On the Capacity of Thermal Covert Channels in Multicores

Davide B. Bartolini[†]
davideb@ethz.ch

Philipp Miedl[†]
miedlp@ethz.ch

Lothar Thiele[†]
thiele@tik.ee.ethz.ch

Abstract

Modern multicore processors feature easily accessible temperature sensors that provide useful information for dynamic thermal management. These sensors were recently shown to be a potential security threat, since otherwise isolated applications can exploit them to establish a thermal covert channel *and leak restricted information. Previous research showed experiments that document the feasibility of (low-rate) communication over this channel, but did not further analyze its fundamental characteristics. For this reason, the important questions of quantifying the channel capacity and achievable rates remain unanswered.*

To address these questions, we devise and exploit a new methodology that leverages both theoretical results from information theory and experimental data to study these thermal covert channels on modern multicores. We use spectral techniques to analyze data from two representative platforms and estimate the capacity of the channels from a source application to temperature sensors on the same or different cores. We estimate the capacity to be in the order of 300 *bits per second (bps) for the same-core channel, i.e., when reading the temperature on the same core where the source application runs, and in the order of* 50 *bps for the 1-hop channel, i.e., when reading the temperature of the core physically next to the one where the source application runs. Moreover, we show a communication scheme that achieves rates of more than* 45 *bps on the same-core channel and more than* 5 *bps on the 1-hop channel, with less than* 1% *error probability. The highest rate shown in previous work was* 1.33 *bps on the 1-hop channel with* 11% *error probability.*

1. Introduction

After the breakdown of Dennard Scaling [8], power density grows with increasing integration in CMOS technology. Due to this effect, switching too many transistors at a time generates more heat than can be dissipated, possibly damaging the chip due to exceeding the maximum safe temperature. While hardware driven Dynamic Thermal Management (DTM) [4] can avoid damages and ensure integrity, it resorts to techniques that severely impair performance, such as sharp speed throttling. For this reason, most modern multicores expose a software interface to the temperature sensors, in order to enable smarter thermal management policies that gracefully impact performance and avoid triggering hardware DTM. For example, Intel Core processors expose one sensor per core; similarly, the ARM big.LITTE SoC exposes one sensor per big core. These sensors are easily accessible on laptops or desktops running Windows or Linux through simple tools that export temperature information to userspace processes. Additionally, we verified that user-installed apps can access temperature sensors on Android-based smartphones and tablets without requiring any specific permissions.

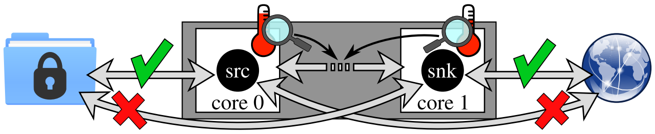

Figure 1: The source app (src) has access to restricted data but no network access; the sink app (snk) has no access to the restricted data but has network access. A compromised source app can leak sensitive data to the sink app through the thermal covert channel, breaking privilege separation.

Temperature sensors are a valuable asset for thermal management, but they can represent a security breach in *privilege-separated*, or *sandboxed*, systems. A widespread example of such systems are Android-based smartphones, where each app has access to data and resources based on user-granted system permissions. Another example is sandboxing in modern browsers, where each tab runs in an isolated process with restricted permissions [25].

Recent research [21] provides evidence that temperature sensors can be used to implement a *covert channel* [18] that allows otherwise isolated applications to communicate and possibly leak sensitive data. For instance, consider the dual-core system depicted in Figure 1. A source (src) app runs on core 0 and has access to sensitive data that is only stored locally, but it does not have network access. A sink (snk) app runs on core 1 and can freely communicate over the network, but has no rights to access the sensitive data. In theory, privilege-separation should disallow communication between the two applications and keep the sensitive data secured, even in presence of a compromised source app and a malicious sink app. However, if the sink app can read the on-chip temperature sensors, communication is possible through the thermal covert channel, regardless of privilege-separation.

If the system load is low, the source app can exploit the *sleep-states* [2], used in modern multicores to save energy and increase battery life, to predictably influence the temperature of its core and, due to heat transfer, the temperature of the nearby cores. When the source app is active, its core wakes up and dissipates heat, thus raising the temperature; when the

[†]ETH Zurich, Computer Engineering and Networks Laboratory (TIK), Gloriastrasse 35, Zurich, Switzerland

source app is idle, its core goes back to sleep and the temperature drops. At low load, the other cores are mostly in sleep-mode and do not introduce much noise. The source app exploits this effect to encode a message into its execution trace; the sink app can retrieve the message by decoding the temperature trace it reads from the on-chip sensors. In Section 3, we specify in more detail our threat model, while Section 4 illustrates how we model this covert communication channel.

Previous work [21] presents an empirical study of the *1-hop channel*, i.e., when the sink app can read the temperature of the core physically next to the one where the source app runs. This study shows experiments that achieve a throughput of up to 1.33 bits per second (bps) with an error rate of 11% on an Intel Xeon-based server. This result demonstrates the feasibility of communication on the 1-hop channel at low rates, but finding the actual channel capacity and the achievable rates, and evaluating different platforms remain challenging open questions. We need to answer these questions in order to understand the possible entity of this threat in current systems.

Contributions. In this paper, we present and exploit a new methodology that mixes theoretical and experimental analysis to tackle two main challenges:

1. Estimating the capacity (under controlled but realistic conditions) of the thermal covert channel; and
2. Finding a communication scheme that improves previous throughput results towards the channel capacity.

Both for estimating the channel capacity and for evaluating the throughput of our communication scheme, we use experimental data collected from two diverse mobile multicores representative of laptops and smartphones, compared to the single server platform studied in previous work. Section 5 illustrates our experimental setup. We estimate that the capacity can be in the order of 50 bps for the 1-hop channel and in the order of 300 bps for the *same-core channel*, i.e., when the sink app can read the temperature of the core where the source app runs (Section 6). Moreover, we show a communication scheme that achieves rates of more than 5 bps on the 1-hop channel and more than 45 bps on the same-core channel, with less than 1% error probability (Section 7). This result is much higher than the maximum rate of 1.33 bps on the 1-hop channel with 11% error probability achieved in previous work [21] with a naïve communication scheme.

2. Background and Related Work

Studying the security issues related with privilege-separation and isolation in computing systems is a well-defined area of research. Back in 1973, Lampson [18] analyzed this *confinement problem* and noted the possibility of exploiting *covert channels*, i.e., observing system properties not originally intended for communication, in order to leak restricted data.

The term *covert channel* is used when the source and the sink app actively share information, as opposed to the term *side channel*, used when an attacker observes an unaware system with the aim of inferring sensitive information, e.g., a cryptographic key [15]. While temperature measurements could be used as a side channel [21], in this paper we focus on their use as a covert channel, as Figure 1 illustrates.

Covert channels can broadly be classified as *storage* or *timing* channels. In storage channels, the source app directly or indirectly writes to a shared resource, which the sink app reads; in timing channels, the source app exploits the ability to influence timing properties of the system that the sink app can observe [22, 32]. The covert channel that we study in this paper is a storage channel: the source app affects the temperature that the sink app can observe.

2.1. Microarchitectural Channels

Complex processor architectures are likely to expose properties that can be exploited to create covert or side channels [34] to leak information across security domains; in particular, shared microarchitectural resources are a major target for this purpose. Modern multicores are an example, as they often feature a last-level cache shared among different cores. Suzaki et al. [30] showed that shared caches can be used as a side channel to disclose the existence of other virtual environments on the same physical machine.

Other researchers demonstrated covert channels that exploit a shared cache to transmit information between two virtual environments running on the same multicore [36, 37]. Besides caches, also other shared microarchitectural resources, e.g., branch predictors [9], were used as covert and side channels; Hunger et al. [14] recently proposed a *bucket model* that captures the common characteristics of these microarchitectural side and covert channels.

2.2. Thermal-Related Attacks

Another target for the realization of side channels are the physical characteristics of the CMOS implementation of a chip. For example, Hutter and Schmidt [15] demonstrated a temperature side channel able to retrieve the private key from an RSA implementation on an AVR microcontroller. They decapsulated the chip to measure the temperature directly on the surface of the silicon substrate and operated the device at 150 °C, beyond its specified temperature range. They found that, under these conditions, the device leaks the Hamming weight of the processed data via the temperature side channel. They exploited this property to retrieve the private key by correlating the temperature, execution, and power traces of the chip for several runs.

Other researchers presented a denial of service attack by creating a hot spot on the silicon to trigger DTM and induce performance throttling [11]. Similarly to this work, our covert channel is based on heat dissipation and temperature variations in chips based on CMOS technology.

2.3. Temperature-Based Covert Channels

Previous work studied covert channels based on different effects related to temperature variations on CMOS chips.

A well-studied timing channel exploits the local clock skew introduced by temperature variations [23, 26, 38, 39]. If a source app can trigger temperature variations on a victim host, it can induce skew in the local clock; the sink app can observe the skew by looking at timestamps and comparing to a reference clock. This channel was exploited to reveal hidden services [23, 26, 38], for example services running under the Tor network. The attacker induces a load pattern that triggers temperature variations on the victim host by frequently accessing the hidden service. The attacker can then localize the hidden service by observing the clock skew of a set of candidate hosts. Another research exploited the same timing channel to infer the topology of a public cloud infrastructure [26]. Zander et al. [39] estimated the capacity of this timing channel to be up to 20.5 bits per hour. Besides clock skew, previous work also investigated channels based on other side-effects related to temperature variations. For example, Brouchier et al. [6] studied a storage channel based on fan speed on a desktop and a laptop.

In contrast to these channels, which exploit side-effects of temperature variations, we focus on the storage channel where the sink app directly observes on-chip temperature variations. This storage channel is not totally new; variants of it were studied in previous work on different platforms. Guri et al. [10] recently studied an indirect variant of this channel to attack air-gapped systems. They showed that communication is possible between two nearby, air-gapped desktops by using the available temperature sensors: the source app runs on one desktop and controls load; the sink app runs on the other desktop and observes temperature variations caused by the heat coming from the source. Variants of the channel that exploit on-chip heat transmission were studied on FPGAs configured with isolated components that cannot communicate through the logic [5, 16, 20]. Work in this direction showed that communication between the isolated components is possible through a covert channel similar to the *1-hop channel* that we study in this paper.

In the previous work more closely related to our research, Masti et al. [21] present an initial study of the 1-hop and 2-hop channels on multicore processors. They show experiments that achieve a transmission rate of up to 1.33 bits per second (bps) with an error rate of 11% for the 1-hop channel on an Intel Xeon-based server. This work only looks at these channels from an empirical perspective, while we present a new methodology that uses both experimental results and theoretical analysis to characterize the family of thermal covert channels (including the same-core channel, see Section 4) on modern multicores. Thanks to this methodology, we are able to provide upper bounds on the channel capacity, which they did not study; moreover, we show a transmission scheme that, at the same 11% error rate, achieves a 20× faster rate of 27 bps for the same channel on the same platform they used.

3. Threat Model

We are interested in the scenario introduced in the example of Figure 1. Without loss of generality, we assume that the sink app just records a temperature trace by reading the sensors and later sends it to the attacker over the network; message decoding is done offline by the attacker. Thus, the sink app is mostly idle and only periodically wakes up to read the sensor.

We target modern mobile devices, which implement per-core sleep states to extend battery life. On these devices, the Operating System (OS) puts idle cores to sleep and, when sleeping, cores consume close to zero power and produce almost zero heat. On Intel Core processors, when scheduling the idle thread the OS calls the `mwait` instruction to switch the current core from the active state to a lower *c–state* and save power. For instance, the `C1-HSW` state, which implements clock-gating on the Haswell generation of these processors, brings most of the power savings for a cheap wakeup latency of $2\,\mu$s [2]. Switching to deeper c-states saves more power, but implies a higher wakeup latency, up to hundreds of μs. ARM big.LITTLE multicores implement a similar, while simpler, hierarchy, where the `C1` state implements clock-gating. Assuming no scheduling artifacts, even a costly wakeup latency of $200\,\mu$s only puts a loose upper bound of 5 KHz on how fast the source app can switch.

We note that the mobile devices that we target are idle or lightly-loaded most of the time (e.g., a smartphone resting in a pocket or a laptop just running a text editor). Thus, the source and sink app can wait for the system load to be low before starting to use the covert channel, so as to avoid interference. We briefly evaluate the impact of background load in Section 7.3, but we leave a more detailed study of interference to future work. In this paper, we focus on bounding the channel capacity and studying achievable rates in controlled, while still realistic, conditions that enable repeatability of our experiments. Thus, we set the environment to limit interference and noise as much as possible (Section 5); Section 7.3 presents a study of the sensitivity of our results to departure from this controlled environment.

Finally, we note that modern mobile multicores, e.g., Intel Core mobile processors or ARM multicores, generally feature one temperature sensor per core and that these sensors are easily accessible by userspace processes or apps. For instance, on Linux, `lm_sensors` exports a simple command-line interface; on Windows, CoreTemp offers a graphical interface. While setting up these tools might require administrative rights (e.g., `# sensors_detect`), they are commonly installed on client devices. On Android devices, the temperature sensors are even easier to access for the apps: we verified that the CPU-Z app (v. 1.15), available on the Google Play Store, requires no system permissions to be installed and it reports several temperature measurements on a Nexus 4 running Android 5.0.2.

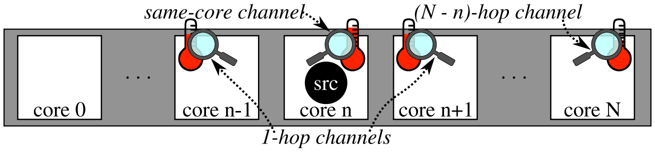

Figure 2: The sink app can establish several channels, depending on the physical location of the temperature sensor it reads with respect to the location of the source app.

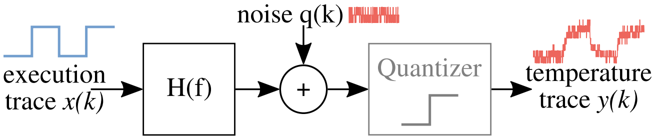

Figure 3: Discrete linear channel model with transfer function $H(f)$ from the execution trace $x(k)$ to the temperature trace $y(k)$, with additive noise $q(k)$. In our analysis, we neglect the quantizer.

Moreover, once the sensors are exposed, any app can read all sensors through the userspace interface, regardless of which core it runs on.

4. Communication Channel Model

We study a family of *storage covert channels* [22, 32] where a source and a sink app share a multicore processor and covertly communicate through the on-chip temperature sensors. Assuming that the source app runs on core n, we can define at least as many channels as there are temperature sensors. Similarly to previous work [21], we consider one sensor per core and a floorplan with cores in a linear array, as commonly found on multicores with a moderate number of cores.While the actual floorplan of our experimental platforms is not documented, the results we obtain are compatible with this assumption; our definitions can be adapted to a more general topology. Since the sink app is mostly idle and, on current systems, it usually has access to all the sensors, it is not so important on which core it runs; we just assume that it runs on a different core than the source app. As Figure 2 illustrates, when the sink app reads the temperature of core n (the one where the source app runs), we have the *same-core channel*. Similarly, we have an *m-hop channel* when the sink app reads the temperature of a core m hops away from core n, i.e., core $(n \pm m)$.

We expect the same-core channel to have the highest capacity, as the thermal resistivity of silicon degrades the signal for the m-hop channels. In fact, the sink app can simply record a trace for each sensor and send all the data to the attacker, who could always exploit the same-core channel. Studying the m-hop channels is, however, still interesting, since system virtualization may restrict the sink app to only have visibility over the sensor of its local core(s).

We consider the discrete-time channel model of Figure 3. The input to the channel is $x(k)$, the execution trace of the source app; at each instant k, $x(k) = 0$ if the source app is idle and $x(k) = 1$ if it is active. The output of the channel is $y(k)$, i.e. temperature trace from the corresponding sensor. Similar to previous work [19, 24, 29], we use the linear block with transfer function $H(f)$ to model the temperature variations at the sensor caused by the execution trace. The additive noise $q(k)$ models thermal noise and any disturbances from other apps or the OS. The quantizer block models the fact that commercial processors offer a coarse sensor resolution, e.g., 1 °Con our two platforms. Explicitly considering the quantizer might increase the model accuracy, but adds a nonlinear component, which is complex to analyze. For this reason, in our analysis we ignore the quantizer and consider a linear approximation of the system. Our results (Sections 6 and 7) indicate that this approximation is reasonable.

Thanks to the model of Figure 3 (excluding the quantizer), we can employ the powerful tools available for the analysis of discrete linear dynamic systems for estimating the channel capacity (Section 6). Additionally, we refer to this model to design the experiments that evaluate the throughput achieved with our transmission scheme (Section 7).

5. Experimental Setup

We base our analysis on experimental data collected from two diverse and representative hardware platforms:

1. a Lenovo ThinkPad T440p laptop, featuring a quad-core Intel Core i7-4710MQ processor clocked at 2.5 GHz;
2. an Odroid-XU3 board, featuring a Samsung Exynos 5422 SoC including an ARM big.LITTLE processor with two quad-core clusters of Cortex-A7 and Cortex-A15 cores, respectively. The big cluster is clocked at 2.1 GHz.

In the rest of the paper, we refer to platform 1 as *Laptop* and to platform 2 as *Smartphone*. *Laptop* is representative of current business laptops; *Smartphone* is representative of hand-held devices (it has the same SoC as the Samsung Galaxy S5 SM-900H smartphone). We use the two platforms both to analyze the channels for capacity estimation and to evaluate a communication scheme that achieves higher rates than previous work; in both cases, we use the following experimental setup. Additionally, we reproduce previous results [21] on our two platforms and evaluate our communication scheme on a third *Server* platform (Section 7).

5.1. System settings

On both *Laptop* and *Smartphone*, we install Ubuntu 14.04.2 and we use the `/dev/cpu_dma_latency` interface of the Linux kernel to limit the maximum wakeup latency to 10 μs. With this setting, the deepest *c-state* for *Laptop* is limited to `C1E-HSW`, with a wakeup latency of 10 μs; the deepest sleep state for *Smartphone* is `C1`, with a wakeup latency of 1 μs[1].

On *Laptop*, the temperature sensors are refreshed every 1 ms [17]. We were not able to find the sensors refresh period for *Smartphone* on the SoC documentation. To determine

[1] We retrieve wakeup latencies from the `sysfs` interface exposed at `/sys/devices/system/cpu/cpu$i/cpuidle/state/$n/latency`.

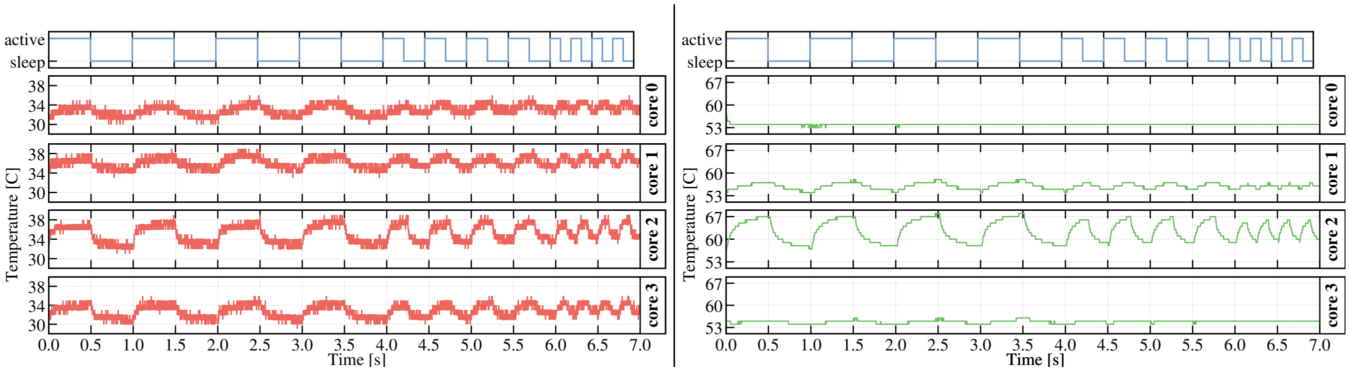

Figure 4: Traces from *Laptop* (left) and *Smartphone* (right) when the source app executes on core 2; the top plot shows the active/idle execution trace of the source app, the other plots show the temperature traces from the four cores.

this parameter, we collected several traces with a varying system load, using 1 ms as the sampling period; we noticed that the temperature only changed every 5 ms, which we take as the sensor refresh rate for this platform. Based on these characteristics, we set the sampling period to $T = 1$ ms for *Laptop* and $T = 5$ ms for *Smartphone*. Therefore, the Nyquist frequency of our discrete system is 0.5/1 ms= 500 Hz for *Laptop* and 100 Hz for *Smartphone*.

To favor repeatability, we run all experiments in a controlled, while still realistic, environment. We set both devices in an air-conditioned server room with an ambient temperature of $\approx 23\,\mathrm{C}^\circ$ and, for both, we fix the fan speed to the maximum level[2] and set the clock frequency of active cores to the maximum, i.e., 2.5 GHz for *Laptop* and 2.1 GHz for the big cores on *Smartphone*. In order to avoid scheduling artifacts, we run the source and sink app with the `SCHED_FIFO` scheduling class at highest priority by using the `pthread_setschedparam()` interface and we pin the source app to one core by using the `pthread_setaffinity_np()` interface. During all experiments, the system is idle except for the source and sink apps and the default system services of the Ubuntu installation.

For both the four cores of *Laptop* and the four big cores of *Smartphone* we assume a linear floorplan, as shown in Figure 2. While the actual floorplan of the two platforms is not documented, our results are compatible with this assumption. We run the source app on the third core in the array, i.e., on core 4 on *Laptop*, which has eight virtual cores with two-way hyper-threading, and on core 6 on *Smartphone*, where cores 0 to 3 are the LITTLE cores and cores 4 to 7 are the big cores. In the rest of the paper, we only count the four physical (big) cores, starting from 0; thus, for both platforms, we say that we run the source app on core 2 and we record the temperature traces from cores 0 to 3. On *Smartphone*, we run the source app on the big cores, since the LITTLE cores provide no temperature sensors and they do not sensibly affect the measurements on the big cores. This setup allows us to analyze one same-core channel (when looking at the temperature trace of core 2), two different 1-hop channels (when looking at either core 1 or core 3), and one 2-hop channel (when looking at core 0).

```
int time, state;
while (!cin.eof()) {
  cin >> time;
  cin >> state;
  if (state)
    run_for(time);
  else
    usleep(time);
}
```

Snippet 1: Stripped-down code for the reference source app.

```
string log;
ofstream logfile;
// initialize...
while (!interrupt) {
  log_temps(&log);
  // sample every T us
  usleep(T);
}
logfile << log;
```

Snippet 2: Stripped-down code for the reference sink app.

On *Laptop*, we exploit hyper-threading and we run the sink app with four parallel threads on the odd-numbered virtual cores; each thread reads the temperature of its core from the `/dev/cpu/$i/msr` interface. On *Smartphone*, all the sensors are exposed in the single virtual file `/sys/devices/10060000.tmu/temp`; here we run the sink app single-threaded on the first LITTLE core. This setup avoids timing interference between the source and the sink app.

Unless differently specified, we use these settings in all our experiments. Since a real attack would not benefit from this controlled environment, in Section 7.3 we analyze the sensitivity of our results to variations to these settings.

5.2. Reference apps

We develop a reference source and sink app in C++. Snippets 1 and 2 show the key parts of their main loop.

The source app (Snippet 1) replays the execution trace that is passed on standard input (`cin` in C++ terminology). If the next state is 1 (active), then it keeps the core active for the specified time; if the next state is 0 (sleep), it goes idle by calling `usleep()`. The `run_for()` function executes a tight loop similar to the one of the popular `cpuburn` stress-test[3]; the loop periodically (every several iterations, about

[2] For *Laptop*, `# echo 'level 7' > /proc/acpi/ibm/fan`; for *Smartphone*, `# echo 255 > /sys/devices/odroid_fan.14/pwm_duty`.

[3] https://patrickmn.com/projects/cpuburn/

every 1 μs) checks whether the elapsed time exceeded the requested active time and terminates when this condition is verified. For this check, we use the `gettimeofday()` call, which proves precise enough for this purpose; since we are keeping the core active anyway, its overhead is not so important in this case. Additionally (not shown in Snippet 1), the source app keeps track of the overall elapsed time and keeps adjusting the value of `time` to avoid drifting apart due to jitter in `run_for()` or `usleep()`. We find `gettimeofday()` precise and lightweight enough also for this task.

The sink app (Snippet 2) samples the temperature sensors every T μs ($T = 1000$ for *Laptop*, $T = 5000$ for *Smartphone*) and keeps a preallocated in-memory `log`, which it dumps to the `logfile` at the end. Similarly to the source app, the sink app keeps track of the elapsed time and adjusts `T`, in order to avoid long-term timing skew. The parallel version of the sink app that runs on *Laptop* additionally handles thread synchronization through barriers. We register a signal handler to set the `interrupt` flag at the experiment end and, at that point, we retrieve the log file and analyze it offline.

5.3. Platform characterization

Figure 4 shows the results of a preliminary experiment that characterizes the temperature range and dynamics of our two platforms. On both *Laptop* (Figure 4, left) and *Smartphone* (Figure 4, right), the source app runs on core 2 with the execution trace shown in the top plots (blue lines). The execution trace is an active/sleep square wave with 50% duty cycle and varying frequency, with 4 periods each at 1 Hz, 2 Hz, and 4 Hz. The bottom plots report the resulting temperature traces for cores 0 to 3, i.e., for the same-core channel (core 2), the two 1-hop channels (cores 1 and 3), and the 2-hop channel (core 0).

For both platforms, the same-core channel resembles the response of a low-pass filter that oscillates between a high and a low value with a smoothened version of the input wave. In both cases, it is easy to see that it is possible to reconstruct the input wave from the temperature trace for the whole experiment. As expected, the execution trace is harder to infer from the 1-hop channels, due to the farther distance on the silicon of the corresponding sensors from the area that generates heat. Moreover, the two 1-hop channels show a different amount of attenuation and distortion: the trace from core 3 looks "better" than core 1 for *Laptop*, while the opposite is true for *Smartphone*. Finally, *Laptop* shows much less attenuation for the 2-hop channel than *Smartphone*, for which the temperature trace is basically flat, making the input trace impossible to reconstruct.

The dynamic temperature range on the different channels is also different across the two devices, as Table 1 highlights. For the same-core channel, *Smartphone* has a wider dynamic range of 10 °C, while the dynamic range on *Laptop* is just 7 °C. On the contrary, for the 1-hop channels the dynamic range is wider on *Laptop*, where it is at least 6 °Cfor both core 1 and core 3, compared to the dynamic ranges of just 4 °Cand 2 °C, respectively, measured on *Smartphone*. Similarly, *Laptop* has a much wider dynamic range on the 2-hop channel, which still oscillates up to 6 °C, while for *Smartphone* the temperature trace of core 0 is basically flat. This different behavior depends on the floorplan, fabrication characteristics, and cooling system of the two platforms. Two characteristics that probably play a role are the lower TDP (less than 20 W versus 47 W TDP) and the reduced package area (213 mm^2 versus 1200 mm^2) of the big.LITTLE SoC of *Smartphone* compared to the Intel Core processor of *Laptop*.

Platform	core 0	core 1	core 2	core 3
Laptop	[30,36] °C	[33,39] °C	[32,39] °C	[29,36] °C
Smartphone	[53,54] °C	[54,58] °C	[58,68] °C	[54,56] °C

Table 1: Dynamic temperature ranges measured on *Laptop* and *Smartphone* for the experiment of Figure 4.

Intuitively, on both platforms and for all channels, the dynamic range shrinks as the frequency of the input increases. As a notable example, the temperature trace of core 3 of *Smartphone* shows significant variations as long as the input frequency is 1 Hz, but the signal is quickly lost as the frequency increases (from time 4 s on).

Finally, another important difference between the two platforms lies in the incidence of noise in the temperature traces. The traces from *Laptop* present a sensible amount of noise, with the temperature constantly oscillating by 1 °C. Instead, the traces from *Smartphone* show almost no noise and have an accentuated staircase-like quantization effect, probably due to internal filtering in the sensors, which have a slower refresh rate compared to *Laptop* (5 ms versus 1 ms). The lack of noise on *Smartphone* accentuates the signal attenuation at higher frequencies and further distance, since temperature variations are only observable if the actual temperature varies across a quantization boundary; otherwise, variations are hidden by the quantization. Despite this difference, we found that we are able to stick to the linear channel model of Figure 3 for both platforms in our study to estimate the channel capacity (Section 6).

The heterogeneity in the behavior of these two platforms makes them good candidates as the source of representative data for our study of the capacity bounds (Section 6) and for the evaluation of our communication scheme (Section 7).

6. Capacity Estimation

In the 1985 *Orange Book* [32], the US department of defense reports that "*a covert channel bandwidth that exceeds a rate of one hundred (*100*) bits per second is considered high*" and that covert channels with "*maximum bandwidths of less than one (*1*) bit per second are acceptable in most application environments*". While these numbers may look somewhat different if estimated today, the 1.33 bps transmission rate with 11% error probability achieved by Masti et al. [21] for the 1-hop channel seems too low to be considered a threat in practice.

Still, much higher rates with much lower error probability are possible when considering the same-core channel or a better communication scheme, as we show in Section 7. In order to evaluate whether these channels can or cannot be a security threat, we need to find a reliable estimation of their capacity C, i.e., we need to find the upper bound on the rate of communication achievable through them with arbitrarily small error probability [7, 28].

Following Shannon's seminal work [28], researchers extensively studied ways to determine the capacity of a wide range of channel models [7]. Still, even with this vast theoretical literature available, estimating the capacity of a physical channel remains very challenging: it requires using an appropriate model and retrieving quantitatively accurate measurements of the channel parameters, despite of noise and limited precision. We tackle this challenge by leveraging the simple model described in Section 4 and determining its transfer function $H(f)$ through carefully designed experiments based on the experimental setup described in Section 5.

6.1. Finding Capacity Bounds
— *The theory* —

The first step towards determining a good estimate of the channel capacity C is finding a suitable mathematical expression to compute it based on observable parameters. One of the simplest expressions for the channel capacity is given by the Shannon-Hartley theorem [7], reported in Equation (1). The

$$C = B \log_2 \left(1 + \frac{S}{N}\right) \text{ [bps]} \tag{1}$$

theorem gives the capacity C for the ideal, additive white gaussian noise (AWGN), band-limited channel with bandwidth B and Signal-to-Noise-Ratio (SNR) S/N.

Since Equation (1) applies exactly only to an ideal, band-limited, channel, we first need to verify whether we can reasonably approximate our channels this way. If this approximation is possible, we can determine the bandwidth B and the SNR S/N of our channels based on experimental measurements and use these values to estimate the capacity. In order to find the bandwidth, we try to fit a discrete-time dynamical system model to match the dynamics of the channels. For instance, we were able to fit the same-core channel of *Smartphone* with a discrete-time model with six poles and four zeros [13]. Figure 5 shows the measured and modeled step response for this channel (left) and the magnitude Bode diagram of the corresponding model (right). The step response plot visually shows how well the model can predict the measured behavior of the channel, while the Bode diagram shows the asymptotic approximation of the frequency response in logarithmic scale. In an ideal band-limited channel, we would expect the bode diagram to have a *rectangular* shape that lets a band of frequencies pass and blocks all the rest of the spectrum. While the model fits the step response well (the normalized mean-squared-error is 4.7%), its Bode magnitude plot does not allow to easily define the bandwidth B. On the one hand, the commonly used cutoff frequency at the 3 dB drop (shown in Figure 5) does not seem to be a good choice to determine the bandwidth in this case, since the magnitude keeps decreasing slowly up to about 10 Hz, where there is a clear knee. On the other hand, using the frequency at the knee for the bandwidth would be rather arbitrary as well, since the amplitude is far from constant up to there, with a 15 dB drop. Moreover, looking at the preliminary experiment of Figure 4, we notice that there is a significant attenuation when increasing the input frequency, even just from 1 Hz to 4 Hz; therefore, using a fixed SNR value for the whole passband would not be accurate. In the interest of space, we omit the step responses and Bode diagrams for the other channels and for *Laptop*; similar considerations apply in those cases. From these observations, we conclude that Equation (1) is not adequate to estimate the capacity of our channels, since we are not able to reliably estimate the required parameters.

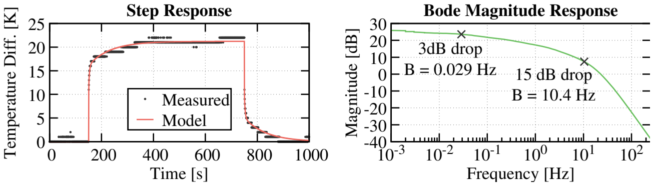

Figure 5: Step response of the same-core channel on *Smartphone*; the input is 1 in the interval $[150, 750)$ s, 0 elsewhere.

While using the Shannon-Hartley theorem is not effective in our case, we can leverage a different approach to find the capacity [7, 33]. We can search, among all the possible input patterns $x(k)$, the one that has the frequency characteristics that make the most information pass through the channel; in other words, we need to find the best allocation of the input power $\hat{S}_{xx}(f)$ across the frequency spectrum. If we can find this ideal allocation $\hat{S}_{xx}(f)$, we can use results from the information theory literature to compute the channel capacity. The key observation in this method is that we can only allocate as much power as we are able to put into our input signal, i.e., we have a power cap p_0 on how much power we can input into our system. The general approach to determining $\hat{S}_{xx}(f)$, and thus C, subject to a power cap p_0 is known as *water-filling* [7, 33] The *water-filling* technique is based on the assumption that the optimal input spectrum is the one that allocates power such that the sum of the noise and the signal power is constant over the whole channel spectrum; so more power of the signal is in parts of the spectrum with high SNR. We study two different solutions based on this technique. First, we consider the *classic* solution [33], which considers the constraint p_0 on the average input power. Second, we analyze a *constrained-input* solution [12] that explicitly considers the extra constraint that the input to our channels is a binary value (active/idle).

Classic water-filling approach. The classic water-filling technique allows to compute the capacity of channels with arbitrary transfer function $H(f)$ and additive Gaussian noise

$q(k)$, not necessarily white [7, 33]. If we can estimate the power spectrum of the channel $S_{hh} = |H(f)|^2$ and of the noise S_{qq} then, given a cap p_0 on the average input power, we can derive the channel capacity according to Equation (2) [33, Eq. (6.15)]. The capacity C_b is determined by the *spectral power allocation* $S_{xx}(f)$, which cannot exceed the power cap p_0, as Equation (3) states. We can maximize the expression in Equation (2) and determine the capacity by intelligently shaping the power allocation S_{xx} so that more power is allocated at those frequencies with better SNR. This ideal allocation $\hat{S}_{xx}$ can be determined with a *water-filling* procedure [7, 33], which we do not describe in details here.

$$C_b = \max_{S_{xx}} \left\{ \int_{\mathscr{F}} \log_2 \left(1 + \frac{S_{xx}(f) \cdot S_{hh}(f)}{S_{qq}(f)} \right) df \right\} \text{ [bps]}, \tag{2}$$

$$\text{under the constraint that } \int_{\mathscr{F}} S_{xx}(f) df \le p_0 \tag{3}$$

As we will show in Section 6.2, we are able to estimate S_{hh} and S_{qq} for our channels; thus, we can use the water-filling procedure on Equation (2) to estimate the capacity C_b. We expect C_b to be an upper bound on the real capacity C, because the classic water-filling approach does not consider the more stringent constraint that our input is required to be a binary value. In order to evaluate how much more stringent this constraint is, we use an additional result from the literature to compute a tighter upper bound on the real capacity.

Constrained-input water-filling. In a 1992 paper, Heegard and Ozarow [12] studied the capacity of *saturation recording*, i.e., the capacity of storage systems such as tape recorders or optical disks. While this problem has, in general, little to do with our study, it has the same *saturation* constraint on the channel input: input values can only be either 0 or 1. This shared property allows us to leverage their expression for an upper bound C_a on the channel capacity C [12, Eq. (11)]. We report this result (with minor notation changes) in Equation (4). C_a depends on the value of the power spectrum of the channel S_{hh} and the parameter λ over A_λ, which is the set of frequencies $f \in (-\infty, \infty)$ for which $\lambda \cdot S_{hh} \ge 1$. The parameter λ must be maximized subject to the constraint of Equation (5), which makes sure that the SNR does not exceed the ratio of the power cap p_0 over the noise power N_0. These equations assume that the noise is white, i.e., that the noise has a constant power spectrum $S_{qq} = N_0$ across the frequency range A_λ. Since, in our channels, S_{qq} is not constant, we use this constrained-input solution only after splitting the channel into sub-bands where S_{qq} can be assumed constant; Section 6.3 explains this technique in more details. Finding the λ that maximizes Equation (4) subject to Equation (5) follows again a water-filling procedure.

$$C \le C_a = \max_{\lambda} \left\{ \frac{1}{2} \int_{A_\lambda} \log_2 (\lambda \cdot S_{hh}(f)) \, df \right\} \text{ [bps]}, \tag{4}$$

$$\frac{1}{2} \int_{A_\lambda} \left(\lambda - \frac{1}{S_{hh}(f)} \right) df \le \frac{p_0}{N_0} \tag{5}$$

6.2. Determining the Power Spectra — *The practice* —

To use the water-filling methods, we need to find reliable estimates for the power spectra of the noise and our channels on our two platforms. Computing reliable estimates from experimental data is challenging mainly due to (i) the limited temperature resolution (1 K) of the sensors, (ii) the noise (on *Laptop*), (iii) the quantization effect (on *Smartphone*), and (iv) the saturation constraint on the input.

Noise spectra. S_{qq} is easier to estimate than S_{hh}, since the input constraint does not play a role in this case. For both platforms, we just record a 120 s long temperature trace for each channel, with the system idle except for the sink app, which records the traces, and the default system services. Then, we compute the power spectral density $S_{qq}(f)$ over the frequency range $[0.5, f_m]$ Hz for each channel, with $f_m = 250$ for *Laptop* and $f_m = 100$ for *Smartphone*, which is limited by the lower sampling rate. After subtracting the mean value from the temperature traces, to remove the DC component, we get the spectra through fast fourier transforms (FFTs) [3] of each temperature trace. To improve the accuracy of our analysis, we use Welch's method [35] and a Blackman-Harris window [1]. Welch's method is commonly used to minimizes the variability in the calculation of the power spectral density, i.e. the noise in the power spectrum, compared to standard Fourier analysis. The Blackman-Harris window is designed to minimize the side-lobes in the frequency domain and therefore the influence of neighbouring frequencies on each other. We report the resulting high-resolution noise spectra in Figure 6, together with the channel spectra S_{hh}, which we illustrate next.

Channel spectra. Determining S_{hh} is more challenging because of the constraint on the input. This constraint basically restricts the variety of input signals that we can use to rectangular waves of different frequency, similar to the one we used in the preliminary experiment of Figure 4. Our approach to determine S_{hh} consists in designing a set of experiments $\{\mathscr{E}_f\}$ where experiment $\mathscr{E}_f$ gives us an estimate of the value of the channel power $S_{hh}(f)$ at frequency f. We go into the details through the example of Figure 7, which illustrates how we determine S_{hh} for the 1-hop channel of core 1 of *Laptop*. The data used to draw Figure 7 come from five separate experiments $\mathscr{E}_f$, with $f \in \{5.1, 14.9, 25.0, 34.5, 45.5\}$ Hz. Each experiment $\mathscr{E}_f$ consists in using a modified version of the source app to excite the system with a square wave at frequency f and in computing the power spectra of the input and the output, which are superimposed in the left and right plots of Figure 7, respectively. To compute these spectra, we use the same FFT-based method that we use to compute S_{qq}. The

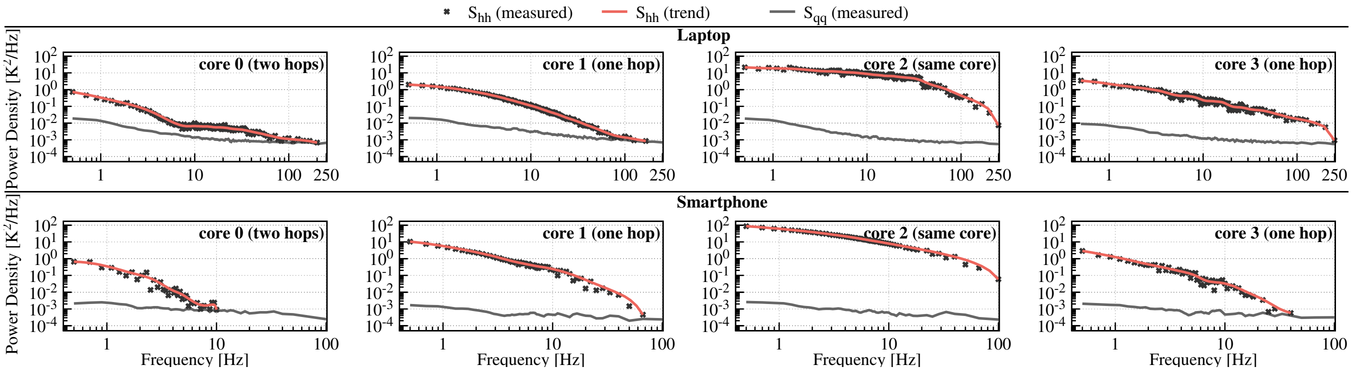

Figure 6: Power density spectra S_{hh} for the four channels measured on *Laptop* (top) and *Smartphone* (bottom). The crosses are measured values and the red solid line is the bezier trend for S_{hh}. The dotted grey lines are the spectra of the noise S_{qq}. Both axes are in logarithmic scale.

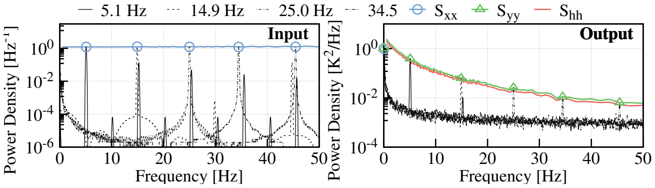

Figure 7: Input (left) and output (right) spectra from core 1 of *Laptop* for the five experiments $\mathscr{E}_f$ at the frequencies f reported in the legend. We use the spectra peaks to build S_{xx} and S_{yy}; then, $S_{hh} = S_{yy}/S_{xx}$. The y-axis is in logarithmic scale.

spectra from experiment $\mathscr{E}_f$ show a peak at frequency f, which is where most of the power is allocated. We take these peaks as the values of the input S_{xx} (blue circles in Figure 7) and output S_{yy} (green triangles in Figure 7) power spectra. Then, we can simply compute the power spectrum of the channel S_{hh} as the sample-wise output-over-input ratio S_{yy}/S_{xx}. Figure 6 reports the values of the S_{hh} spectra that we derive with this methodology for the four channels on our two platforms, along with the noise spectra S_{qq}.

6.2.1. Additional notes on the experiments $\{\mathscr{E}_f\}$

Each experiment $\mathscr{E}_f$ lasts 120 s on *Laptop* and 600 s on *Smartphone*, so that we collect the same number of samples (120 k) for both platforms. The longer experiments on *Smartphone* also help to make sure that we can actually observe enough variations in the temperature traces to build a meaningful spectrum (recall the accentuated quantization effect on *Smartphone* that was discussed in Section 5.3). Finally, for all the channels, we only keep the S_{hh} points up to the frequency f where $S_{hh}(f)$ drops at or below the noise level $S_{qq}(f)$.

We determine the frequency range $\{f\}$ for the experiments $\{\mathscr{E}_f\}$ so as to reduce measurement errors as much as possible. We only use frequencies that, at the sampling period of either 1 ms (*Laptop*) or 5 ms (*Smartphone*), have an integer number of samples per period of the square wave. We start from 0.5 Hz and we proceed in steps of either 0.2 Hz or one fewer sample per period, whichever yields the largest step. The crosses in Figure 6 are located at these frequencies along the x-axis. In total, we evaluate 138 different frequencies for *Laptop* and 60 different frequencies for *Smartphone*.

Due to the constraints on the input, we use square waves as an approximation of sine waves, which would be the most appropriate waveform to concentrate the input power at the corresponding frequency. In practice, the non-idealities of our channels (particularly, the c-state sleep/wakeup latency) make sure that our *logical* square waves are really steep ramps that approximate a sine wave well enough. In fact, the spectra of Figure 7 clearly show the peaks at the fundamental frequencies, with some negligible harmonics.

One way to better approximate sine waves on the input would be to use active/sleep Pulse-Width-Modulation (PWM) at at a rate r much higher than the frequency corresponding to the sampling time T we use (i.e., $r \gg 1\text{KHz}$). In this way, it is possible to obtain different power levels and to generate a sampled sine wave. Since the c-state and scheduling latencies are fast enough to do so, we actually implemented this PWM approach in a modified version of the source app. However, we found that the results were not significantly different; thus, we decided to stick with the "square" waves.

6.3. Computing the Capacity Bounds — *Theory meets practice* —

We can finally compute the two capacity bounds C_b and C_a, with the classic and constrained-input water-filling methods, respectively. Since we work with discrete spectra, we accordingly adapt the equations of Section 6.1 to use summations instead of integrals and to consider the discretization intervals along the frequency range. While the noise spectra S_{qq} already come with a high frequency resolution, the S_{hh} spectra are more coarsely quantized, as the crosses in Figure 7 show. To simplify the computations, we linearly interpolate all the spectra on a regular frequency grid with 0.1 Hz spacing.

Classic water-filling. This method can handle non-white noise spectra S_{qq}, which is the case in our measurements (see Figure 6). We determine the input power cap p_0 as the average of measured input spectrum S_{xx}. To find C_b, we compute the

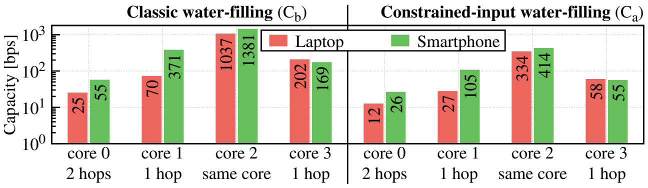

Figure 8: Upper bounds C_b (left) and C_a (right) on the channel capacity C for the four channels on *Laptop* and *Smartphone*. The y-axis is in logarithmic scale.

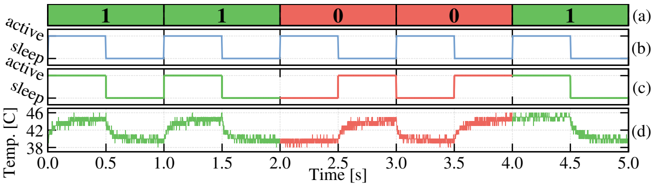

Figure 9: An input message (a), encoded onto the 1 Hz clock (b), gives the execution trace (c), which leads to the temperature trace (d) on the same-core channel of *Laptop*.

ideal power allocation $\hat{S}_{xx}$ by iteratively refining the value of the parameter λ until the condition of Equation (5) is met (almost) with equality (with a maximum error of 10^{-6}).

Constrained-input water-filling. In order to compute C_a, Equation (4) assumes that the additive noise is white, with constant power density $S_{qq} = N_0$ across the relevant frequency range. However, our measured S_{qq} spectra vary significantly across the frequency range we are interested in. To address this issue, we split the channel into sub-bands [33, Chap. 6.5] where the noise S_{qq} does not vary by more than 50 % of the smallest value in the sub-band. This operation gives us about 10 to 20 sub-bands per channel, depending on the different shape of the S_{qq} spectra. For each sub-band k over the frequency range $[f_i, f_j)$, we use the reference noise level $N_k = \text{mean}\{S_{qq}(f)\} \mid f_i \leq f < f_j\}$. Given the global power cap p_0, which we determine as in the classic water-filling case, we compute the optimal allocation to the sub-bands based on their width and their noise level [33, Chap. 6.5]. Finally, we consider one sub-band at a time and we independently compute the capacity in an iterative way, similar to how we do it for the classic case. To compute C_a, we sum the resulting capacity in all the sub-bands.

Capacity bounds.Figure 8 shows the capacity bounds C_b (left) and C_a (right) that we compute with the classic and constrained-input water-filling methods, respectively. As expected, $C_b > C_a$ and the bound for the same-core channel is the highest for both platforms and both methods. In general, the trend across the four channels seems consistent on the two platforms and the bounds on the two different 1-hop channels are consistent with the observations of Section 5.3: the channel on core 1 is better than the one on core 3 for *Smartphone*, while the opposite is true for *Laptop*. These results do not exclude that the same-core channel might be a security threat, with C_a well above 100 bps for both platforms. While the bounds for the 1-hop channels are (mostly) below 100 bps, they are still much higher than our initial expectations based on previous research. In Section 7 we show a transmission scheme able to notably increase previous results on transmission rates.

7. Transmission Scheme and Achieved Rates

The transmission scheme that Masti et al. [21] used to evaluate the 1-hop channel is based on *ON-OFF* keying: the source app is active to transmit a 1 and it goes idle to transmit a 0. A major issue with this simple scheme is that the average load level depends on the input message: a message with several *ones* (respectively, *zeros*) in a row will leave the source core active (respectively, idle) for a long time compared to the symbol duration, causing the average temperature to drift up and down. This drift of the operating point unpredictably changes the temperature dynamics over time, making the channel non-stationary and the decoding more complicated. This issue, coupled with the simplistic edge-detection decoding method they used, could explain the poor performance that they measured (see Section 2.3) compared to the capacity bounds that we derived in Section 6. In this section, we evaluate a simple communication scheme that overcomes this issue.

7.1. Encoding and Decoding Scheme

A simple way to keep the channel in the dynamic range during communication is to encode the input message so as to maintain, on average, a constant load. To do so, we use square waves with a 50% duty cycle as a *clock* signal onto which we *encode* the input message.

Message encoding. We generate the execution trace of the source app with the *Manchester encoding* scheme [31], as Figure 9 illustrates for a 5-bit message and a 1 Hz clock. A *one* in the message is encoded into an unmodified clock signal in the execution trace; a *zero* becomes a 180° phase-shifted clock signal in the execution trace. The resulting execution trace leads to temperature traces oscillating around a roughly constant average, as Figure 9 (d) shows for the same-core channel on *Laptop*. The transmission rate directly depends on the frequency of the clock signal, since the trace carries 1 bit of information per period of the clock, i.e., r bps for a r Hz clock.

Message decoding. Message decoding happens offline (see Section 3) from the temperature traces recorded by the sink app. The first step of decoding is determining the phase of the clock signal. For simplicity, we synchronize our experiments so that the beginning of the temperature trace coincides with the beginning of the message. In a real attack, where this synchronization would not be possible, the source app could send a known preamble that the sink app can use to detect the clock phase. Once the clock phase is detected, it will not change during an experiment, since our source and sink app

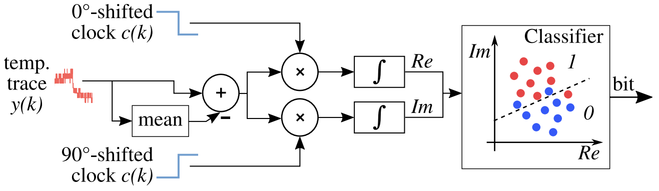

Figure 10: Block diagram of our bit-wise decoding scheme.

are designed to not accumulate clock skew (see Section 5.2). To proceed with decoding, we look at each clock period, i.e., at each bit, separately. As Figure 10 shows, for each bit, we first get a 0-mean signal by subtracting its mean temperature; in this way, the decoding is robust against long-term temperature variations due to environmental changes. We decode the resulting trace with traditional signal-processing techniques [33]. We first multiply the trace with a 90° and a 0° phase-shifted clock signals and we integrate over the two resulting signals ($\int$ blocks in Figure 10). The two resulting numbers are the real (*Re*) and imaginary (*Im*) parts of a representation of the bit in the complex plane $\mathbb{C}$. To classify each bit as a 1 or a 0 in this signal space, we use a naïve-Bayes classifier [27] with a kernel smoothing density estimate[4], previously trained on data from the same platform.

7.2. Performance Evaluation

To evaluate our transmission scheme, we encode several random messages onto clock signals at different frequencies and we use our source and sink app to transmit and record these messages on our two platforms, configured according to the reference setup of Section 5. We decode the temperature trace from each channel with our classifier; as the performance indicator, we use the error probability, as measured through the empirical bit error rate, i.e., the relative number of misclassified bits. We just report raw transmission rates and error probabilities and do not evaluate error correction strategies; we leave such study to future work.

Error probability at increasing rates. As a first test, we generate a 1000 bit and a 5000 bit message and we evaluate the error probability of our channels at increasing transmission rates, from 1 bps in 1 bps steps. For each channel, we use the 1000 bit message to train the classifier, which we evaluate on decoding the 5000 bit message. In a real attack, the source app could first transmit a known message that the sink app can use for training the classifier and then the actual information, which the sink app can decode with the trained classifier. Figure 11 shows the resulting error probability (measurements and bezier trends) for the four channels on our two platforms. For both *Laptop* (left) and *Smartphone* (right), the same-core channel shows very few errors ($\ll 1\%$) up to ≈ 40 bps; Figure 12 zooms in to this region. Up to this rate, *Smartphone* performs better than *Laptop*, thanks to the much lower noise.

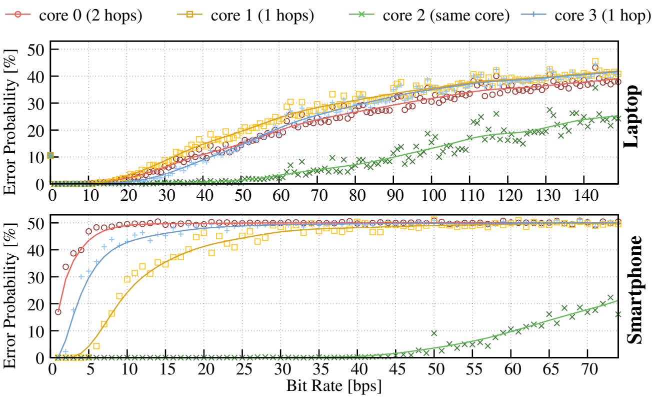

Figure 11: Error probability on decoding a 5000 bit random message for the four channels on *Laptop* (top) and *Smartphone* (bottom), for transmission rates up to 150 bps and 80 bps, respectively.

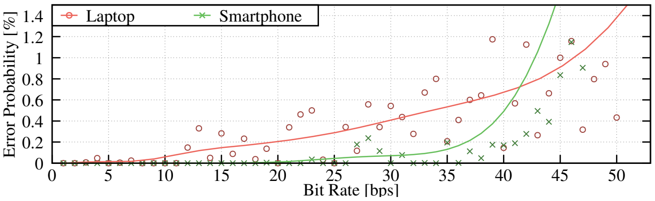

Figure 12: Zoom-in to the low-rate area of Figure 11; showing only the same-core channel.

At increased rates, errors increase more slowly on *Laptop*, where we achieve ≈ 90 bps at 10% error probability, than on *Smartphone*, where the rate is ≈ 60 bps at the same error level. *Laptop* shows better performance also for the 1-hop and 2-hop channels, where the error probability remains very close to 0 up to ≈ 10 bps and hits the 10% level between 30 bps and 40 bps. On *Laptop*, the 2-hop channel does not perform much worse than the 1-hop channels; instead, on *Smartphone* the error probability immediately increases steeply and the performance gaps are more evident, with the 2-hop channel showing several errors already at 1 bps. These results relate with the stronger quantization effect and the higher attenuation for these two channels on *Smartphone*.

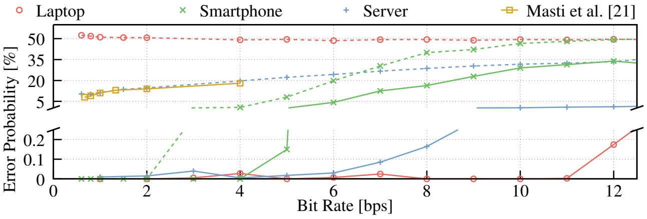

Figure 13: Direct comparison with Masti et al. [21, Tab. 1] for the 1-hop channel. The solid lines show the results with our scheme (see Section 7.1), the dashed lines show the results reported by Masti et al. [21] on *Server* and the results we obtained using their same scheme (ON-OFF keying).

While directly comparing these results with the capacity bounds of Figure 8 is not rigorous, since we are not con-

[4] We use the `NaiveBayes` object of Matlab R2015a, with default settings.

sidering the overhead of error correction, we can observe that *Smartphone* generally performs worse than *Laptop* when compared to the capacity bounds. In fact, for the capacity study, we hid the negative effects of quantization on *Smartphone* through longer experiments (see Section 6.2), while our transmission scheme is oblivious to this effect. A better transmission scheme for *Smartphone* might leverage temperature observations in the source app in order to tune the duty cycle of the clock signal so as to bring the average temperature at a quantization boundary, thus making the small 1 K variations visible and reducing the errors at high rates.

Direct comparison with previous work. To evaluate our transmission scheme against the naïve ON-OFF keying scheme used by Masti et al. [21], we provide a direct comparison. We both evaluate our scheme on the exact same platform they used[5] (we refer to it as *Server*) and implement the ON-OFF keying scheme in our framework, to evaluate it on *Server*, *Laptop* and *Smartphone*. Figure 13 shows all these results for the 1-hop channel; for all platforms, we plot the best of the two 1-hop channels. The solid lines show the results we obtained with our scheme on the three platforms (*Laptop*, *Smartphone*, and *Server*). The dashed lines show the results that we obtained on the three platforms by implementing the ON-OFF keying scheme with an edge detection decoder as described in Masti et al. [21] and additionally report the original results from this previous work [21, Tab. 1], which only covers a smaller range of bit rates. As Figure 13 shows, on *Server* our results with ON-OFF keying are very close to the original ones. Our scheme achieves 8 bps at about 0.1% error probability, while ON-OFF keying does not go below 8% error probability at 0.6 bps [21]. On *Laptop*, establishing a communication with ON-OFF keying proves virtually impossible due to the high level of noise, which hinders the edge detection algorithm; instead, our scheme proves more robust and obtains better results than on *Server*. On *Smartphone*, where there is almost no noise, ON-OFF keying matches the performance of our scheme for bit rates up to 2 bps and shows a slightly higher error probability for higher rates. From this extensive comparison, we conclude that our transmission scheme ensures much better performance than ON-OFF keying in all cases.

Spectral efficiency. To get a feeling of whether our scheme could be further improved, Figure 14 compares the input (green, top) and output (red, bottom) power spectra of the 5000 bit evaluation sequences at 5 bps and 80 bps with the ideal water-filling power allocation S_{xx} for the same-core channel on *Laptop*. The comparison is purely indicative, since the water-filling solution only gives an upper bound on the capacity of our channels (see Section 6.1), but is nonetheless interesting. On the one hand, the 5 bps input spectrum allocates much power at low frequency, resulting in very little distortion in the output spectrum in that area, which is where most information is encoded. On the other hand, the 80 bps input spectrum shifts most of the power at higher frequency, leading to visible distortion in the output spectrum due to the noise which, as Figure 6 shows, is stronger at lower frequencies. A better scheme should have a leveled input power allocation across the spectrum; finding such a scheme, despite the limitations on the input, is an interesting challenge for future work.

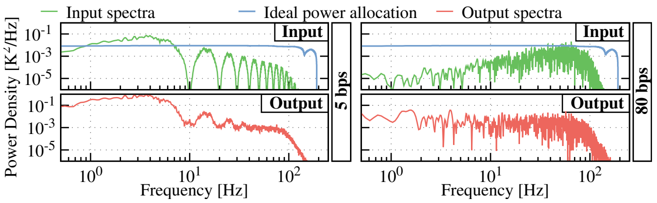

Figure 14: Input and output (same-core channel on *Laptop*) power spectra of the evaluation sequences at 5 bps and 80 bps, compared to the ideal water-filling power allocation.

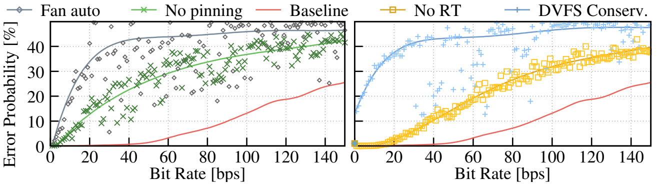

Figure 15: Sensitivity of the error probability to using automatic fan speed, not pinning the apps to cores, no real-time scheduling, or the conservative Linux DVFS governor.

7.3. Sensitivity to Environmental Conditions

Finally, we evaluate how variations in the environmental conditions affect the error probability on our channels. We identify four important parameters that, in a real attack, would not be fixed as in our experimental setup (Section 5) and we evaluate the sensitivity of our results to variations of these parameters. As a representative case, we show the results of this study on the same-core channel on *Laptop*.

Figure 15 shows how the error probability is affected when changing these four parameters in the experimental setup:

1. Setting the fan speed to automatic (Fan auto);
2. not pinning the apps to a specific core (No pinning);
3. using the default, Linux scheduling policy (`SCHED_OTHER`) instead of the high-priority `SCHED_FIFO` (No RT);
4. letting the conservative Linux DVFS governor change the frequency of the cores (DVFS Conserv.).

These four parameters have different impact on our baseline results, represented in Figure 15 by the solid red line.

Automatic fan speed. Using a variable, automatic fan speed highly affects the channel and makes the it very chaotic. This result is intuitive, as the fan controller is designed to keep the temperature on a low constant level, strongly hindering the possibility to encode data in temperature variations.

Conservative DVFS governor. Similarly to variable fan speed, enabling DVFS has a strong effect on the communica-

[5] A dual-socket server with two Intel Xeon E5-2690 multicores clocked at 2.90 GHz.

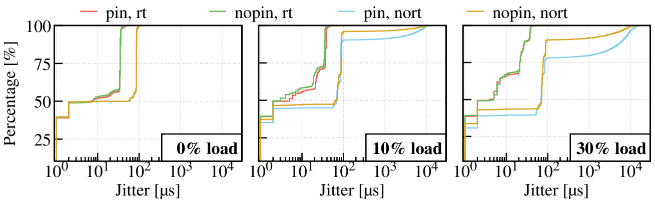

Figure 16: CDF of the transition jitter of the source app on *Laptop* with or without real-time scheduling ([no]rt) and thread pinning ([no]pin) and with different background load.

tion channel, which becomes highly unstable. This result is due to the fact that the active frequency of the cores largely determines the active power consumption, and thus temperature. Notice, however, that since on both platforms all active cores run at the same frequency, load-level based frequency scaling (which the Linux conservative governor implements) might enable another covert channel, where the sink app observes frequency variations induced by the source app. We plan to apply our methodology to study this channel in the future.

No real-time priority. Dropping real-time priority significantly affects the error probability only for rates faster than ≈ 15 bps. The additional errors are due to increased jitter in the timing of the source and sink apps; Figure 16 further investigates this effect by analyzing the jitter in the state transitions of the source app when running a 100s random trace with our baseline setup (pin, rt) and when dropping real-time priority (nort) or thread pinning (nopin). We repeat the experiments with different levels of system load, which we simulate by pinning one additional source app to each core, each running a different random execution trace with the appropriate duty-cycle. At low load, dropping real-time priority causes the jitter to increase to $\approx 100\,\mu$s in $\approx 50\%$ of the transitions; the sink app is similarly affected in the precision of its sampling rate. Figure 15 shows that this effect only starts impairing the performance of our scheme at rates faster than ≈ 15 bps. The jitter is higher at increased load, but it does not exceed 1 ms for 90% of the transitions at 30% load for the nopin, nort case; thus, error correction should still enable communication at low rates even with system load. Finally, we note that, with increasing load, not pinning the source app to a core (nopin in Figure 16) leads to reduced jitter, thanks to smart core migrations by the Linux scheduler.

No thread pinning. When the source and sink apps are not pinned to a specific core, the different channels effectively move with the source app. As an example, Figure 17 shows part of a trace from *Smartphone* where the source app, which is transmitting a 1 Hz clock signal, migrates between cores 1 and 2. Initially, reading the temperature from core 2 corresponds to a same-core channel, while it becomes a 1-hop channel at time ≈ 1.5 s, when the source app migrates to core 1. As Figure 18 shows, if the sink app always observes the same core (core 2 on *Laptop* in this case), the error probability without thread pinning will sensibly increase compared to the baseline, since the channel type keeps changing. However, there is a simple way to work around this issue. Since the sink app can always read the temperature of all the cores, we can simply look at the *all-cores channel*, which is the sum of the temperatures from all cores. As Figure 18 shows, the all-cores channel has performance comparable (or possibly better) than the same-core channel.

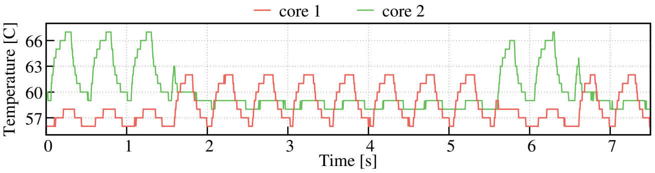

Figure 17: Traces from cores 1 and 2 of *Smartphone*; the source app is not pinned.

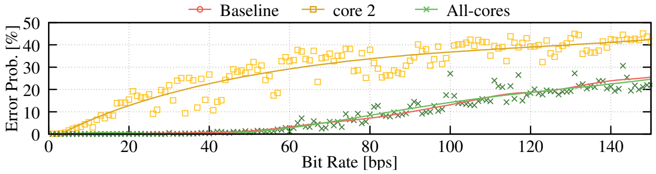

Figure 18: Same-core vs. all-cores channel comparison with no pinning on *Laptop*.

We conclude that our communication scheme is robust to disabling thread pinning and, to some extent, to dropping real-time priorities and having background system load. The most sensitive parameters are varying fan speed and enabling the DVFS governor, which makes communication impossible with our scheme but might enable a different covert channel when all cores share the same active frequency.

8. CONCLUDING REMARKS

In this paper, we analyzed a family of covert channels where a source app induces temperature variations on a multicore processor and the sink app observes these changes through the on-chip temperature sensors.

Summary and takeaways. Our two main contributions with this paper are providing upper bounds on the capacity of these channels and showing a transmission scheme that improves previous results on communication rates by more than 20×. Based on experimental data from two diverse platforms representative of laptops and smartphones, we derived capacity bounds by leveraging information theory and spectral analysis. Based on our results, we cannot exclude the possibility that these channels might be a security issue, as the capacity could be in the order of 300 bps for the same-core channel. We presented a transmission scheme based on Manchester encoding that sensibly improves the performance of previous work and we studied the sensitivity of our results to non-ideal conditions. With this scheme, we were able to achieve rates of more than 45 bps on the same-core channel and more than 5 bps on the 1-hop channel, with less than 1% error probability.

Threat mitigation. As we reported in Sections 1 and 3, the on-chip temperature sensors that enable the thermal covert channels we studied are easily accessible by user-level apps on current mobile systems. A technically simple way to block the potential threats coming from these channels is to restrict access to the temperature sensors to trusted code. If temperature information needs to be made available to user apps (e.g., a CPU temperature monitor), viable mitigation strategies include increasing the refresh interval from milliseconds to seconds or minutes and reducing the sensor resolution, thus directly limiting the capacity of the thermal covert channels. While mitigating this threat is not technically challenging, it requires shipping security patches to a huge base of affected devices running different versions of different system software stacks. Our aim with this paper was building awareness on the potential threat that current systems are exposed to and providing a quantitative study that can be used as a base to decide what actions to take in order to mitigate this threat.

Directions for future work. The methodology based on spectral analysis that we devised in order to estimate the channel capacity (Section 6) introduces a new way to quantify the potential threat of complex covert channels, which are often only analyzed from an empirical standpoint. We are planning to exploit this same methodology to analyze other covert channels and we hope that others will find it a useful guideline. Sensible avenues for further research focusing on the thermal covert channels, include reporting results of real attacks (e.g., leaking an encryption key) using these channels on real systems, analyzing the impact of error-correction schemes on the achievable rates, and finding more efficient communication schemes to reduce the gap between the capacity estimations and achieved rates.

ACKNOWLEDGEMENTS

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 Lukas Sigrist, Georgia Giannopoulou and Rehan Ahmed for their help with polishing the paper.

REFERENCES

[1] G. Andria, M. Savino, and A. Trotta. Windows and interpolation algorithms to improve electrical measurement accuracy. *IEEE Transactions on Instrumentation and Measurement*, 38(4):856–863, Aug 1989.

[2] D. B. Bartolini. *Techniques and Tools for Efficient, QoS-Driven Warehouse-Scale Computing*. Tesi di Dottorato (PhD Thesis), Politecnico di Milano, 2015.

[3] E. O. Brigham. *The Fast Fourier Transform and Its Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[4] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 171–, 2001.

[5] J. Brouchier, N. Dabbous, T. Kean, C. Marsh, and D. Naccache. Thermocommunication. Cryptology ePrint Archive, Report 2009/002, 2009.

[6] J. Brouchier, T. Kean, C. Marsh, and D. Naccache. Temperature Attacks. *Security Privacy, IEEE*, 7(2):79–82, 2009.

[7] T. M. Cover and J. A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006. ISBN 0471241954.

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 365–376, 2011.

[9] D. Evtyushkin, 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.

[10] M. Guri, M. Monitz, Y. Mirski, and Y. Elovici. BitWhisper: Covert Signaling Channel between Air-Gapped Computers using Thermal Manipulations. URL http://arxiv.org/abs/1503.07919.

[11] J. Hasan, A. Jalote, T. Vijaykumar, and C. Brodley. Heat stroke: power-density-based denial of service in SMT. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 166–177, 2005.

[12] C. Heegard and L. Ozarow. Bounding the capacity of saturation recording: the Lorentz model and applications. *Selected Areas in Communications, IEEE Journal on*, 10(1):145–156, Jan 1992. ISSN 0733-8716.

[13] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[14] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari. Understanding contention-based channels and using them for defense. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, pages 639–650, 2015.

[15] M. Hutter and J.-M. Schmidt. The Temperature Side Channel and Heating Fault Attacks. *IACR Cryptology ePrint Archive*, 2014:190, 2014.

[16] T. Iakymchuk, M. Nikodem, and K. Kepa. Temperature-based covert channel in FPGA systems. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–7, June 2011. doi: 10.1109/ReCoSoC.2011.5981510.

[17] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manuals volume 3: System programming guide*, 2015.

[18] B. W. Lampson. A Note on the Confinement Problem. *Commun. ACM*, 16:613–615, Oct. 1973.

[19] C. H. Lim, W. R. Daasch, and G. Cai. A thermal-aware superscalar microprocessor. In *Quality Electronic Design, 2002. Proceedings. International Symposium on*, pages 517–522. IEEE, 2002.

[20] C. Marsh and D. McLaren. Poster: Temperature Side Channels. In *In the Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), 2007*, 2007.

[21] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. Thermal Covert Channels on Multi-core Platforms. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 865–880, Washington, D.C., Aug. 2015. USENIX Association. ISBN 978-1-931971-232. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti.

[22] A. Mileva and B. Panajotov. Covert channels in TCP/IP protocol stack - extended version-. *Central European Journal of Computer Science*, 4(2):45–66, 2014. ISSN 1896-1533.

[23] 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.

[24] D. Rai, H. Yang, I. Bacivarov, and L. Thiele. Power Agnostic Technique for Efficient Temperature Estimation of Multicore Embedded Systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, 2012.

[25] C. Reis, A. Barth, and C. Pizano. Browser Security: Lessons from Google Chrome. *ACM Queue*, 7(5):3, 2009.

[26] 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.

[27] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, second edition, 2003.

[28] C. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[29] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 17–, 2002.

[30] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory Deduplication As a Threat to the Guest OS. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 1:1–1:6, 2011.

[31] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[32] U.S. Department of Defense. *DOD Trusted Computer System Evaluation Criteria "The Orange Book" [DOD 5200.28]*. National Computer Security Center, 1985.

[33] P. P. Vaidyanathan, S.-M. Phoong, and Y.-P. Lin. *Signal Processing and Optimization for Transceiver Systems*. Cambridge University Press, 2010. ISBN 9781139042741. URL http://dx.doi.org/10.1017/CBO9781139042741. Cambridge Books Online.

[34] Z. Wang and R. Lee. Covert and Side Channels Due to Processor Architecture. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 473–482, 2006.

[35] P. D. Welch. The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electroacoustics*, 15(2):70–73, Jun 1967.

[36] 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.

[37] 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.

[38] S. Zander and S. J. Murdoch. An Improved Clock-skew Measurement Technique for Revealing Hidden Services. In *USENIX Security Symposium*, pages 211–226, 2008.

[39] S. Zander, P. Branch, and G. Armitage. Capacity of Temperature-Based Covert Channels. *Communications Letters, IEEE*, 15(1):82–84, 2011.