UnCovert3: Covert Channel Attacks on Commercial Multicore Systems

Breaking application isolation using physical system-characteristics

Mirko Selber

Advisor: Philipp Miedl

Professor: Prof. Dr. Lothar Thiele

Department of Information Technology and Electrical Engineering

28.04.2017

Contents

Figures

Abstract

To break the security paradigm of isolation, two appications can use the thermal covert channel. The thermal covert channel exploits temperature variations of the CPU, to establish communication. In this master thesis I evaluate the interference factors of the thermal covert channel exposed by previous work [1], and find a way to compensate for them outside a controlled environment. I then develop a transmission scheme that achieves bitrates as high as 20 bits per second with less than 5% error probability. Last, I manage to leak a RSA private key through the thermal covert channel in a real attack scenario, with an average goodput of 1.358 bits per second.

1 Introduction

Multicore processors are vastly used inside modern devices, from desktop computers to hand-held devices. Processes run in parallel and share resources, allowing the system achieve a higher performance. If applications share physical components, new security concerns regarding processes isolation arise, i.e. it is not fully clear if it is safe to run sensitive applications (e.g. bank, health) and non sensitive-applications on the same silicon piece.

The Operating System (OS) has the responsibility of providing security to applications by using well established techniques. Confidentiality of an application's information can be enforced using permission separation and application isolation, i.e. by using sandboxing. What threatens applications segregation are covert channels, which are communication channels between applications hidden from the OS. If a covert channel is established between two applications, the isolation imposed by the OS is broken and confidentiality cannot be granted.

There are many types of covert channel that exploit various vulnerabilities of the system, such as network protocols [2] or cache timing [3]. Physical covert channels exploit physical characteristics of the device, for example the temperature of the CPU or its frequency. This thesis focuses on the study of thermal covert channel.

1.1 Previous Work

Thermal covert channels are physical covert channels created by observing the temperature of the processor's cores. Thermal variations in a core are caused by the heating generated by running processes, and the cooling. Modern devices employ thermal sensors close to the processing units in order to apply thermal management (e.g. adapt fans' speed). The possibility of accessing the temperature measurements on the system creates a vulnerability that can be exploited to create a thermal covert channel between two applications. One application generates a variation in the cores' temperature, while the second application reads the data from the thermal sensors.

On multicore system, the thermal covert channel has been studied in the work of Masti et al. [4]. The authors show how to establish communication between a sending application (*source*) and a receiving application (*sink*), reaching a transmission rate of 12.5 bits per second (bps). Bartolini et al. [1] expand the research on the multicore thermal covert channel, by finding a capacity bound and proposing a more robust transmission scheme.

In order to determine the thermal channel capacity, Bartolini et al. [1] devised a new methodology, which uses empirical data and information theory to find a capacity bound. The authors show that the thermal covert channel has a maximum capacity of up to 300 bps, and an implementation with a robust communication scheme that achieves rates higher than 45 bps.

The transmission rates reached by Masti et al. [4] and Bartolini et al. [1] in their respective work were achieved under laboratory conditions in which they tested the thermal covert channel. Interfering system characteristics have been mitigated and the testing devices were hold in a thermally controlled server room.

Whether the thermal covert channel can still be considered a threat in a real case scenario has still not been shown. In this master thesis I aim to demonstrate the potential of this threat by leaking sensitive information through the thermal covert channel on two representative of different modern devices in a real attack scenario, i.e. outside of a laboratory setup.

1.2 Threat Model

The threat model used to study the effectiveness of the thermal covert channel in a real scenario is based on the model described by Bartolini et al. [1].

In the proposed attack scenario, depicted in Figure 1-1, two applications are running on different cores on the same CPU and are isolated from one another. The *source* has access to sensitive information, but doesn't have permissions to communicate through the internet or access the thermal sensors. This way, the segregated *source* should not be able to leak the important information even if it has been maliciously infected. The second application in the scenario is the *sink*. Opposed to the *source*, the *sink* doesn't have access to the sensitive data, but can freely communicate over the internet. If both applications manage to successfully establish a covert communication channel, the isolation is broken and the data can be leaked to a malicious third party.

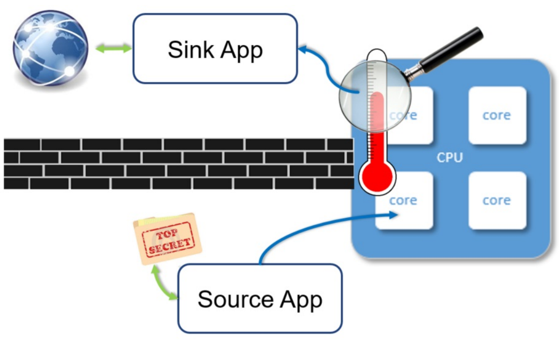

Figure 1-1
Attack scenario exploiting a thermal covert channel

In order to evaluate the threat potential in a real case scenario, some more limitations have to be added to the threat model. None of the two applications can have root access at runtime, but *sink* can use a one-time root access or privilege escalation at installation time. Furthermore, system characteristics that may influence thermal behaviour cannot be modified. Last, the attacked device is placed in a normal working environment (e.g. office) and not a cooled server room or similar.

The final evaluation of the covert channel will be made on its ability to leak a cryptographic key under real conditions. Moskowitz and Kang [5] defined the *small message criterion* and pointed out that the bandwidth and capacity of a covert channel are not a sufficient measure to quantify its threat potential. The scenario and the leaked data are also important factors that needs to be taken into consideration. The quantification of the threat potential can only happen after a holistic analysis of the channel that includes capacity and threat model. So I assume that the ability to leak a cryptographic key undetected under realistic circumstances is a valid criterion to evaluate the threat potential of the thermal covert channel.

2

Influence Factors on the Thermal Covert Channel

The power consumption of a CPU is correlated to its heat generation. By controlling how much power a core is currently using, one can influence the core's temperature. By inducing variations in the power consumption and consequently variations in the temperature of the CPU, the *source* app can transmit data through the thermal covert channel. In order to increase the power consumption, the *source* generates utilization on the system's cores. The transmitted symbols can then be encoded in different utilization pattern and decoded from the resulting temperature variations.

In an ideal scenario, the thermal behaviour can be entirely controlled by the *source*, however in modern systems this is not the case. Power and thermal management, and other system's performance tools change the way the *source* affects the channel. Other processes may also impose CPU utilization creating thermal noise.

In order to shift the experiments from the lab environment to a realistic scenario, all the factors that can influence the thermal behaviour of a system have to be taken into account. Bartolini et al. [1] provide a description of these factors and how they can be controlled in a laboratory environment, in order to prevent strong interferences on the thermal channel.

In this chapter, I will analyse the interference factors and will show their respective mitigation strategy devised, to comply with the threat model as described in section 1.2.

2.1 Thermal Noise

Thermal noise can be caused by two main factors: changes in the device surrounding temperature and other processes generating high load on the CPU. In the work of Bartolini et al. [1] the utilization noise caused by other processes is kept to a minimum, allowing just the core processes of the OS to run. The temperature changes are then almost entirely caused by the *source*.

In a real attack scenario, the same condition of low base utilization can be reached when the system is not actively used, such as a phone charging at night or a laptop left turned on in the office during the weekend. I consider the covert channel attack to happen during those low usage periods. I expect that the remaining background noise will not be able to disturb the thermal channel, meaning that no high external utilization will occur over a long period of time.

In order to exclude thermal noise generated by the *sink*, the rate at which the temperature measures are sampled and the execution of the sampling itself cannot generate a high load on the CPU. This means that the sampling rate of the *sink* needs to be adapted accordingly.

2.2 Fans

Fans and passive thermal radiators are the most used combination to physically cool devices' hardware. The fan speed alters the efficiency of heat disposal. Modern systems usually change the fan speed depending on the current temperature, altering the behaviour of the thermal channel with correlated thermal noise.

To exclude interference from changes in fan speed, Bartolini et al. [1] fixed it to the maximum level during the experiments. When fixing the fan speed is not possible, the transmission scheme has to be adapted to changes in thermal dynamics. Since the changes in thermal dynamics caused by the fan speed is correlated to the actual temperature, the *source* could adapt the transmission scheme depending on the measurement of the thermal sensors. However, in the proposed thread model, the *source* has no access to the temperature readings. This means that the *source* does not have the ability to create a back-channel, i.e. cannot use the output of the channel to improve its input.

2.3 Core Pinning

Modern multicore systems take advantage of their multiple logical cores to increase the system performance. This can result in processes being shifted between logical cores. To exclude these events from happening, Bartolini et al. [1] pinned the *source* and *sink* processes to specific cores. However core pinning is not an option in the proposed threat model, since it changes the normal behaviour of the thermal channel and this is not allowed. Thus the *source* will generate load on different cores.

Since the attack is happening on a not actively used system, only the core on which the *source* is running will have strong temperature variations. Bartolini et al. [1] defined the *allcore channel*, which consists of the thermal channel created by observing the summed temperature of all cores. All the thermal noise generated by other processes are also summed up in the *allcore channel* along with the temperature variations of the transmission. Thus in a system with low thermal noise, the main variations of the *allcore channel* represent the transmitted signal. The *sink* will collect ans sum the temperature measurement of all cores to decode the signal on the *allcore channel*.

2.4 Sleep States

When the system is idle, as in the attack scenario, the CPU can enter so called sleep states to optimize energy consumption. Waking up a physical core takes some time, lowering the immediate performance and introducing delays in the execution of a process.

The system used by Bartolini et al. [1] was not allowed to enter deep sleep states by limiting the *cpu dma latency*. In the threat model presented in section 1.2, this system functionality cannot be modified by our attack because changing the *cpu dma latency* requires root access. Hence the *source* will have the task to wake up the CPU before transmitting in the case all cores entered a sleep state. This can be done by generating an initial utilization on the system, and not allowing longer idle periods during the transmission.

2.5 Frequency Governor

Modern devices try to optimize power consumption through Direct Voltage and Frequency Scaling (DVFS), which changes CPU frequency at runtime in order to decrease the amount of power consumed, while still maintaining high performance. The OS software that regulates the operating frequency is called governor. The governor uses the CPU utilization to decide which frequency to set, depending on its policy. Since different frequency levels produce different amount of heat, in a system with varying frequency the thermal behaviour will have a more complex correlation with the utilization generated by the running processes.

In the systems used by Bartolini et al. [1], the frequency was fixed to exclude interference from the governor's behaviour. In the presented scenario, processes won't have the rights to fix the frequency, so the effects of governors on the system have to be taken into account. Different devices have different default governors on them, which may hinder or enhance the thermal dynamics of the thermal channel. However, frequency governors base their policy for the operating frequency on the CPU utilization, which *source* can influence. This means that I can adapt the transmission scheme to the frequency governor running on the target device.

2.6 Scheduling

Scheduling is used by the OS to assign resources to the different processes. For example schedulers can work with a priority system, where higher priority processes will have precedence over lower priority ones in receiving access to resources. Moreover, a scheduler may force a process to leave resources to another one, in order to prevent starvation or to support critical system processes. Thus the scheduler can cause timing jitters in the channel, which means that there are variations between the expected and the actual execution timing of processes.

To exclude interference from the scheduler, First In First Out (FIFO) scheduling has been used in previous work [1] and *source* and *sink* had highest priority. Using FIFO scheduling, also known as 'first come, first served', any process that uses a resource cannot be interrupted until completion by processes with same or lower priority. FIFO scheduling and maximum priority allows *source* and *sink* to work undisturbed.

In the real case the attacker cannot change scheduling algorithm, which means that the default has to be used. However, in an idle system, timing jitter caused by the scheduling algorithm should be sufficiently small to be corrected by inserting synchronization patterns in the transmission.

3

Transmission Scheme

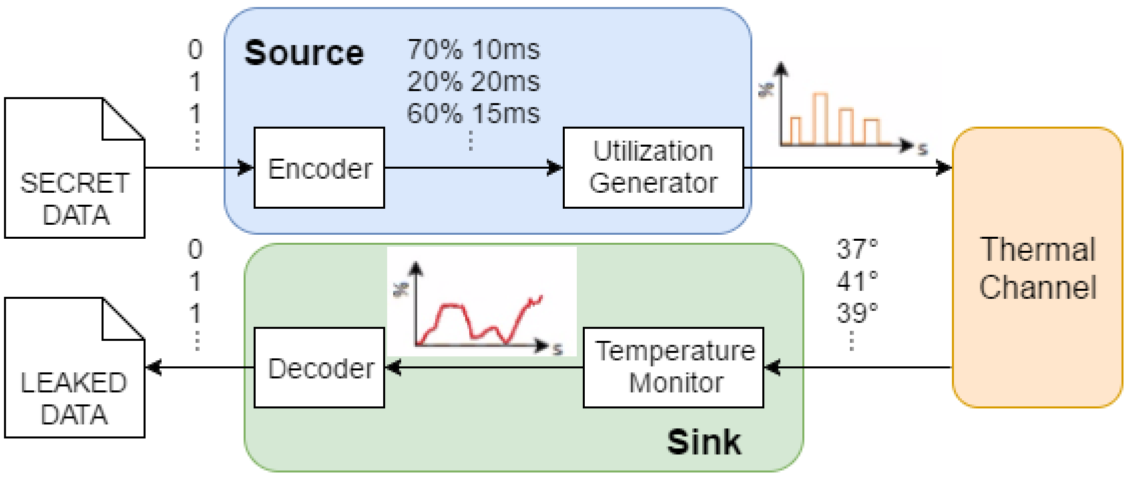

Figure 3-1
Transmission scheme of the communication through the thermal covert channel

The transmission from a sender *source* to a receiving *sink* is depicted in figure 3-1. Initially the message bits are encoded in symbols, which are composed by a utilization in percentage and a duration in seconds. The *source* sends the sequence of symbols through the channel by applying utilization on the core. The heat generated by the fixed utilization patterns caused by the *source* on the CPU produce temperature variations, which are measured by the *sink* through the thermal sensors. By analysing the resulting thermal trace the symbols are extracted and then decoded back into bits.

3.1 Symbol Coding

Taking into account the thermal channel interfering factors showed in chapter 2, the *source* would need a feedback loop, also called back-channel, of thermal measurements in order to be able to reach and maintain a determined temperature.

Since in our threat model, described in section 1.2, the *source* doesn't have access to the readings of the thermal sensors, the symbols cannot be coded using specific temperatures or temperature variations. A line code based on just one type of pulse is more suited for the symbol coding. The choice made by Bartolini et al. [1] is Manchester coding, which they show to work very well for the transmission on the thermal covert channel.

Manchester coding uses a binary code in which the two symbols, 0 and 1, are composed of a high utilization part and a low utilization part, as depicted in figure 3-2(a). A 0 starts with low and then switches to high utilization, while a 1 starts high end ends low. Lets name the high utilization part of a symbol x_{up} and the low part x_{dn}, then a 0 can be written as x_{dn}, x_{up}, while a 1 is x_{up}, x_{dn}. In our transmission scheme, the high utilization is set at the maximum of 100% and the low one at the minimum, which is 0%. This choice has been made to maximize the effect of the encoded symbol on the core's temperature.

The increase in temperature caused when passing from a low to a high utilization period is not instantaneous, there is a channel delay between the applied load and its thermal effect. On the other hand the temperature starts decreasing as soon as the transition from high to low utilization occurs, which shoes that there are different channel dynamics for the rising and falling edge. The steepness of the change in temperature mostly depends on the DVFS governor used. Since higher frequencies drain more power than lower ones, governors that can jump between high and low frequencies quickly, will generate a more steep temperature variation.

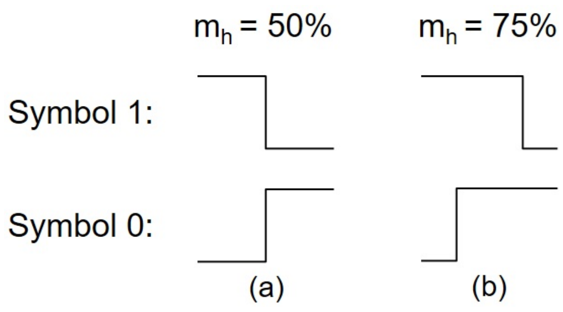

Figure 3-2
Manchester code with different m_h

The effect of the thermal delay on the transmitted 0 or 1 results in an output symbol with an high-to-low ratio smaller than the input symbol. Higher transmission rates suffer more than lower ones from this effect due to the fact that the delay remains of similar length, while the symbol duration is shorter. To achieve a higher rate, the Manchester code can be adapted to the situation.

While in the standard Manchester code half of the symbol is low and the other half is high, a different partition may improve the detectability of the output symbols at higher transmission rates. Instead of using a $50\% - 50\%$ partition of the input symbol, a different high-to-low ratio can be used. By increasing the length of the high utilization part the effect of the thermal delay and the different channel dynamics on rising and falling edge on the output symbol can be compensated. Lets then name the percentage of x_{up} with respect to the whole symbol m_h. In figure 3-2, two different m_h values are presented. The standard Manchester code with $m_h = 50\%$ in (a), and a modified symbol with $m_h = 75\%$ in (b).

3.2 Packet Structure

A message sent by the *source* through the thermal channel will be composed of one or more packets. Each packet is structured as in figure 3-3, starting with a synchronization pulse, in short *sync pulse*, followed by the payload composed of the data bits and error detection coding.

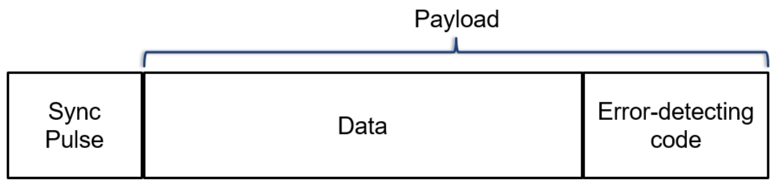

Figure 3-3
Packet structure

The sync pulse is used during the decoding of the thermal trace to localize the packets in order to extract the payload. This initial pulse needs to be distinguishable from the rest of the signal, but the longer it is, the more the effective transmission rate is reduced. We construct the sync pulse by using high utilizations x_{up} and low utilizations x_{dn} of a symbol. Our sync pulse is the sequence $x_{dn}, x_{up}, x_{up}, x_{up}, x_{dn}$. The proposed sync pulse cannot be created by any combination of symbols 0 and 1, thus it's distinguishable from the rest of the signal inside the measured trace. Its duration is exactly two symbol lengths plus a high utilization part.

Another aspect of the transmission that the implementation of sync pulses helps is to correct timing jitters inside packets. Timing jitters occur because *source* and *sink* use different timing sources, which are not synchronized. For example when one of the applications runs inside of a Virtual Machine (VM) and the other on the host OS. The two timing sources used are different and timing jitter on the VM process causes synchronization issues between *source* and *sink*.

To correct the timing of a packet, two sync pulses of neighbouring packets are used. By locating the falling edge of the two pulses, the timing of the packet between them can be stretched or compressed to match the expected timing. This procedure can be repeated for all packets except the last one, because no other sync pulse follows the end of the message. The end of the last packet is calculated by taking into account the expected length and the timing jitter of the preceding packet. In order to choose a good packet length the maximum timing jitter has to be taken into account with respect to the symbol length. For example, with a maximum timing jitter of 0.5% per symbol, the 40^{th} symbol can have a maximum of 20% jitter with respect to the expected timing.

Payload symbols are divided between data and error detection. Since the transmission is not error free, given the interfering factors in chapter 2, the packets need error detection to allow the receiving part to discard faulty packets. We use Cyclic Redundancy Check (CRC) error-detecting code inside the packets sent in our experiments. The CRC coding uses a fixed polynomial to calculate the remainder of the polynomial division on the packet's data. The remainder is then attached at the end of the packet as a check value, which is recomputed and compared after the transmission. CRC is good at detecting random errors generated by noise in transmission channels and easy to implement.

4

Experimental Setup

The thermal covert channel attack is performed on two platforms representative of modern commercial devices, a business laptop and a smartphone.

The business laptop is a Lenovo ThinkPad T460s, with an Intel Core i7-6600U CPU that has two physical each running two threads resulting in four logical cores. The OS on the device is Ubuntu 15.10, and to grant isolation between the two applications, Orcale VM VirtualBox is used. The used VM runs the same OS (Ubuntu 15.10), has two cores and the CPU execution cap is at 100%. Other notable specifications of the laptop that influence the thermal covert channels are the default frequency governor, which is *intel p-state* powersave, and the default scheduler called Completely Fair Scheduler (CFS).

A Samsung Galaxy S5 with an Exynos 5422 Octa chipset is used as for smartphones. It's running Android 5.0 Lollipop and uses Cortex A15 and A7 in a big.Little composition. The application sandboxing enforced by Android is already sufficient to grant the isolation between *source* and *sink*.

Apart from the presented laptop and a smartphone, other two devices are also used in the experimental setup: a computer that runs the data processing framework and a server for storage purposes. Data processing is done with Matlab while the necessary scripts to control the framework are written in bash. The *source* and *sink* applications are written in C++ for the laptop, and most of the code has been encapsulated into Java using the Android Native Development Kit (NDK) for the smartphone. The frequency governor of the phone is the default interactive governor, which has a quick reaction to utilization changes on the CPU.

4.1 Source

The core of the *source* works as follows. The symbols are fed to the main loop in the form of a list of instructions. An instruction contains a mode, 1 or 0, and the duration in microseconds of the mode. If the mode is 1, 100% utilization is generated on the CPU for the necessary duration, using a tight loop similar to *cpuburn*. If the instruction mode is a 0 instead, the process goes to sleep for the requested amount of time. The timing is calculated using Time Stamp Counter (TSC) on the laptop and the *clock_gettime* function on the smartphone. After the completion of the set of instruction the loop is finished.

In a first experimental version used for the empirical evaluations in section 5.1.1, the instructions are given to the *source* through an external file the application reads. The application begins the transmission as soon as an external scripts, which handles the experiment execution.

In the final version use for the real attack in section 5.1.2, the *source* directly encodes the bits of a target file and creates the list of instructions internally. Moreover, the *source* will start the transmission on the thermal covert channel at previously hardcoded rendezvous points using the local time of the system.

4.2 Sink

The application reads the temperature from *"/dev/cpu/*[CPU#]*/msr"* for the laptop, and *"/sys/devices/10060000.tmu/curr_temp"* in the smartphone. While for the Android device no extra permission has to be granted to access the thermal data, reading privileges to the *msr* file are not granted by default on the laptop. In order to allow the *sink* access to the temperature reading without root permissions at runtime, three actions need to be executed with highest privileges during installation. Firstly we make sure that the *msr* module is loaded with *"modprobe msr"*. Afterwards the *msr* file needs to be made readable and writeable by all, and then the command *"setcap cap_sys_rawio=ep <executable>"* has to be run on the *sink* binary.

Timing within the application is handled as in the *source*. When the measurement part is over, the remaining time until the next sampling instant is calculated and the process is put to sleep for the remaining duration. The *sink* samples the temperature each five milliseconds in our experimental setup. Higher sampling rates may improve the decoding process, but it is limited by the measurement rate of the thermal sensors.

As explained in the previous section, the transmission on the covert channel is performed at rendezvous times. The transmission rate and the data size are fixed, since the target file for the attack is chosen beforehand and its path is already hardcoded inside the *source*. This way the *sink* knows approximately how long it has to measure the temperature to ensure recording the whole message.

4.3 Data Processing

The output traces of the *sink* are fetched by the device responsible for the data processing after an arbitrary amount of time, e.g. one night or weekend. On the device the data is fed to the data processing framework, which analyses the temperature traces and decodes the signal.

Finding the beginning of the transmission inside the thermal trace is the first problem to be solved during the data processing. Before starting the transmission the *source* waits one second before sending the message. This prevents the *source* to start sending the transmission before the *sink* has started recording temperature values. The one second margin results in a quiet channel before the first sync pulse begins, which is easily identifiable on the thermal trace.

After the beginning of the signal has been identified, the message is split into packets by searching for the sync pulses one at a time. To correct timing jitters, the method described in 3.2 is used. The end of the signal is reached once the temperature variations are below a certain threshold.

The symbols of each packet are extracted and then multiplied with two perpendicular carriers. The resulting symbol space is analysed with decision device, which decodes the symbols into bits. The decision device needs a training round, which is executed during the first evaluation phase in section 5.1.1. An attacker can use a device equal to the target device for the training round of the decoder.

Once the bits are decoded the CRC remainder of every packet is calculated and compared with the error-detecting bits at the end of every packet. Faulty packets are discarded and error free packets are kept, and, since we are decoding bits of an ASCII file as described in section 5.1.2, the characters are extracted from the bits. Only characters composed of bits coming from good packets are considered for the final decoding of the message.

The attack is considered successful as soon as every packet has been transmitted without error at least once. This also includes the packets which had both data and error-correcting symbols wrongly decoded, but the resulting packet CRC remainder indicates a good reception. To counter this problem each packet has to be correctly decoded at least twice, and if the two resulting bit sequences are different a third correct decoding of the packet is needed to exclude the faulty one.

5

Experimental evaluation

To perform the thermal covert channel attack proposed in this thesis we first need to empirically evaluate the communication channel at our disposal, and devise a good set of parameters for the transmission scheme. Thus the first experiments will be performed with the first more easily controllable version of *source* and *sink*. Experiment execution is managed by external scripts, which take care of the synchronization between the two application via signalling.

The final transmission parameters are then chosen from the empirical results and hardcoded in the final version of the two applications. The real attack is then performed by transmitting a private RSA cryptographic key generated with *ssh-keygen -t rsa*, which has a length of 1679 bytes.

5.1 Laptop

The parameters needed for the laptop transmission scheme are packet length, amount of error-detecting bits, m_h value and transmission rate. Once the parameters have been chosen and implemented, the attack through the thermal covert channel will be executed periodically, at a rate depending on the length of one transmission.

5.1.1 Empirical Evaluation

The packet length depends on the amount of timing jitter induced by the VM on the resulting trace, and the amount of error-detecting bits needed depends on the packet length. The first test runs on the laptop showed a maximum timing jitter around 2.3% jitter per symbol. I take some security margin on this empirical value and use 2.5% jitter per symbol to calculate a good packet size. I allow a maximum accumulated timing jitter of 50%, which is reached after 20 symbols. I thus choose to have packets with a payload of 20 bits, 16 bits carry the information, while 4 bits are used for error detection.

The bits transmitted by the *source* are encoded using Manchester coding with different m_h percentages as explained in section 3.1. Thus I evaluate the different

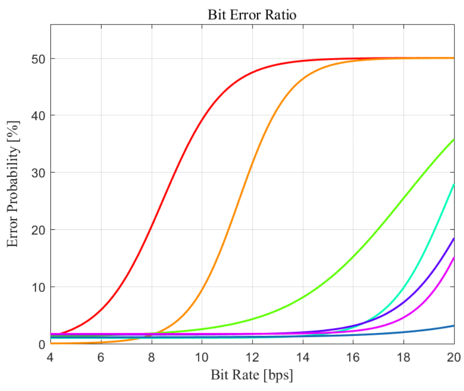

Figure 5-1
Laptop, BER of different m_h values over transmission rates

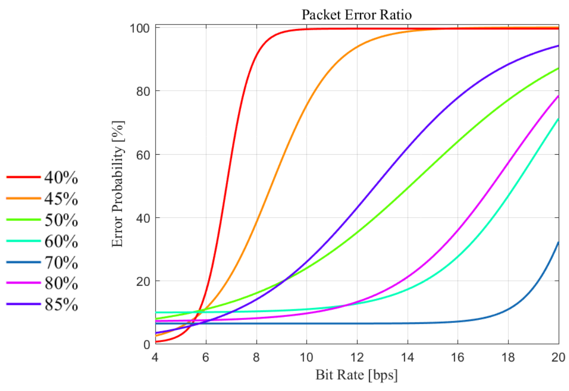

Figure 5-2
Laptop, PER of different m_h values over transmission rates

transmission schemes for symbols with m_h from 35% to 85%, each 5% step, at transmission rates from 4 to 20 bps. Each run is composed of a training and an evaluation bit sequence. The training sequence, which is then used to train the decision device, contains $1'000$ randomly generated bits, while the evaluation part contains $10'000$ random bits.

The results are presented in the figure 5-1 and 5-2, which show BER and PER of some of the evaluated transmission schemes. To better show the results, only part of the evaluated transmissions has been plotted. The curves of both figures are the trend lines of the resulting data fitted with a sigmoid function.

In figure 5-1 the trends in bit error probability of transmissions with different m_h values are plotted against the transmission rate. As is can be seen in the plot, transmissions with small m_h with respect to the standard Manchester code, which has $m_h = 50\%$, rapidly reach a high BER with increasing bit rate. Because of the delay between the utilization rise and the thermal effect, at high rates most of the symbols are not able to generate the necessary thermal variation.

A consistent improvement in performance can be achieved when using a higher m_h. With respect to the $m_h = 50\%$ case, all higher m_h encoding show a more robust behaviour at faster transmission rates. However, there is a limit to how much we can increase the m_h percentage. The optimum m_h in our case is $m_h = 70\%$, values above and below it show a worse performance. This makes 70% a good m_h choice for the real attack. Even at the highest rate of 20 bps, the $m_h = 70\%$ trend line reaches less than 5% BER.

Figure 5-2 shows the PER of the same set of data, with respect to the transmission rates. Since we are interested in decoding good packets, the PER gives a better understanding of the performance of a specific combination of m_h value and transmission rate during a real attack.

The plot shows that the PER of the traces with high m_h values, i.e. 80% and 85%, suffers a greater decrease in performance at higher rates compared to their BER trend. In particular the $m_h = 85\%$ trace at high rates has a worse PER, with respecto to $m_h = 50\%$, but half its BER. The distribution of faulty bits inside the message is worse at high m_h values, since it generates a higher PER.

fewer close to the performance of the standard manchester code with $m_h = 50\%$. In the BER plot however, the performance of the first trace was higher than the second one, meaning that higher m_h values may lead to an error distribution which causes high PER.

For the real attack we need a transmission scheme with a good trade-off between PER and bit rate. We choose the combination of $m_h = 70\%$ and a transmission rate of 17 bps. As it can be seen from figure 5-2 this combination leads to a packet loss of less than 10%, while maintaining a high performance. We then hardcode the final transmission scheme in the attack version of the *source*.

Before executing the attack through the thermal covert channel, we investigated the effect of a different frequency governor on the transmission. In the previous work, Bartolini et al. [1] found in their robustness study, that the actions of the *acpi-cpufreq* conservative governor degenerated the performance of the transmission in a critical way. We had the same problem when we applied the transmission scheme we developed for the *intel p-states* powersave governor, the temperature changes caused by utilization pulses were sometimes normal and other times too small to be decoded. After an investigation on the conservative governor behaviour, we managed to recover most of the lost performance, by inserting a long utilization period at the beginning of the transmission. This forces the frequency governor to increase the frequency to its maximum level, and by ensuring short idle period during the transmission itself the frequency stays high, enhancing the thermal dynamics of the channel. We intended to continue investigating the possibility to also exploit the thermal channel of the conservative governor, but we didn't have enough time to continue.

5.1.2 Realistic Attack

During the final attack with the powersave frequency governor setup, we intend to leak the *id_rsa* private SSH key file over the thermal covert channel. The file contains $13'432$ bits of ASCII characters that are spread over 840 packets, the last one being zero padded. Each packet also contains 4 bits of error-detecting code and a sync pulse, which is 2.7 bits long. The total length of the message is $19'068$ bits, which will take approximately 18 minutes and 42 seconds for a complete transmission. The communication will happen at rendezvous points between *source* and *sink* every 30 minutes $.00$ and $.30$ hour mark. Since our target devices are always idle, we don't need to differentiate between busy and idle periods to send the message.

The resulting average PER of the attack is around 21.942%, an I managed to successfully decode the message after 8.818 complete transmissions on a total of 100 transmissions. The resulting average goodput is 1.358 bps, which was calculated by taking the total amount of data bits ($13'432$) and dividing it by the netto transmission time ($8.818 \cdot 18'42''$).

I also discovered that activating the Graphical User Interface (GUI) of the VM degenerates the transmission on the thermal channel. Since we are using the VM as a mean to enforce the isolation between *source* and *sink*, and not because of the threat model, the resulting average goodput we obtained without a GUI on the VM is still correct.

5.2 Smartphone

Since the smartphone thermal channel does not present timing jitter, more than one synchronization pulse is not necessary during the empirical evaluation part, thus the message is sent using one single packet.

The results from the phone are not nearly as good as those on the laptop. We did not grasp the thermal dynamics of its channel yet, and we cannot pursue the research on the smartphone due to timing limitations.

In figure 5-3 we show an extract with 20 bits of a transmission of 100 total bits on the thermal channel. The transmission parameters used are $m_h = 70\%$ at a bit rate of 1 bps. The figure shows the temperature variation that this communication has managed to induce on the system.

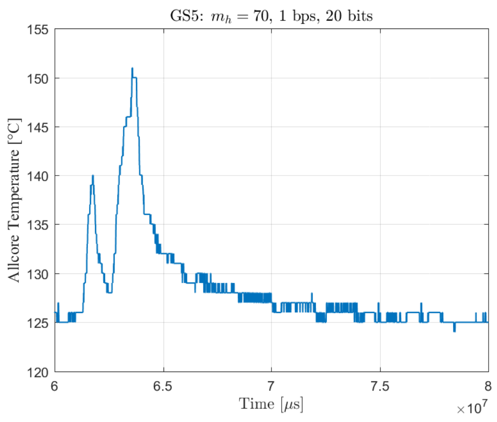

Figure 5-3
Smartphone, extract of output trace showing allcore temperature

It is intuitive that the presented trace cannot be decoded to obtain the 20 bits it should contain. After the first two temperature pulses the temperature drops to its low level, while following utilization pulses do not manage to generate enough heat to be recorded. The data comes from the four thermal sensors placed on the eight cores of the processor. As stated before due to the project time we couldn't further investigate the cause of this behaviour.

6 Concluding Remarks

In this thesis we analysed the previous work done by Bartolini et al. [1] and found its main weaknesses, which are the limited robustness study they performed and the fact that they didn't perform an attack in a real case scenario. We then compensated the interference factors exposed by Bartolini et al. [1] in order to develop a robust transmission scheme for a real attack.

The transmission scheme we developed reached bitrates as high as 20 bps with less than 5% BER. This result has been reached mainly thanks to the modified Manchester coding we applied when encoding the symbols for the transmission. This result can still be improved, for example by optimizing the transmitted symbols by exploiting the state of the channel after the previous one. If the previous symbol ended with high utilization and the current one starts with high utilization, i.e. 01 symbol sequence, the temperature is already high. The utilization of the current symbol could be improved to increase the distance between regions in the symbol space.

I then successfully performed an attack in a real case scenario exploiting the thermal covert channel, successfully leaking the $13'432$ bits of a *id_rsa* file containing a private SSH key. The key was correctly decoded after an average of six transmissions of all packets, which has a resulting average goodput of 1.358 bps.

In its *Orange Book* [6], the US department of defence (DOD) stated that trusted computing environments should have the capability to audit covert channels with bandwidths of more than 0.1 bps. The DOD also reported in the same book that in most application environment bandwidth of at maximum 1 bps are acceptable. The average goodput of 1.358 bps seems indeed to be low, but it has to be taken into consideration that the goodput is already the throughput at application level. Moreover, as stated in section 1.2, Moskowitz and Kang [5] argued that bandwidth and capacity alone are not a sufficient measure of the threat potential of a covert channel.

I thus showed that an attack that uses the thermal covert channel is indeed possible even in a real case scenario.

A

Appendix: Data Storage Location

Here are the file paths for the data used. On the storage server the folder containing all the files is: */home/thermal/experimental_results/thermal-cc/*

Empirical evaluation on laptop. Different m_h values test:

- Pallando_20pk_35pw_LaptopVM_2017-03-16_1705.tar.gz

  Pallando_20pk_40pw_LaptopVM_2017-03-16_2320.tar.gz
- Pallando_20pk_45pw_LaptopVM_2017-03-17_0535.tar.gz
- Pallando_20pk_50pw_LaptopVM_2017-03-17_1152.tar.gz
- Pallando_20pk_55pw_LaptopVM_2017-03-17_1811.tar.gz
- Pallando_20pk_60pw_LaptopVM_2017-03-18_0031.tar.gz
- Pallando_20pk_65pw_LaptopVM_2017-03-18_0650.tar.gz
- Pallando_20pk_70pw_LaptopVM_2017-03-18_1309.tar.gz
- Pallando_20pk_75pw_LaptopVM_2017-03-18_1930.tar.gz
- Pallando_20pk_80pw_LaptopVM_2017-03-19_0151.tar.gz
- Pallando_20pk_85pw_LaptopVM_2017-03-20_1947.tar.gz

Empirical evaluation on laptop. Conservative governor test:

- Pallando_20pk_70pw_con_LaptopVM_2017-04-14_1906.tar.gz

  Pallando_20pk_70pw_con_LaptopVM_2017-04-20_1744.tar.gz

Real attack on laptop:

- Pallando_real_attack_final_2017-04-30.tar.gz

Empirical evaluation on phone:

- GS5_70pw_GS5_2017-04-23_1013.tar.gz

Bibliography

[1] Davide B. Bartolini, Philipp Miedl, and Lothar 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`.

[2] S. N. Omar, I. Ahmedy, and M. A. Ngadi. Indirect dns covert channel based on name reference for minima length distribution. In *ICIMU 2011 : Proceedings of the 5th international Conference on Information Technology Multimedia*, pages 1–6, Nov 2011. doi: 10.1109/ICIMU.2011.6122768.

[3] 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 *2015 IEEE Fifth International Conference on Big Data and Cloud Computing*, pages 68–75, Aug 2015. doi: 10.1109/BDCloud.2015.12.

[4] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 865–880, Washington, D.C., 2015. USENIX Association. ISBN 978-1-931971-232. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti`.

[5] 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, Jun 1994. doi: 10.1109/CMPASS.1994.318449.

[6] *DOD Trusted Computer System Evaluation Criteria "The Orange Book"*. •, 1985.