/sched-design.txt`, 2002. [Online; accessed 01-December-2008].

[Mol07] I. Molnar. The CFS scheduler. `http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt`, 2007. [Online; accessed 01-December-2008].

[MS07] H. Mantel and H. Sudbrock. Comparing countermeasures against interrupt-related covert channels in an information-theoretic framework. In *20th IEEE Computer Security Foundations Symposium, CSF 2007*, pages 326–340, 2007.

[MS08] H. Mantel and H. Sudbrock. Information-theoretic modeling and analysis of interrupt-related covert channels. In *Pre-Proceedings of the Workshop on Formal Aspects in Security and Trust (FAST)*, 2008.

[Rea97] Realtek Semiconductor Corp. *RTL8029AS - Realtek PCI Full-Duplex Ethernet Controller with built-in SRAM*, 1997.

[Rob03] J. Roberson. ULE: A modern scheduler for FreeBSD. In *Proceedings of the BSD Conference*, pages 17–28, 2003.

[Rus92] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, Computer Science Laboratory, SRI International, 1992.

[Rut04] J. Rutkowska. The implementation of passive covert channels in the Linux kernel. In *21th Chaos Communication Congress*, 2004.

[Sha48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[SMB06] G. Shah, A. Molina, and M. Blaze. Keyboards and covert channels. In *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

[Smi07] M. Smihily. Internet usage in 2007 – households and individuals. Eurostat DATA in focus, 2007.

[Tan01] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, second edition, 2001.

[TG88] C. R. Tsai and V. D. Gligor. A bandwidth computation model for covert storage channels and its applications. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 108–121, 1988.

[Tro98] J. Trostle. Timing attacks against trusted path. In *IEEE Symposium on Security and Privacy*, pages 125–135, 1998.

[Wik08] Wikipedia. G-test — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/G-test`, 2008. [Online; accessed 01-December-2008].

[Wol05] J. Wolf. *Linux-Unix-Programmierung.* Galileo Computing, 2005.

Index

Bernoulli trial, 32
Bit error rate, 35
Bit interval, 11
  length, 11, 29

Capacity, 6, **7**
Channel
  covert, 1, 3
  discrete, 5
  interquantum-time, 4, 16
  interrupt-related, 1, **4**
  memoryless, 5, 6
  overt, 3
  quantum-time, 4, 14, 15
  storage, 3
  timing, 3
Collision (of interrupts), 32, **37**
Compound measuring, **43**, 49
Configuration
  basic, 7
  extended, 9

Distribution
  binomial, 32
  Poisson, 50
  uniform, 44, 51

Entropy, 6

False positives, 27, **54**

G-test, 33
Gap (between bit intervals), 25, 38

Hamming code, 36, **71**

Information
  mutual, **7**, 67, 70
Interrupt, 4
  delay, 37, 41
  duration, 11, 41
  generation, 41
  handler, 4
  request number, 4, 8

Maximum likelihood estimator, 33
Maximum transmission unit, 58
Misaccounting, 28, 33, **44**

Polling, 4, 39

Scheduler, 14
  O(1), 16
  O(n), 16
  4BSD, 16
  CFS, 16
  fixed quantum, 15
  ULE, 16
Skew, 25, 38

Task, 16
  receiver, 15, 17
  sender, 15, 17
Threshold, 14, **35**
Timer
  interrupt duration, 41
  interval, 41
Timeslice, 15
  granularity, 18
Transition matrix
  lossless, 30
  lossy, 32