

# Frequency Scaling as a Security Threat on Multicore Systems

**Conference Paper****Author(s):**

Miedl, Philipp ; He, Xiaoxi; Meyer, Matthias ; Bartolini, Davide B.; Thiele, Lothar

**Publication date:**

2018-11

**Permanent link:**

<https://doi.org/https://doi.org/10.3929/ethz-b-000278307>

**Rights / license:**

[In Copyright - Non-Commercial Use Permitted](#)

**Originally published in:**

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37(11), <https://doi.org/10.1109/TCAD.2018.2857038>

# Frequency Scaling as a Security Threat on Multicore Systems

Philipp Miedl<sup>†</sup>  
miedlp@ethz.ch

Xiaoxi He<sup>†</sup>  
hex@ethz.ch

Matthias Meyer<sup>†</sup>  
matthmey@ethz.ch

Davide Basilio Bartolini\*  
davide.bartolini@oracle.com

Lothar Thiele<sup>†</sup>  
thiele@ethz.ch

## Abstract

Most modern processors use Dynamic Voltage and Frequency Scaling (DVFS) for power management. DVFS allows to optimize power consumption by scaling voltage and frequency depending on performance demand. Previous research has indicated that this frequency scaling might pose a security threat in the form of a covert channel, which could leak sensitive information. However, an analysis able to determine whether DVFS is a serious security issue is still missing. In this paper, we conduct a detailed analysis of the threat potential of a DVFS-based covert channel. We investigate two multicore platforms representative of modern laptops and hand-held devices. Furthermore, we develop a channel model to determine an upper bound to the channel capacity, which is in the order of 1 bit per channel use. Last, we perform an experimental analysis using a novel transceiver implementation. The neural network based receiver yields packet error rates between 1% and 8% at average throughputs of up to 1.83 and 1.20 bits per second for platforms representative of laptops and hand-held devices, respectively. Considering the well-known small message criterion, our results show that a relevant covert channel can be established by exploiting the behaviour of computing systems with DVFS.

## 1. INTRODUCTION

Current mobile computing systems are expected to optimize their power consumption for various reasons, e.g., for prolonging the battery life time or for thermal protection. One of the most commonly used techniques for power optimization is Dynamic Voltage and Frequency Scaling (DVFS). DVFS allows the optimization of the energy consumption by changing the operating frequency and the supply voltage, according to performance needs. Typically, a software component in the operating system, denoted as governor, is responsible for selecting the frequency based on a specific predefined policy, the associated parameters, and run-time information.

Another important design challenge for current computing systems is to provide security and confidentiality of sensitive information, despite the concurrent execution of applications.

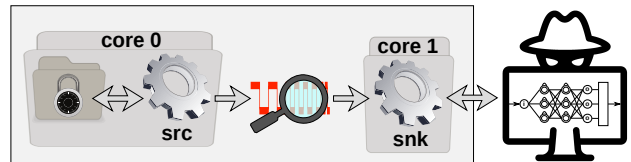


Figure 1: The two applications source (src) and sink (snk) are placed on different cores and are isolated from each other. While src has access to restricted data, snk can use the systems communication interfaces. If the two applications can establish a covert communication channel to transfer data, the restricted data can be leaked to an attacker. This covert channel compromises the security paradigm of permission separation and application isolation.

To ensure security and confidentiality, system designers often apply the security paradigm of permission separation and application isolation. This paradigm dictates that applications have specific sets of permissions to shared resources and memory, and are executed in isolation. The corresponding mechanisms like sandboxing, virtualisation, and fine-grained permission control are typically implemented by the Operating System (OS), possibly supported by hardware. Examples are the process sandboxing implemented in Android or the usage of virtualisation to separate business and private data on a single device (e. g. Virtual Machine). One of the biggest threats for the security paradigm of isolation and permission separation are covert channels, as they are used to compromise the confidentiality of data by allowing secret communication between applications.

In this paper, we investigate the implications of frequency scaling on the security of mobile multicore systems, i.e., the possibility to establish a covert channel between two isolated applications with a sufficient bandwidth and a low error rate to break the security paradigm of permission separation and application isolation. With emerging trends like edge computing, nowadays also many embedded systems, besides consumer laptops and hand-held devices, are equipped with more powerful processors using DVFS. This means, that this security threat has an impact on a large number of current and future devices. We will show, that a DVFS-based covert channel can be established using frequency scaling for optimizing the energy consumption, without the need for elevated privileges by the attacker. This finding makes a DVFS-based covert channel based a threat for a broad range of devices.

<sup>†</sup>ETH Zurich, Computer Engineering and Networks Laboratory (TIK), Gloriastrasse 35, Zurich, Switzerland

\*Oracle Labs Switzerland, Hardstrasse 201, Zurich, Switzerland  
©2018 IEEE

Figure 1 illustrates the example scenario we use to analyse the security threat. Similar to previous research on covert channels [2], we consider two colluding applications trying to establish a covert channel on a multicore system. The first application, the *source application* (`src`) running on *core 0*, has access to restricted data but no access to the communication interfaces. While a covert channel can be used to leak any kind of data, in our example we assume that the restricted data has very high security demands. Such a restricted piece of data can be a cryptographic key, which only needs a channel with a very low bandwidth to be leaked. The second application, called *sink application* (`snk`), runs on *core 1* and has full access to a communication interface but no permitted access to the restricted data available to `src`. The two applications are isolated from each other and, according to this security principle, are not allowed to communicate. We consider a common setting in current multicore architectures; a processor with multiple cores in the same voltage and frequency domain, i. e., the cores share the same voltage and frequency level at any point in time. Further, we assume that the system is idle, hence only the OS and its vital processes are active at attack time. As the system is idle, the behaviour of DVFS will depend on the utilization generated by the source application. If the sink application is able to detect the frequency changes induced by the source application, it can establish a covert channel. We will refer to this channel as *frequency covert channel*, as the key shared resource is the frequency of the cores.

After the frequency trace is transferred from the sink application to the attacker, the restricted information can be extracted. In our scenario, the attacker uses a Neural Network (NN) for signal decoding, to compensate two characteristics of the frequency covert channel. First, the frequency covert channel shows the platform-dependent governor behaviour and second, the current operating frequency of a system not only depends on the utilization, but also on the past operating frequencies. While static decoders would have problems handling these characteristics, NN based decoders with recurrent units allow decoding these frequency sequences.

**Contributions.** Our main contributions in this work are:

1. We are the first to apply a formal communication model to the frequency covert channel.
2. We derive an upper bound on the capacity of the frequency covert channel. A tight capacity bound allows us to estimate the threat potential of a frequency covert channel for a specific hardware-software platform and is an important characteristic for the design of mitigation strategies.
3. We present an implementation of a robust communication scheme, which is based on our communication model and exploits its characteristics. In contrast to previous work, our implementation does not require elevated privileges for system file access and allows the source and the sink application to communicate if they run on different cores. In addition, we show the feasibility of the frequency covert channel under realistic conditions and all of our findings

are supported by extensive experimental validation on two platforms representative of modern laptops and hand-held devices.

4. To the best of our knowledge, we are the first ones to present the usage of NN for signal decoding in a covert channel attack.

## 2. RELATED WORK

Security issues related to privilege separation are well studied and were first discussed in 1973 by Lampson [8]. He defined the *confinement problem* and stated the issue of exploitable *covert channels* and *side channels*. In this paper, we only discuss covert channels; they allow active information transfer between entities that are not supposed to communicate by the security policy.

The US department of defence reported in its 1985 *Orange Book* [16] that trusted computing environments must have “*the capability to audit the use of covert channel mechanisms with bandwidths that may exceed a rate of one (1) bit in ten (10) seconds*”. Furthermore, Moskowitz and Kang [12] defined the *small message criterion* and concluded that bandwidth and capacity alone are insufficient metrics to quantify the threat potential of covert channels. They argue that, if the amount of sensitive information is very small, the capacity of the leakage channel is not an accurate measure to define the threat potential. Consequently, we can state that even a covert channel with very low capacity poses a substantial threat, if the leakage of small amounts of highly sensitive data can compromise the system. Therefore, we will determine the capacity of the frequency covert channel, which exists in almost all computing architectures, and carefully state a possible application scenario to justify the threat potential emphasising the small message criterion.

**Architectural Covert Channels.** Due to the complex architecture of modern processors and resource sharing among different processes and cores, there are many possibilities for the implementation of covert channels. Researchers exploited cache covert channels to transfer data between two virtual environments running on the same hardware platform [19, 18, 15, 9]. A variety of other shared resources in current multicore systems can be used for the implementation of covert channel attacks, for example branch predictors [4], inter process communication [6], process priorities in Android devices [7], or more recently hardware random number generators have also been used to establish high throughput covert channels [3]. Bartolini et al. [2] analysed a thermal covert channel which takes advantage of the temperature measurements in multicore systems and found channel capacity bounds in the order of 300bits per second (bps). Moreover, they showed an experimental implementation that yields data rates in the order of 50bps. Similar to this previous research, we use characteristics of the system architecture to establish a covert channel.

**Covert Timing Channels.** Covert timing channels are based on injecting and monitoring timing variations on certain events. Murdoch [13] presented the possibility to identify servers in the Tor network by exploiting the clock skew in response packets, which is caused by temperature variations of the system. This analysis was further extended to detect whether virtual machines share the same infrastructure or to actively transmit data between two entities with a data rate of 20.5 bits per hour [14, 22, 21]. Yue et al. [20] presented a covert channel where information was encoded into the inter-packet spacing of the network traffic by directly controlling the operating frequency of a device. Similarly, we will establish a covert channel by taking advantage of the timing variation of specific operations. However, we do not control the operating frequency through operations that require elevated privilege levels, nor do we decode information by reading system timestamps. Furthermore, while the above mentioned covert channels are established within a communication network, the covert channel we analyse is established between different applications within the same device.

**Utilization based Covert Channels.** Wang and Lee [17] presented a covert timing channel which allows the communication between two otherwise isolated applications. The source blocks a functional unit for a certain amount of time to cause a delay in computation of the sink. This is interpreted as sending a 1; to send a 0, the source stays idle. Marforio et al. [10] analysed a frequency based covert channel where the source utilizes the processor or stays idle, causing a change of the frequency of the core. The sink application detects frequency changes by reading the core frequency from system files, and can interpret whether a 0 or a 1 is sent. The frequency covert channel we study in this paper is similar to the timing channel presented by Wang and Lee [17] and the frequency covert channel presented by Marforio et al. [10]. All of these channels use the core utilization to encode data. Opposed to the covert channel by Wang and Lee [17], we place the source and the sink application on two different cores allowing cross core communication. Wang and Lee [17] could control the timing changes directly by blocking a functional unit. In contrast, we indirectly control the timing changes by changing the utilization of the core such that the frequency governor scales the operating frequency in a desired manner. As the governor may not behave as desired, our implementation takes advantage of a feedback loop that allows the sender to react to not desired governor behaviour. Marforio et al. [10] uses system readings for the implementation of the frequency covert channel, which can be easily blocked by changing the access permission to these system files. In contrast to that, our implementation uses timing measurements to determine the frequency, as it is harder to mitigate such measurements. Therefore, we can state that our work presents a new covert channel implementation which combines and extends the two previously presented timing and frequency covert channels. Further, unlike Marforio et al. [10], we analyse the characteristics of the frequency covert channel

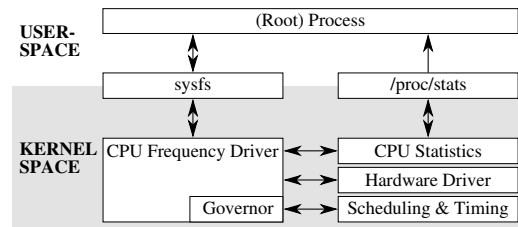


Figure 2: A simplified illustration of the CPU frequency control in the Linux Kernel, depicting which modules interact.

in a detailed way by providing a capacity bound that gives an initial estimate of the threat level of such a channel, and we show a simple and robust implementation of the frequency covert channel.

### 3. FREQUENCY SCALING IN LINUX

In this section we outline the implementation of DVFS in the Linux Kernel, as we base our investigation on this system. Many devices also feature additional modules that can control the operating frequency, as for example CPU throttling through Dynamic Thermal Management (DTM) or device manufacturer specific power and performance optimization or CPU hot-plug techniques. These additional frequency control modules are not analysed in detail in this work. Our threat scenario considers the device to be idle and we can therefore assume CPU throttling caused by DTM will not occur. Other power optimization techniques for commercial devices can be considered (systematic) interference in the channel, as long as there is a change of the frequency depending on the execution pattern, and are therefore not considered in this work.

Linux is used among a diverse range of devices; e. g. server or desktop systems, laptops, powering Android on smartphones or tablets and various embedded systems like the Raspberry Pi. In addition, the open source nature of the Linux Kernel and its components allows us to review the code which handles frequency scaling. Examples of CPU frequency driver implementations in Linux are the *intel\_p\_state* or the *acpi-cpufreq*<sup>1</sup> driver. In this work, we consider the *acpi-cpufreq* driver, which is used in Android systems as well as in Ubuntu and similar OSs. Figure 2 gives a simplified overview of its main components, which we discuss in detail in this section.

#### 3.1. THE CPU FREQUENCY DRIVER

The CPU frequency driver operates as an interface to all the other Kernel components, most importantly the *sysfs* nodes and the hardware driver. One responsibility of the CPU frequency driver is to maintain the *sysfs* nodes used to control frequency scaling from userspace. Furthermore, the CPU frequency driver instructs the hardware driver to set the frequency for each frequency domain, interacts with the scheduling unit

<sup>1</sup><http://www.acpi.info>

and passes information like the utilization statistics from the Kernel to the governor.

### 3.2. THE GOVERNOR

The governor should be called periodically with the timer period  $T_s$ , also called *sampling period*, to determine the new frequency  $f_{set}$  for every frequency domain. Due to differences in hardware and user needs, many customized frequency governors are available for devices based on open source platforms like the Linux Kernel. The multitude of governor options allows the user to set the trade-off between performance and battery lifetime. Different governors can vary in terms of static characteristics like minimum and maximum frequency, but also in their frequency dynamics. In this work, we will focus on the *conservative governor*, which is one of the most commonly used governors in mobile, battery powered systems.

Roughly speaking, the conservative governor uses the average core utilization in the past time interval to determine the new frequency  $f_{set}$ . If the utilization is below or above a certain threshold, the governor reduces or increases the frequency, respectively. Here,  $f_{cur}$  is the current frequency target and  $\Delta f$  is the frequency scaling step. To scale the frequency, the governor uses the relationship between idle time  $t_i$  and the total measurement time  $t_m$ , which is equal to the time elapsed between the last and the current call time of the governor. For simplicity, let us call the term  $t_i/t_m$  *idleness*. The governor adapts the frequency depending on the lower idleness threshold  $I_{low}$  and the higher threshold  $I_{high}$ . There are three scaling cases: (i) If the idleness is lower than  $I_{low}$ , the frequency  $f_{new}$  is increased by  $\Delta f$ . In case the target would be higher than  $f_{max}$ ,  $f_{new}$  is set to  $f_{max}$ . (ii) If the idleness is between  $I_{low}$  and  $I_{high}$ , the frequency is not changed. (iii) If the idleness is higher than  $I_{high}$ , the frequency  $f_{new}$  is decreased by  $\Delta f$ . Whenever the target would be lower than  $f_{min}$ ,  $f_{new}$  is set to  $f_{min}$ .

After calculating  $f_{new}$ , the CPU frequency driver selects  $f_{set}$  from the discrete set of frequency levels that are available on the system. To select the frequencies, the CPU frequency driver applies one of two rules: (A) If the frequency is scaled up,  $f_{new} > f_{cur}$ ,  $f_{set}$  is set to the highest available frequency below or at the target. (B) If the frequency is scaled down,  $f_{new} < f_{cur}$ ,  $f_{set}$  is set to the lowest available frequency at or above the target. After the frequency has been changed to  $f_{set}$ ,  $f_{cur}$  is set to  $f_{new}$ .

## 4. NEURAL NETWORKS AS SIGNAL DECODER

A NN can be employed to make the decoding adaptive, thus less error prone, compared to static decoders. In particular, Connectionist Temporal Classification (CTC) introduced by Graves [5, Chapter 7], can be used for this application, as the decoding can be categorised as time-sequence classification with variable length symbols. CTC is the state-of-the-art technique for dealing with temporal classification tasks. It allows a recurrent NN to make soft labelling decisions at

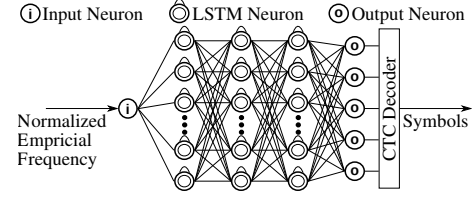


Figure 3: Network architecture example of a signal decoder based on a Neural Network (NN) with Long-Short-Term-Memory (LSTM) neurons using Connectionist Temporal Classification (CTC).

every timestamp and calculate the probability of the correct sequence. By using this probability for the loss function, the NN is able to learn to predict the label sequence without the pre-knowledge of the occurring locations and quantity of the labels. An NN example architecture is illustrated in Figure 3.

The NN itself consists of one input neuron, bidirectional Long-Short-Term-Memory (LSTM) hidden layers, which are improved recurrent NN layers, and one layer fully connected to the last hidden layer. The output layer has to consist of  $Y + 1$  neurons, where  $Y$  is the number of used symbols. One output neuron for each symbol used in the transmission and one additional one for the blank symbol. The blank symbol is used as separator for the other symbols and can be placed by the NN whenever no other symbol seems probable. The output of the output layer is then fed to the so called *CTC-decoder*, which converts this output into the final symbol sequence.

The downside of a NN based decoder is that it needs a lot of training data and high computational overhead for training. However, the training of the network can be done offline and training data can be generated easily as the attacked platform setup can easily be replicated for commercial devices, like the ones analysed in this work. The training of the NN is crucial for the performance of the decoder. If the NN is not designed fitting the complexity of the decoding task, the training is badly parametrised or the training data is of bad quality, the decoder might not work at all.

## 5. THREAT MODEL

We consider the scenario presented in Figure 1, whereas we assume that the source and sink application are already deployed. This could either be done through code injection, or by installation by the user. For example, an attacker could put two separate applications on the application store, which are not suspicious because each on its own does not violate any security restrictions and might be marked coming from a different source.

The sink application  $snk$  measures and records the current frequency. The gathered information can then be forwarded over a communication interface to an attacker device. Further, we assume that the attacked device is idle or only lightly utilized during the attack, e.g., a hand-held device like a smartphone during the night or a laptop powered on in an empty

office during a weekend. Therefore, the source application `src` and the sink application `snk` can wait until the average system utilization is low and will presumably stay low for some time.

The target platforms for this work are mobile devices with shared frequency domains among multiple cores. This architectural feature is present in almost all state-of-the-art processor architectures that are used in computing devices such as mobile phones, tablet computers, laptops or servers. For instance, commonly used big.LITTLE architectures in mobile processors typically feature one frequency domain for all LITTLE and one frequency domain for all big cores, e.g. the Samsung Exynos 5422 Octa. Moreover, in systems with hyper-threading, where multiple logical cores reside on a single physical core, these logical cores share the same frequency domain. The frequency of cores can be inspected with two methods: (a) reading system files, or (b) timing measurements. On Linux, method (a), can easily be blocked by requiring elevated privilege levels to read the system file<sup>2</sup>. The main restriction of method (b) is that it only works if the source and the sink application are placed on two cores within the same frequency domain.

**Timing Measurements Method.** To inspect the frequency via timing measurements with method (b), the sink application `snk` executes a tight loop with a fixed number of instructions and measures the computation time. This measurement is then used to determine the empirical frequency of the core. The sink application `snk` just needs access to a reliable timer in order to determine the time between starting and stopping the measurement load. For instance, the sink application can use the `gettimeofday()` system call standard interface, which is a POSIX-conforming, not requiring elevated privileges. While the timing measurements method (b) does not need any elevated privileges, it comes with two major challenges: (i) its accuracy suffers from interference from other tasks and the measurement load, and (ii) the measurement load increases the total utilization of the cores and it can influence the frequency via the governor, i. e., the measurements indirectly can change the quantity to be determined. In Section 9 we show how to implement the frequency covert channel without the need for elevated privileges in an implementation that overcomes the challenges of the timing measurement method.

## 6. CHANNEL CAPACITY BOUND

The frequency covert channel can be considered value and time-discrete for the following two reasons: (i) it has a discrete number of different frequency levels, and (ii) only updates the operating frequency with a period of  $T_s$ . Using this value and time-discrete channel model, we can calculate the channel capacity using the number of states the channel can take and the state diagram, based on the derivations presented by Miedel

<sup>2</sup>I. e. with the `acpi-cpufreq` drivers

`/sys/devices/system/cpu/cpu$i/cpufreq/scaling_cur_freq`

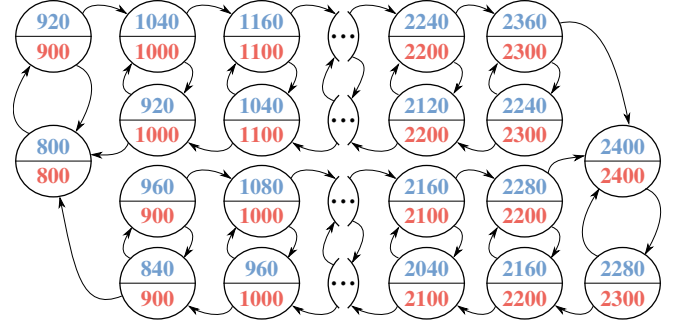


Figure 4: The state diagram of the frequency covert channel for *Laptop* and the conservative governor. Every state is defined by the tuple  $(f_{new}, f_{set})$ , e.g.  $(920, 900)$ .

and Thiele [11]. The number of states in the state diagram  $|S|$  does not only depend on the possible frequency levels of the system, but also on the characteristics of the governor. As  $|S|$  influences the complexity and capacity bound of the channel, let us derive a bound for  $|S|$ . Based on the detailed description of the conservative governor in Section 3, a state can be characterized by the pair  $(f_{new}, f_{set})$  with the target frequency  $f_{new}$  and the actual frequency  $f_{set}$  which is selected by the CPU frequency driver. The value of  $f_{new}$  can change by  $\Delta f$  only and due to the clipping of  $f_{new}$  to  $f_{min}$  and  $f_{max}$  we have at most  $2\lceil(f_{max} - f_{min})/\Delta f\rceil$  different possible values of  $f_{new}$ . As there are two possible rules to determine  $f_{set}$  from  $f_{new}$ , see rules (A) and (B) in Section 3.2, we find  $4\lceil(f_{max} - f_{min})/\Delta f\rceil$  as an upper bound on the total number of states.

The detailed system setup is described later on in Section 7; but for the purpose of the example we just need the governor parameters  $f_{min}$ ,  $f_{max}$ ,  $\Delta f_{ref}$  and the frequency levels from Table 1. Using the parameters of the governor in one of our platforms, *Laptop*, and using the assumption that a symbol represents either an increase in frequency or a decrease, we obtain the state diagram as shown in Figure 4. Although staying at the same frequency is possible, we do not consider it a valid symbol, because of implementation artifacts of the governor presented in Section 8.1. Using this state diagram, we

$$C = \log_2 \lambda_1 \quad (1) \quad \Bigg| \quad B_{max} = \frac{C}{T_s} \quad (2)$$

can derive the connection matrix  $\mathbf{A}$  of size  $|S| \times |S|$ . Now we can calculate the channel capacity bound using Equation (1), where  $\lambda_1$  is the principal eigenvalue of  $\mathbf{A}$ . For *Laptop*, our analysis yields an upper bound on the channel capacity of  $C = 0.972$  bits per channel use. If we apply this scheme to our second platform, *Hand-Held*, we get a capacity of  $C = 0.982$  bits per channel use.

As outlined in Table 1, *Laptop* has a sampling period  $T_s = 80$  ms by default and *Hand-Held*  $T_s = 100$  ms, using Equation (2) this yields a maximum bandwidth  $B_{max} = 12.15$  bps and 9.82 bps, respectively. As we are considering the fre-

quency covert channel to be noiseless, at the same time, this also equals the maximum throughput.

Finally, we note that these calculated upper capacity bounds cannot be achieved in a practical settings, as they are based on idealized conditions. The bounds assume a perfect transmission scheme, no errors due to interferences of other processes, a perfect synchronization between the source and sink applications and no implementation artifacts of the governor. In addition, the bound calculation is based on the assumption that every state transition is observable. We expect a further degradation of the achievable capacity for two main reasons: (i) the sink application can only observe  $f_{set}$  and it is not possible to determine  $f_{new}$ ; (ii) some state transitions cannot be detected. An example for a not observable state transition is the transition from state (1040, 1000) to (920, 1000) on *Laptop*. Therefore, we present a simple implementation of the channel in Section 9, to determine how tight the capacity bound is. The methodology to derive the channel capacity bound presented here can also be applied to other platforms with other governors. The only constraint is that it must be possible to derive the state diagram of the channel.

## 7. SETUP AND INITIAL TESTS

Our experiments are carried out on two diverse hardware platforms representative for two kinds of mobile devices:

1. A Lenovo ThinkPad T440p laptop, based on a 4<sup>th</sup> generation Intel Core i7-4710MQ quad-core processor. It can be clocked at frequencies in the range from 800 MHz to 2.4 GHz in 15 frequency levels, excluding the Intel Turbo Boost;
2. An Odroid-XU3 board, featuring a Samsung Exynos 5422 System-on-Chip (SoC) with an ARM big.LITTLE processor with two quad-core clusters of Cortex-A7 and Cortex-A15 cores. The LITTLE cluster is clocked at frequencies in the range of 200 MHz to 1.4 GHz in 13 frequency levels; the big cluster in a range of 200 MHz to 2.0 GHz in 19 levels.

In the rest of the paper, we refer to platform 1 as *Laptop* and to platform 2 as *Hand-Held*. While *Laptop* is representative for business laptops, *Hand-Held* is representative for hand-held devices (i.e. tablets or smartphones). *Laptop* is running Ubuntu 16.04.1 LTS and *Hand-Held* is operating on Ubuntu 14.04.4 LTS.

To ensure that all of the experiments can easily be reproduced, we defined a strict experimental environment for our two chosen platforms. On both platforms we use the `/dev/cpu_dma_latency` interface of the Linux kernel to set the maximum wakeup latency to  $0\mu s$ . As the deepest allowed sleep state is `POLL` (C0 active), the system cannot go into sleep mode, which could cause changes in the timing behaviour of the governor.

To ensure repeatability of the experiments, we place both devices in an air-conditioned server room with an ambient temperature of  $\approx 23^\circ C$ . Furthermore, we fix the fan speed of

| Param.                    | Value | Param.    | <i>Laptop</i>     | <i>Hand-Held</i> |
|---------------------------|-------|-----------|-------------------|------------------|
| $\Delta f_{rel}$          | 5%    | $T_s$     | 80ms              | 100ms            |
| $I_{low}$                 | 20%   | $f_{min}$ | 0.8GHz            | 0.2GHz           |
| $I_{high}$                | 80%   | $f_{max}$ | 2.4GHz            | 2.0GHz           |
| f-levels in 0.1 GHz steps |       |           | w/o {1.2, 2.0}GHz | all              |

Table 1: Parameters of the conservative governor and the characteristics of the platforms *Laptop* and *Hand-Held*.

```
if (unlikely(wall_time > (2 * sampling_rate) &&
    j_cdbts->prev_load)) {
    load = j_cdbts->prev_load;
    j_cdbts->prev_load = 0;
} else {
    load = 100 * (wall_time - idle_time) / wall_time;
    j_cdbts->prev_load = load;
}
```

Snippet 1: From `cpufreq_governor.c` in kernel 4.4.0. If the time between two consecutive governor calls exceeds twice the sampling period, the current utilization measurement is discarded and the last one is used, if it is not zero.

both platforms *Laptop* and *Hand-Held* to the maximum level<sup>3</sup>. With these measures we minimize any thermal side effects. To minimize scheduling artifacts we run the source and sink application in the `SCHED_FIFO` scheduling class at highest priority on both platforms, using the `pthread_setschedparam()` interface.

Source and sink application are executed on two separate cores that share a frequency domain. The source application will be placed on core 4 (physical core 2) and the sink application on core 0 (physical core 0) of *Laptop*. On *Hand-Held*, we run the two apps on the cores 6 and 7, two big cores; the channel would still work if the two applications were executed on two of the LITTLE cores. The applications are pinned to the respective cores using the `pthread_setaffinity_np()` interface. Finally we note that during all the experiments the systems are only running the source, the sink and the default system services of the OS. We do not alter the standard settings of the governor, presented in Table 1.

### 7.1. PLATFORM-DEPENDENT GOVERNOR BEHAVIOUR

We conduct initial experiments on *Laptop* using the 4.4.0-112-generic Linux Kernel. These experiments revealed that, in practice, the governor behaviour deviates from the ideal behaviour described in Section 3, causing unexpected frequency scalings, that lead to problematic transmission behaviour.

#### 7.1.1. Timing Issues

The first problem arises due to the fact that the governor is not called with a period of  $T_s$ , therefore the total measurement time  $t_m$  can vary (see Section 3.2). According to a communication

<sup>3</sup>*Laptop*: # echo 'level 7' > /proc/acpi/ibm/fan  
*Hand-Held*: # echo 255 > /sys/devices/odroid\_fan.14/pwm\_duty

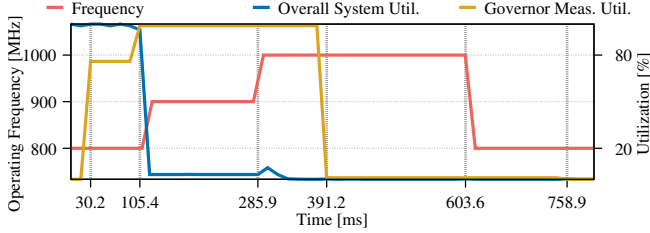


Figure 5: The plot shows the unexpected governor behaviour due to non periodical governor calls, indicated by the vertical dashed lines, and architecture dependencies. At the third call (285.5 ms) the governor assumes that the core was idle and does not update the utilization value, which causes a frequency upscaling while the utilization was below 20%. Furthermore, due to limited visibility of governor states only one frequency scaling can be observed going from 1000 MHz to 800 MHz at the fifth governor call (603.6 ms).

exchange with one of the acpi developers, this can be due to one of two reasons, (i) the core is not active, or (ii) the call to the governor is managed by a *work queue* in kernel versions older than 4.7. After version 4.7, the developers decided to change the implementation and hook the governor calls with the scheduler, to increase the governor call periodicity. Therefore, this behaviour might not be observable using newer versions of the Linux kernel.

As the system should give rescheduled threads a chance to start on a reasonably high frequency after a core has been idle, the developers introduced Snippet 1 into the governor code. This piece of code causes a further deviation from the ideal governor behaviour as described in Section 3.2. The developers assume that the core was idle if the time between two consecutive governor calls ( $wall\_time$ ) is bigger than twice the governors sampling period  $T_s$  ( $sampling\_rate$ ). In this case, the governor discards the current utilization measurement ( $100 * (wall\_time - idle\_time) / wall\_time$ ) and uses the last measurement ( $j\_cdfs \rightarrow prev\_load$ ), if it is not zero. As the replacement of the measured utilization must not happen multiple times in a row, the governor performs a destructive read on the last measurement ( $j\_cdfs \rightarrow prev\_load = 0$ ).

By setting up a debug Kernel, we were able to insert additional debug outputs, to observe the timing behaviour of the governor. As illustrated in Figure 5, we can observe that the governor is not called periodically but with a jitter. The intention of the experiment was to scale up once, and then scale down, which should take only two channel uses. As the governor is called the second time at 105.4 ms and the third time at 285.9 ms, the elapsed time between the two calls is 180.5 ms. This is longer than twice the sampling period  $T_s$ , namely 160 ms, and the governor assumes that the core was idle and does not update the utilization, but keeps assuming an utilization of 80%. As a result, the governor scales the frequency up instead of down, despite the fact that the utilization is below the lower utilization threshold of 20%.

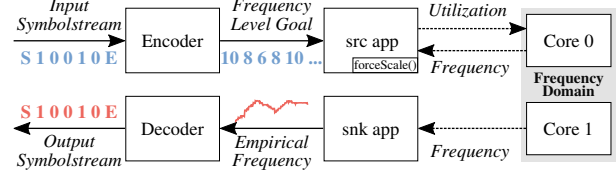


Figure 6: Block diagram of the transceiver system with signal flow indicated by solid arrows and indirect influences of one component on another with dashed arrows.

### 7.1.2. State Transition Issues

The second problem that arises was already mentioned in Section 6: some state transitions are not visible as frequency changes of the core. While this behaviour is exactly as expected in our analysis in Section 3, the calculation of the capacity bound in Section 6 assumes that every state is a valid symbol. As it is not possible to identify every state with the frequency  $f_{set}$  alone and as it is not possible to observe  $f_{new}$ , we cannot use all states to encode symbols. From our example in Figure 5, we can conclude that the governor is in the state (1040, 1000) after the third governor call (285.9 ms), considering the governor state diagram in Figure 4. While the governor is in the state (1040, 1000), we can observe the frequency 1000 MHz and expect that it will decrease to 900 MHz with the next governor call (391.2 ms), as the utilization is lower than 20%. However, with the fourth governor call the governor moves from state (1040, 1000) to (920, 1000), which cannot be observed as the frequency stays at 1000 MHz. At 603.6 ms the governor is called the fifth time and the state changes from (920, 1000) to (800, 800), resulting in a visible state transition due to the frequency scaling from 1000 MHz to 800 MHz.

We can conclude that we can expect a degradation of the bandwidth of the frequency covert channel, mainly caused by three problems: (i) the governor is not called periodically with the sampling period  $T_s$ , but with a certain jitter, (ii) due to the jitter, the governor may use old utilization measurements instead of the actual one (see Snippet 1), and (iii) not all state transitions lead to visible frequency scaling.

## 8. CHANNEL IMPLEMENTATION

Figure 6 illustrates the structure of the frequency covert channel implementation. In this section, we describe the used transmission scheme and the two applications source (src) and sink (snk).

### 8.1. TRANSMISSION SCHEME

Considering the conservative governor functionality outlined in Section 3, its practical behaviour analysed in Section 7.1, as well as the standard parameters outlined in Table 1, we can derive a robust transmission scheme, based on following constraints:

1. As the timing of the frequency scaling is not foreseeable by our applications, a transmission scheme relying on fixed symbol length is not possible. Therefore, staying at the

same frequency level is not considered a valid symbol in our calculations and designs, as this would not allow multiple consecutive symbols of the same value.

2. Due to the functionality of the governor (see Section 3), any symbol can only be based on an incremental change of the frequency level.
3. It is not possible to further decrease the frequency if the core is at  $f_{\min}$ , nor is it possible to increase it if the core is at  $f_{\max}$ . Also, considering our threat model (see Section 5) the attack will only start if the system is idle, therefore we can assume that the core frequency is  $f_{\min}$  at the transmission start. Thus, we have to make sure that the first channel use increases the frequency of the core. In addition, we have to restrict the number of consecutive symbols that would decrease the frequency.
4. We have to take into account the non-ideal governor behaviour shown in Section 7.1, i. e. unexpected frequency scalings during a transmission. As our source application cannot forecast but only detect and react to frequency scalings, we have to tolerate a frequency variation by one level.
5. Due to the characteristics of the frequency covert channel, we know that an error that is not recognized and corrected by the source application will corrupt all following symbols.

In our transmission scheme we allow two kinds of data symbols, 0 and 1. A 0 is transmitted by scaling up 2 levels and then back to the center frequency, a 1 is transmitted by scaling down 2 levels and then back to the center frequency, respectively. To avoid that one non-correctable error corrupts all following symbols, the symbol-stream is divided into packets of fixed length. Each of these packets has a pre- and postamble. The preamble consists of scaling up the frequency to the  $10^{th}$  frequency level and then scaling down by 2 levels to reach the center frequency. Similarly, the postamble consists of scaling up 2 levels and then scale down to the lowest frequency level. By going back to the lowest frequency level we can guarantee that the channel is reset after every packet and ensure that no error is dragged on from one packet to another.

## 8.2. SOURCE APPLICATION

As shown in Figure 6, the main task of the source application is to utilize the core such that the frequency is scaled depending on the input frequency level goal. Moreover, the source application compensates for non observable state transitions, governor miss-scalings and non-periodical execution of the governor (see Section 7.1) by detecting and tracking the current frequency level.

This functionality is mostly implemented in the source application sub-function `forceScale()`, illustrated in Figure 7. The inputs of the function `forceScale()` are the current frequency level goal `flg`, which describes the desired frequency level, and the current measured empirical frequency `curEF`, which is at the same time also an output. `curEF` is first set at the initialization phase of the source application and updated

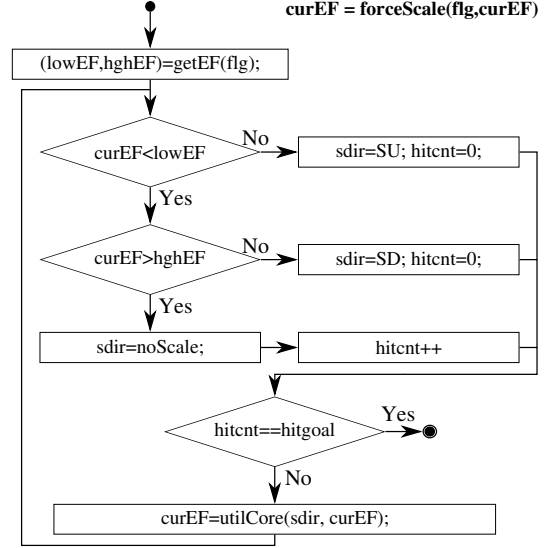


Figure 7: A simplified flowchart of the function `forceScale()` of the source application using pseudocode with C-like notation. The inputs are the current frequency level goal `flg` and the current empirical frequency value `curEF`, with `curEF` also being the output of the function.

every time `forceScale()` is called.

`forceScale()` is called whenever a new symbol is ready to be transmitted. First, the sub-function `getEF` is called to determine the low and high threshold for the empirical frequency. As measurement can vary slightly, the source application uses these thresholds to define an interval within the empirical frequency has to be for a certain frequency level, rather than using fixed values.

Next, the source application checks whether the current empirical frequency measurement `curEF` lies within or outside of the low and high threshold. Here, it can distinguish three cases: (I) If `curEF` is smaller than the lower threshold `lowEF`, the frequency needs to be scaled up (`sdir=SU`). (II) If `curEF` is bigger than the higher threshold `highEF`, the frequency needs to be scaled down (`sdir=SD`). (III) If `curEF` is between the two thresholds, the desired frequency level has been reached and no more scaling is required (`sdir=noScale`). Case (III) is called a *hit* and the counter variable `hitcnt` is incremented, otherwise `hitcnt` is reset. As faulty measurements can occur, a symbol transmission (frequency scaling) is only considered successful if `hitgoal` consecutive hits occur.

If a symbol transmission was successful, `forceScale()` returns. In case the transmission was not completed successfully, the function `utilCore()` is called to force the desired frequency scaling. The inputs to `utilCore()` are the desired scaling direction `sdir` and the current empirical frequency `curEF`, which are used to determine the parameters needed to generate the appropriate core utilization. The function `utilCore()` also implements a so called *backchannel*, which

allows the source application to update the current empirical frequency based on the timing measurements, as described in Section 5. The timing measurements are done using a tight loop similar to the one in the `cpuburn` stress-test<sup>4</sup> and `gettimeofday()`. One timing measurement is performed during the initialization of the application to get a reference value, which is used as divisor for all later timing measurements to determine the empirical frequency.

The main of the source application implements a timeout function, which aborts sending of a packet after a pre-defined time and restarts the transmission process.

### 8.3. SINK APPLICATION

The frequency is indirectly measured by the sink application, with a sampling period  $T$ ; by default  $T = 20$ ms. With the period  $T$ , the sink application inspects the frequency using the same timing measurement method as the source application (also see Section 5). The sink application performs multiple timing measurements which are later averaged when determining the empirical frequency, in order to minimize measurement uncertainty. To increase the accuracy of the empirical frequency measurements, the length of the tight loop used for the timing measurements can be increased. However, a long tight loop also causes more utilization, which leads to a higher channel interference. Therefore, we need to tune the length of the tight loop depending on the attacked device to achieve a good trade-off between accuracy and interference. All time measurements are performed using `gettimeofday()`, which proves precise and lightweight enough for our purposes.

All samples are stored in a preallocated in-memory `log` and dumped at the end of the execution to a log-file. After the experiment has terminated, the log-file can be transferred for off-line analysis. In our implementation, the conversion from the empirical frequency measurement to output symbols is done offline.

## 9. EXPERIMENTAL ANALYSIS

We first analyse a static decoding strategy, based on an off-line evaluation of the platform to determine the empirical frequency to frequency level mapping. An example for a message transmission with a packet length of 5 bits is illustrated in Figure 8. Starting from the top, plot (a) shows the input symbols, where *S* indicates the preamble and *E* the postamble. In plot (b), we can see the goal frequency level input to the source application. According to this frequency goal and input through the backchannel, the source application tries to generate the utilization presented in plot (c). Plot (d) shows the sink application empirical frequency measurements including the measurement artifacts.

Most of the measurement artifacts shown in (d) can be eliminated from the signal in the off-line post-processing. The output signal of the post-processing is illustrated in plot (e). In the

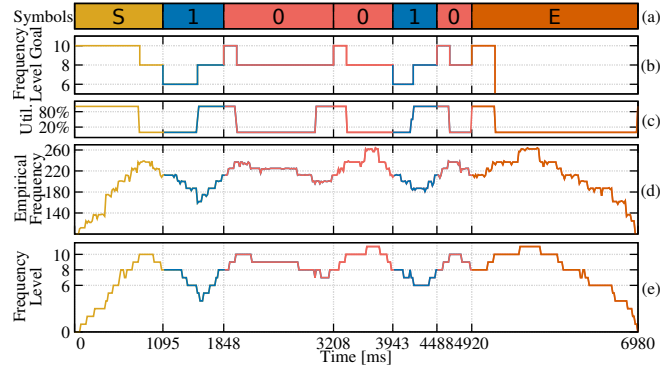


Figure 8: The input symbol stream (a), is converted to the goal frequency level (b). Using this input, the source generates the utilization trace (c). This utilization causes frequency scalings visible in the empirical frequency measurements trace (d). By filtering and discretize the frequency levels trace (e) is obtained, to reconstruct the symbol stream.

off-line post-processing we apply an average filter with a window size of 9 samples and discretize the empirical frequency measurements according to the determined platform specific empirical frequency to frequency level mapping. Although the frequency scaling is not exactly as intended by the source application, the example shows that it is still possible to correctly decode the signal. This is thanks to the source application and coding measures which can compensate most governor implementation artifacts mentioned in Section 7.1. For example, at the end of the second bit (shortly before 3208ms), the frequency level drops below the center frequency. To compensate this drop, the source application increases the utilization again to force the system back to the center frequency.

While the static decoding implementation worked fine for short and few packets, in depth experiments showed its limitations. Using the static decoding scheme, we were not able to reliably reproduce the transmission error rates results using multiple datasets. Packet error rates varied between 20% and up to 60% for repeated transmissions of the same 200 packets with a length of 8 bit. The main reason the static decoding strategy did not work is that the system shows a higher sensibility to interferences than we anticipated. Unlike in our initial tests, the measurement artifacts could not be fully compensated by the offline post-processing and caused more symbol errors. Furthermore, traces were scaled or offset by a random factor, due to system interferences in the initialization process of the source or the sink application. Due to this empirical frequency scaling, the static mapping from empirical frequency to frequency level was incorrect and led to false decoding.

### 9.1. NEURAL NETWORK BASED DECODING

To address the limitations of the static decoding scheme, we employ a NN as a signal decoder that takes a normalized empirical frequency measurement as an input. The LSTM layers contain 72 neurons each using *tanh* as activation function for

<sup>4</sup><https://patrickmn.com/projects/cpuburn/>

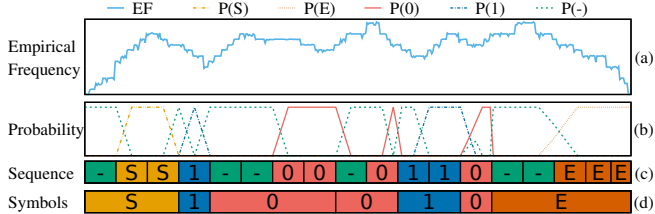


Figure 9: Using the empirical frequency measurements (a) as input, the Neural Network (NN) determines the symbol probabilities (b). The Connectionist Temporal Classification (CTC) decoder uses the probabilities to generate the label sequence (c) and then determine the final output symbol stream (d), equal to the input shown in Figure 8.

the output of the neurons and *sigmoid* for the gates. The output layer consists of 5 neurons using the *softmax* activation function. We need 5 output neurons for the 5 possible labels *S* (preamble), *E* (postamble), 0, 1 and *blank* (-). An example for a packet decoding is illustrated in Figure 9 for the same packet as used in Figure 8. Plot (b) illustrates the probabilities for the different symbols, determined by the NN. The plots (c) and (d) illustrate the two step task the CTC-decoder performs. First, the one-hot encoded label vectors are converted to a label sequence and further to the final output symbol sequence.

For the training, we generated packets with a random length between 8 and 32 bits, summing up to a total of 32000 bits. By using random length packets, we can prevent the NN from just learning the length of the packets instead of correctly classifying the pre- and postamble and do not need to re-train it for each different packet length again. We concatenated the recorded traces such that the resulting training sequences consist of 5000 empirical frequency readings; such a sequence is called sample. In the training, we used 200 of those samples, grouped into 10 mini-batches, where 20 samples (1 mini-batch) is used for validation. The training itself consist of two phases (i) 200 epochs for the initial training and no regulative measures, and (ii) 1000 epochs for the fine tuning, using a clip-norm of 70.0 to prevent gradient explosion. We used a Nesterov accelerated stochastic gradient descent optimizer with a learning rate of 0.001, a momentum of 0.9 and a decay of 0. Furthermore, if the validation loss decreases we save the NN model after each epoch, to make sure we use the best model in the final decoder.

## 9.2. THREAT LEVEL EVALUATION

We evaluate the robustness of our transmission scheme for different payload length of the packets, i. e. 8, 16, 32 and 64 bit. For every packet length we send 5 traces of 200 packets each with random payload bits generated using the python random packet. The payload of each of our traces is big enough to contain multiple 512 bit elliptic-curve cryptography keys [1], see Table 2.

To evaluate the throughput degradation of the frequency

| Packet Payload | Payload Bits per Trace | <i>Laptop</i> BL | <i>Hand-Held</i> BL |
|----------------|------------------------|------------------|---------------------|
| 8 bit          | 1600 bit               | 1.79 bps         | 1.43 bps            |
| 16 bit         | 3200 bit               | 2.27 bps         | 1.82 bps            |
| 32 bit         | 6400 bit               | 2.63 bps         | 2.11 bps            |
| 64 bit         | 12800 bit              | 2.86 bps         | 2.29 bps            |

Table 2: Packet payload, the corresponding number of data bits per trace and the throughput on the baseline platforms.

covert channel, caused by the governor implementation artifacts, we compare the results of our experiments with two theoretical baseline platforms. These baseline platforms have the same parameters as the respective real platform *Laptop* and *Hand-Held* (see Table 1), but we assume that none of the frequency covert channel and the governor implementation artifacts occur (see Section 7.1 and Section 8). We calculate the

$$TP_{ref} = \frac{payload}{T_s \cdot (CU_{PRE} + CU_{BIT} \cdot payload + CU_{POST})} \quad (3)$$

throughput  $TP_{ref}$  of the baseline platforms using Equation (3), in which the *payload* is the number of bits per packet.  $CU_{PRE}$  is the number of channel uses for the preamble and  $CU_{POST}$  for the postamble, which are both 12. The number of channel uses per bit, denoted as  $CU_{BIT}$ , is 4. The respective throughput for each packet length is given in Table 2 for both baseline platforms.

### 9.2.1. Achievable Rates and Error Probability

The upper diagram in Figure 10 shows the achieved throughput in bps, calculated as an average of each single packet throughput for all packets that have been transmitted without error. The packet throughput is calculated by dividing the number of payload bits in a packet by the time needed to send the whole packet, including preamble and postamble. The middle diagram presents the degradation of the throughput between the baseline and the real platforms, i. e. the percentage of throughput loss. The Packet Error Rate (PER) in % is illustrated in the bottom diagram.

Our experimental analysis shows that the achievable throughput is lower than the capacity bound determined in Section 6. The difference between capacity bound and maximum throughput of our baseline platforms can be explained by the transmission scheme, as it needs 4 channel uses per bit in an error free environment without governor implementation artifacts. Therefore we can only achieve a throughput of 0.25 bits per channel use for packets of infinite length, which is significantly lower than the upper channel capacity bound of 0.972 bits per channel use for *Laptop* and 0.982 for *Hand-Held*, respectively. This is necessary due to the platform dependent behaviour and implementation artifacts of the governor (see Section 7.1). The throughput degradation, relative decrease of the throughput on the real platforms *Laptop* and *Hand-Held* to the baseline platforms, are caused by correction

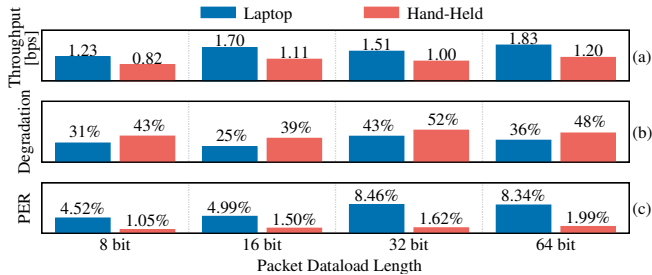


Figure 10: Rising packet length shows increasing trends for throughput (a), throughput degradation in relation to baseline platforms (b) and PER (d). The throughput plunge at 32bit packets indicates high channel disturbance.

scalings the source application has to apply due to unexpected frequency scalings. Moreover, despite the expected increasing trend for the throughput with rising packet length, both platform *Laptop* and *Hand-Held* show a plunge for packets with a length of 32bits. This shows that the platform dependencies and implementation artifacts of the governor cause high disturbance in the channel, independent of the duration of the core utilization. However, we can observe increasing PERs for increasing packet lengths, which correlates with the higher likelihood for bit errors for increasing packet length.

Last, we can observe that the PER on *Hand-Held* is lower approximately by a factor of 4, than on *Laptop*. We chalk the lower PER up to a combination of two reasons: (i) different Kernel versions of *Hand-Held* and *Laptop*, and (ii) the architecture of the processor of *Hand-Held*. *Hand-Held* is operated on Kernel version 3.10.96, while *Laptop* uses 4.4.0-112-generic. As we presented in Section 7.1, different Kernel versions can cause differences in the behaviour of the governor due to implementation artifacts. Furthermore, on *Laptop* all cores share one frequency domain, whereas the LITTLE and the big cores in *Hand-Held* have separate frequency domains. Due to the fact that the *Hand-Held* will try to utilize the LITTLE cores as much as possible and only migrate processes into the big cluster if necessary, we experience less interferences for our transmission. Our analysis suggests that shorter packets show lower PERs but also yield less throughput. However, we can not make a definite statement that is true for every platform, as our analysis also shows platform variations. In order to find the best configuration for a specific platform, further exploration considering the packet length as well as possible error detection/correction codes and protocol overhead has to be done.

### 9.2.2. Interference by other processes

In our scenario, we assume that the attacked device is idle, which is a valid expectation considering the usage pattern of mobile and embedded devices. Nonetheless, other processes could still increase the core utilization and interfere with the channel. While short utilization bursts caused by background processes and idle applications can be handled, a high utilization floor would cause more problems. Short utilization

bursts have the same effect on the governor behaviour as the issues introduced by Snippet 1, namely a single unexpected frequency scaling. We show with our implementation that single unexpected frequency scalings can already be handled with measures like a backchannel, an appropriate transmission scheme (see Section 8) and a smart decoder (see Section 9.1). In contrast, if the interfering utilization is constant and high enough to force the governor to scale to a frequency higher than the lowest possible frequency, the number of reachable frequency levels is reduced. The reduction of frequency levels can only be compensated in the design of the transmission scheme or by a smart source application. In conclusion, we can state that burst interference by other processes can be compensated during an attack, while permanent interference might make an attack impossible. Thus, launching an attack while the device is idle maximizes the chance of success.

### 9.2.3. Threat Classification

In our analysis we consider the small message criterion by Moskowitz and Kang [12] with the following scenario: The highly sensitive data is a cryptographic key which could be used to enter a company network or servers. The cryptographic key is saved on the hand-held device of a company, such that the employee can access the company network and servers while being on a business trip. Today, the National Institute of Standards and Technology considers elliptic-curve cryptography with a key size of 512bit to be highly secure [1]. We assume that an attacker manages to deploy a setup of the slowest implementation of our frequency covert channel on the hand-held device. Furthermore, the attacker uses one parity bit per packet for error detection and a handshake scheme. The handshake scheme involves sending an acknowledge from the sink to the source application after 7 data packets to acknowledge whether the packets were received correctly. In this scheme, every packet transmission is repeated as many times as necessary to transmit it successfully at least once. Considering the throughput of 0.82bps and the packet error rate of 1.05% (see Figure 10), we can state that the application level throughput is 0.56bps. This means, that if the attack is carried out in a night, i. e. the device is not used for 5hours, this 512bit key can be leaked from the source to the sink application 19 times, already accounting for the overhead for error detection, retransmissions and the handshake protocol. Furthermore, as our implementation and transmission scheme are very simple, this attack can be executed more efficiently. For example an attacker could find a more efficient symbol encoding and achieve throughputs closer to the channel capacity bound. Using this reasoning, we can state that the frequency covert channel poses a significant threat for a wide range of devices.

## 10. CONCLUDING REMARKS

In this work we analysed a covert channel based on the frequency of the core, called frequency covert channel. We show

that the main threat posed by the frequency covert channel is the possibility to leak data, which can be used to compromise the widely used OSs security paradigm of permission separation and application isolation. To this end, a source application utilizes the core such that its frequency scalings encode the transmitted data. As a counterpart, a sink application can detect the changes of the frequency by repeatedly measuring the duration of a fixed set of operations. We presented a model of a typical frequency covert channel and derived the corresponding channel capacity bounds which is applicable for every platform-governor combination that allows to derive a channel state diagram. We classified the channel as discrete and noise-free and derived upper channel capacity bounds of about 1 bit per channel use for the conservative governor (see Section 6). The channel capacity bound can be used to compare different kinds of covert channels or help to estimate the associated security risk and develop mitigation strategies.

Based on the channel model, we developed transmission schemes for two distinct and representative platforms, *Laptop* and *Hand-Held*, which we experimentally evaluated. Our findings show that it is possible to achieve throughputs of 1 to 2 bps with packet error rates between 1% to 8%. Furthermore, when considering the achievable performance of the frequency covert channel, we show that there is a high dependency on the used hardware, frequency governor and OS version (see Section 9).

Despite the low capacity of the frequency covert channel, considering that (i) the attacker does not need any special permissions to establish the frequency covert channel, (ii) almost every current mobile multicore system and many embedded systems are affected, (iii) systems can often be compromised by leaking relatively small amount of information, i. e. a cryptographic key or a password, and (iv) these systems are often idle, which makes the execution of the attack easier, we can state that this covert channel needs special attention.

## ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080. This work was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0025. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

Thanks to our colleagues for their help with polishing the paper and to all other reviewers for their valuable feedback.

## REFERENCES

- [1] E. Barker. Recommendation for key management Part 1: General (Revision 4). *NIST special publication*, 800(57):1–147, 2016.
- [2] D. B. Bartolini, P. Miedl, and L. Thiele. On the Capacity of Thermal Covert Channels in Multicores. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16*, pages 24:1–24:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901322. URL <http://doi.acm.org/10.1145/2901318.2901322>.
- [3] D. Evtuyshkin and D. Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 843–857. ACM, 2016.
- [4] D. Evtuyshkin, D. Ponomarev, and N. Abu-Ghazaleh. Covert Channels Through Branch Predictors: A Feasibility Study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, pages 5:1–5:8, 2015.
- [5] A. Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer, 2012.
- [6] T. Heard, D. Johnson, and B. Stackpole. Exploring a high-capacity covert channel on the android operating system. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2015 IEEE 8th International Conference on*, volume 1, pages 393–398. IEEE, 2015.
- [7] J.-F. Lalande and S. Wendzel. Hiding privacy leaks in android applications using low-attention raising covert channels. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 701–710. IEEE, 2013.
- [8] B. W. Lampson. A Note on the Confinement Problem. *Commun. ACM*, 16:613–615, Oct. 1973.
- [9] M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard. Armageddon: Last-level cache attacks on mobile devices. *CoRR abs/1511.04897*, page 169, 2015.
- [10] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.
- [11] P. Miedl and L. Thiele. The Security Risks of Power Measurements in Multicores. In *Proceedings of the 2018 ACM symposium on Applied computing*. ACM, 2018.
- [12] I. S. Moskowitz and M. H. Kang. Covert channels—here to stay? In *Computer Assurance, 1994. COMPASS'94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 235–243. IEEE, 1994.
- [13] S. J. Murdoch. Hot or Not: Revealing Hidden Services by Their Clock Skew. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06*, pages 27–36, 2006.
- [14] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in

- third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.
- [15] H. Rong, H. Wang, J. Liu, X. Zhang, and M. Xian. Windtalker: An efficient and robust protocol of cloud covert channel based on memory deduplication. In *Big Data and Cloud Computing (BDCloud), 2015 IEEE Fifth International Conference on*, pages 68–75. IEEE, 2015.
- [16] U.S. Department of Defense. *DOD Trusted Computer System Evaluation Criteria “The Orange Book” [DOD 5200.28]*. National Computer Security Center, 1985.
- [17] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *null*, pages 473–482. IEEE, 2006.
- [18] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium, Security’12*, pages 9–9, 2012.
- [19] Xu, Yunjing and Bailey, Michael and Jahanian, Farnam and Joshi, Kaustubh and Hiltunen, Matti and Schlichting, Richard. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, CCSW ’11*, pages 29–40, 2011.
- [20] M. Yue, W. H. Robinson, L. Watkins, and C. Corbett. Constructing timing-based covert channels in mobile networks by adjusting cpu frequency. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 2. ACM, 2014.
- [21] S. Zander and S. J. Murdoch. An Improved Clock-skew Measurement Technique for Revealing Hidden Services. In *USENIX Security Symposium*, pages 211–226, 2008.
- [22] S. Zander, P. Branch, and G. Armitage. Capacity of Temperature-Based Covert Channels. *Communications Letters, IEEE*, 15(1):82–84, 2011.