UnCovert: Evaluating thermal covert channels on Android systems

Pascal Wild

August 5, 2016

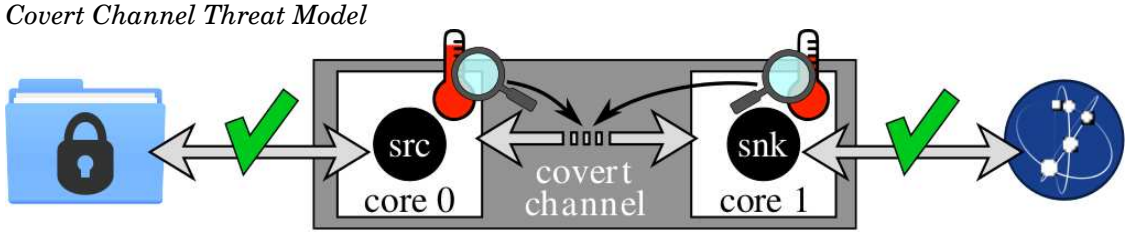
Contents

Figures

Introduction

Todays computing devices contain thousands of semiconductors which generate heat during operation. Therefore temperature management is an important task for system developers to prevent irreparable damage done to the hardware and to extend the lifetime of computing devices. To measure device temperature, modern systems contain various sensors on components like battery, central processing unit (CPU) and graphics processing unit (GPU). The linux kernel exposes sensor data to userspace via the virtual file system, allowing applications to do power management and control device temperature. Sensor information is resource, which poses a security threat, e.g. for covert channels. Covert channels are a medium for a computer security attack where two isolated applications which are not allowed to communicate, set up a communication channel. Covert channels are hidden from the access control of operating system and are therefore also hard to detect. In figure 0.1 the threat model for such an attack is shown. On the left side is an application called source, which is isolated on core 1 and has access to sensitive system data. The sink application, isolated on core 2, has access to the Internet. If the two applications collude, they can set up a covert channel in between. This can be achieved, if the source application controls the core temperature in a way to encode data into temperature trace. The sink application can log the temperature of core 1 and send data over the Internet to an attacker, which then can do the data analysis off-line and read data out of the temperature trace. Previous work of Bartolini et al. [1] has shown that communication over this channel is possible, achieving a throughput of 70 bits per second with a bit error rate (BER) of 20% on Ubuntu platforms. The goal of this work is to port the existing framework to Android and explore the thermal covert channel on a Samsung Galaxy S5 smartphone.

Figure 0-1
Covert Channel Threat Model



1

Framework

The existing framework provided by Bartolini et al. [1] can be divided into three parts, the source application generating the load on the cores, the sink application which is responsible for the temperature measurement and a coordinator application that synchronizes. The source application is implemented in C++, sink in C and the launcher is a BASH script.

1.1 Source

The source application is responsible to control the temperature of one or multiple cores. It takes an input file where the bitstream is defined, that should be transmitted over the covert channel. Depending on this schedule file, the application either heats up the core if it wants to transmit a one or it is idle when transmitting a zero, letting the core cool down. The heating of the core is achieved by running a loop with expensive tasks.

1.2 Sink

The sink application logs the temperature on the cores with a timestamp to a file. It achieves that by accessing the sensor information of the system through the virtual file system. The timestamps are generated by the function `gettimeofday()`.

1.3 Launcher

The launcher script initializes sink and source with the defined experiment arguments. It coordinates sink and source to prevent de-synchronization of the covert channel.

2

Android

Previous work was done on Ubuntu platforms which differs a lot from the Android Operating System. Android is intended for mobile devices which means that it enforces stricter rules for power management and is not intended for server-like, long-running background operations like the Ubuntu Operating System. Additionally, Android has security policies, which also impeded the porting process.

2.1 Application Sandboxing

Android enforces Application Sandboxing to isolate app data and code execution from other applications[2]. Android assigns different User IDs for every application. Communication across two different user IDs is not possible.

2.2 Development

To port the framework to Android, several components have been used which will be explained more in detail in this chapter.

2.2.1 Android Software Development Kit

The foundation of every Android application can be built with the Android Software Development Kit (SDK). In this work, the basis of the applications is therefore also constructed with the SDK.

2.2.1.1 App Components

Every Android application consists of one or multiple elements that are called App Components[3]. These include:

- Activities that represents a single screen with a user interface
- Services that run in the background intended to perform long-running operations without providing a user interface

- Content Providers that manage a shared set of application data, intended for database operations
- Broadcast Receivers that respond to to system-wide information e.g. low battery

It was important for the porting process to make a good decision between.

2.2.1.2 Intents

As UNIX signaling was not an option for inter-process communication on Android because of Application Sandboxing, other options had to be explored. Android SDK implements Intents as a choice for communication between applications. They are mainly intended to launch applications or communicate with a background service. In this work, they were used to start, pass arguments and to control the program flow, allowing synchronization between.

2.2.1.3 Thread Listener

Android offers the Listener class to create an interface between a thread and a forked thread. To avoid blocking of a main thread while waiting for a forked thread, the implementation of a Listener is required. The main thread can then stay idle while the forked thread is doing work and react on events that are sent through the Listener, e.g. when the forked process has finished.

2.2.2 Native Development Kit

The Android NDK is a set of tools allowing the developer to embed C or C++ ("native code") into Android apps. The ability to use native code in Android apps can be particularly useful to developers who wish to do one or more of the following:

- Port their apps between platforms.
- Reuse existing libraries, or provide their own libraries for reuse.
- Increase performance in certain cases, particularly for computationally intensive.

Because the existing framework provided by Bartolini et al. [1] is written in C and C++, it made sense to use the NDK to speed up the porting process and to get similar results to previous work. Additionally, problems that were encountered during experiments were more likely caused by the Android OS then by the implementation, because the core of the apps is the same like on the Ubuntu platform. This fact simplified the debugging process.

2.2.3 Java Native Interface

The Java Native Interface (JNI) enables programmers to write native methods to handle situations when an application cannot be written entirely in the Java programming language, e.g. when the standard Java class library does not support

the platform-specific features or program library. It is also used to modify an existing application—written in another programming language to be accessible to Java applications. Many of the standard library classes depend on JNI to provide functionality to the developer and the user, e.g. file I/O and sound capabilities. Including performance- and platform-sensitive API implementations in the standard library allows all Java applications to access this functionality in a safe and platform-independent manner. The JNI framework lets a native method use Java objects in the same way that Java code uses these objects. A native method can create Java objects and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code[4]. In this work the JNI was used to pass arguments and call C/C++ code.

2.2.4 Android Debug Bridge

Android Debug Bridge (adb) is a versatile command line tool that lets you communicate with an emulator instance or connected Android-powered device [5]. In this work, it was mainly used for debugging the applications, later for starting experiments and transferring data form the Android device to the developer computer.

2.3 Power Manager

The Power Manager on Android tries to make efficient use of the energy saved in the battery to ensure long lifetime of the device without power supply. This is very different to the Ubuntu OS, which is intended for stationary devices plugged to a power supply. To ensure good performance of the applications for the covert channel, it was important to deal with the power saving options, enforced by the Power Manager.

2.3.1 Wake Lock

To avoid drainage of the battery, an Android device that is left idle, quickly falls asleep and puts the CPU and the screen in a sleep state. However for the sink and source applications, it is important to keep the CPU running. The `PowerManager` application programming interface (API) offers a feature called Wake Lock to prevent this.

2.4 Android Scheduler

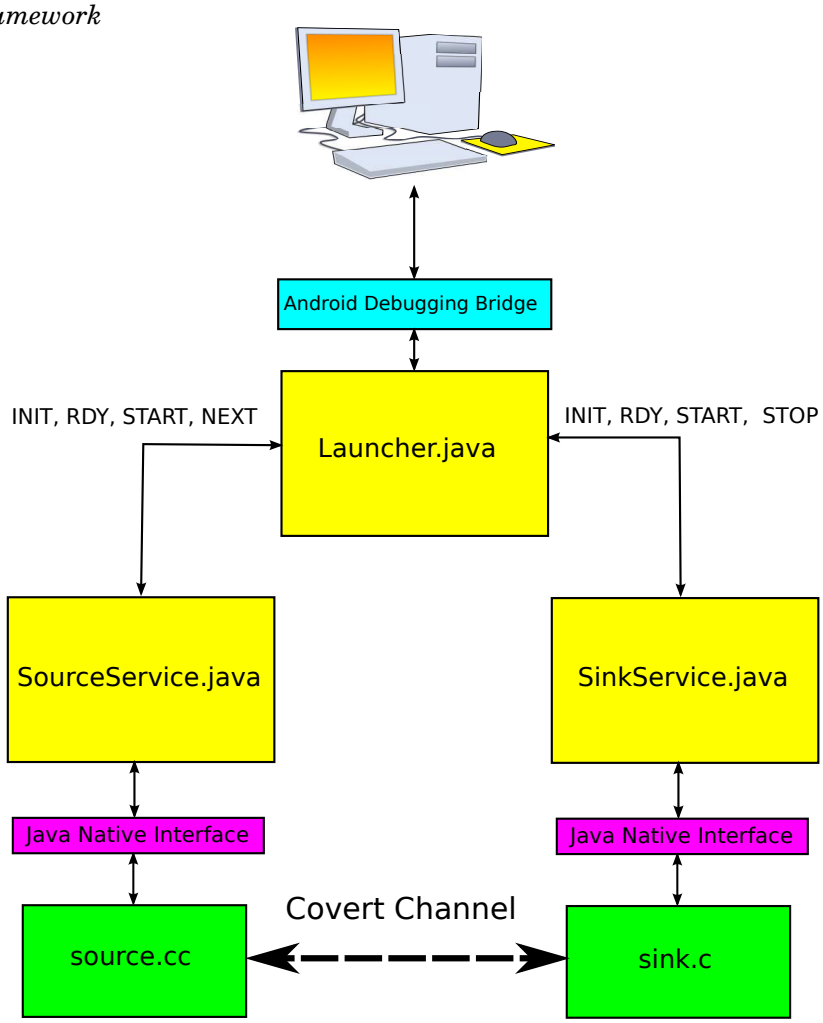
The Android scheduler distinguishes between foreground and background processes. Applications that are e.g. implemented as activities run as a foreground process with high priority to ensure instant reaction of the device if the user interacts with it. On the other side, for example services, that run in the background and do not offer interaction with the user, are scheduled as background processes which means lower priority in execution. Applications can be explicitly started in foreground with the `startForeground()` API. In this work, elevating the processes to foreground was important to increase lifetime and performance of the applications.

3

Implementation

In this chapter, the built Android framework is explained and differences to the existing framework are stated. In particular the three components Launcher, which coordinates and starts sink and source. SinkService that logs system temperature trace and SourceService that deploys load on core(s). All applications acquire a Wake Lock and are put into foreground using the `startForeground` API. In figure 3 the experiment flow is shown which is also explained in detail in the further sections.

Figure 3-1
Android Framework



3.1 SourceService

The source is implemented as a service that can be started and controlled via Intents. Native code in C++ form the previous work on Ubuntu is underlying the service and can be called through the JNI. To start this service, an intent has to be sent with the experiment arguments. In particular it needs following information contained in the Intent:

- Location of the schedule file in the file system
- Location of the logging file
- Core(s), where the load should be deployed

It then calls the native code in a forked thread and adds a Listener to the thread, which is able to call back, when the native thread has finished execution. Therefore the main thread is not blocked and the app does not get killed because of an app not

responsive (ANR) dialog. When initialization of the native thread has finished, the Service sends an Intent back to the Launcher application, that it is ready to start the load. The service stays idle until it gets another Intent, which tells it to start the load. After the load in the native code has finished, the service sends an Intent back to the Launcher application, that the schedule file has been done, kills the service and clears the cache to restore the default state.

3.2 SinkService

The sink is also implemented as a service and has a similar flow to the SourceService. The core temperature measurement takes place in native C code underlying the service. It is also started through an Intent, providing following parameters:

- Location of the logging file
- Core, where the native thread should be pinned on
- Sampling period for the logging

Again, after the native thread has initialized, it sends an Intent back to the Launcher application to inform it that it is ready for measurement. It stays idle until it gets another Intent with the information to start measurement. The native thread keeps logging until the service gets an Intent, informing it that it should stop the measurement. After the log has been dumped to a file, the service sends an Intent back to the Launcher app, that measurement has finished, clears its cache and kills itself. Again to restore the default state of the application.

3.3 Launcher

The Launcher application starts experiments and synchronizes sink and source. It can be started with an Intent that is sent through the ADB from a developer computer to the Android device. It does not need any data from the Intent. The application pre processes every experiment that is contained in the `/sdcard/uncovert` folder. It extracts every schedule file that is found and saves the paths in a list of strings. In addition, it reads the setting file in `/sdcard/uncovert` that is pushed through adb from the developer computer to the device, to set the parameters for SinkService and SourceService. It prepares the Intents for the sink and the source with the information that is needed and sends those Intents, to initialize sink and source. It then waits at a barrier until the Intents from the services arrive with the information that they are initialized. After surpassing the barrier, the application sends Intents to Sink and SourceService to start the covert channel. It stays idle until it receives an Intent from the SinkService, that measurement has finished. Then it continues with the next schedule file until the list of schedules is done. After that it clears its cache and kills itself to restore default state.

4

Results

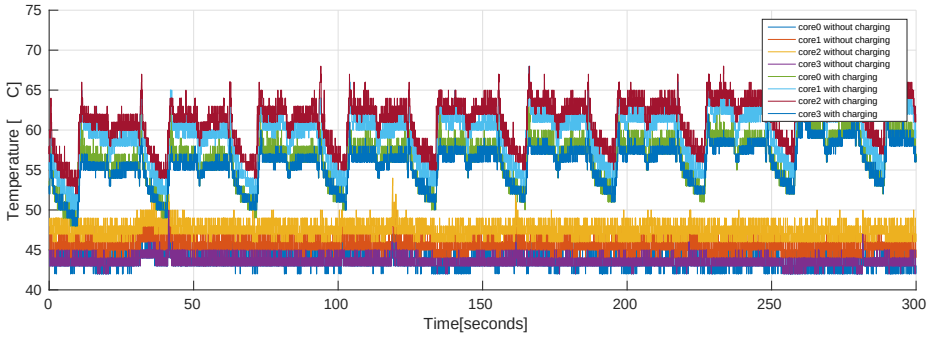
During experiment runs, several problems were encountered which are shown in this chapter. Most results are negative but still show the dynamics of the covert channel on Android and differences to results on previous Ubuntu platforms.

4.1 Experimental Setup

The experiments were conducted on a Samsung Galaxy S5 with a Exynos 5422 Octa chipset, mostly in a controlled, air-conditioned environment.

4.2 Battery Charging

Figure 4-1
Battery Charging Effect

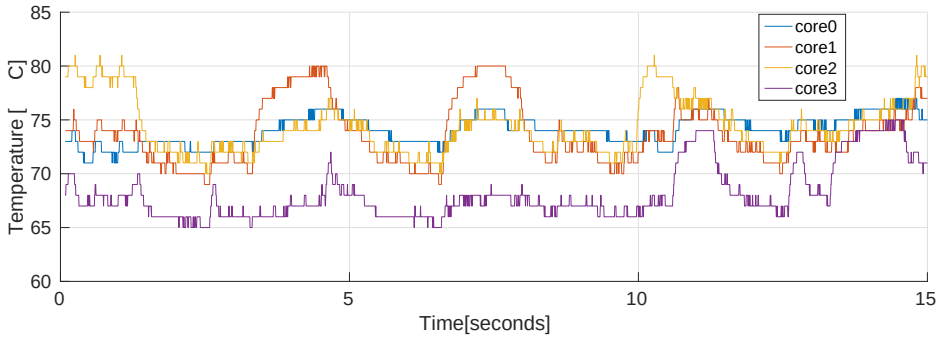


Some experiment runs showed weird interferences with the data signal. To find out where those interferences came from, experiments without load were conducted on the device. Figure 4.2 shows two separate runs, the upper four traces show the temperature trace of the four big cores from a run where the battery of device was charged and still connected to the power supply. The lower four traces show the traces of the cores but without. While the temperature stays constant for the lower

curves without connection to power supply, the upper traces with power supply show a cyclic rise and fall in temperature. This result can be interpreted as cyclic charging that is done by the battery chip to prevent battery damage. In this work, the symbol rate is very low, therefore the period of the charging cycle and the symbol duration are in a similar order of magnitude. This means that interferences can impair data transmission.

4.3 Hardware Throttling

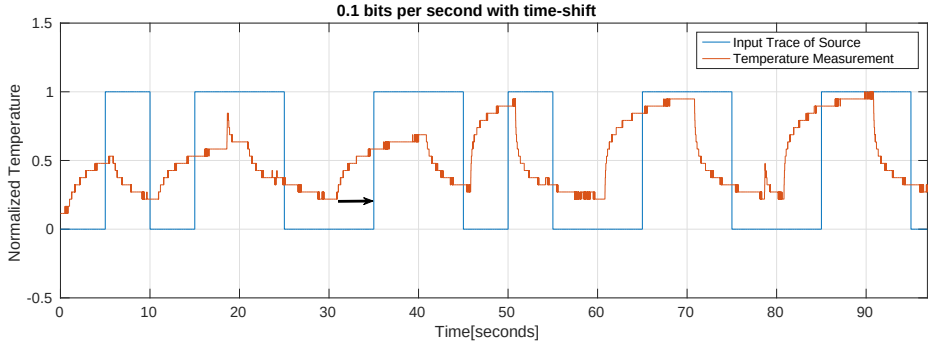
Figure 4-2
Hardware Throttling



While doing experiments outside of the controlled environment, another problem was encountered. Figure 4.3 shows an experiment run where the source was supposed to run and generate heat for fifteen seconds. The result does not show a constant raise in temperature but a throttling behaviour at 80 ° C until the cores cool down to about 70 ° C. In this state of development, the SourceService was not pinned on a specific core and core migration between core2 and core1 is also visible.

4.4 Time-shift

Figure 4-3
Time-shifted signal

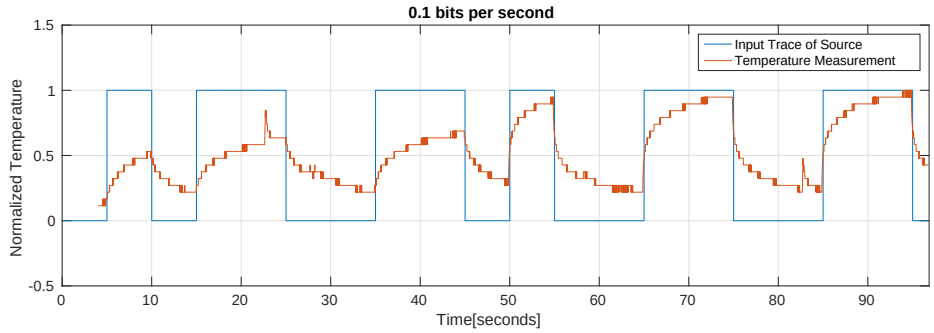


All experiments showed a time-shift in the data signal. In figure 4.4 the results of a run on 0.1 bits per second is shown. The blue trace shows the bitstream that the

SourceService should transmit. The red trace shows the measured trace on the same core where the source is running. The offset varied between different experiments but it stayed constant for a particular schedule file. This was probably caused by a design flaw of the native code. While in the framework on Ubuntu platforms, synchronization of sink and source was done via UNIX signaling, it was changed to Intents for the Android device. In the Android implementation the variable of the loop is changed with Intent through the JNI, which means that the `sleep(10)` function finishes and in the worst case, causes a time difference of 10 seconds between sink and source.

4.5 Time-shift corrected Signal

Figure 4-4
Corrected time-shifted signal



As the time-shift was constant for a particular schedule file, it was possible to manually correct the time-shift. In figure 4.5 the same run like in figure 4.4 with the eliminated time-shift is shown. Whenever the utilization trace is high, temperature is rising and if the utilization trace is low, the temperature is falling.

5

Conclusions and Further Work

Like shown in figure 4.5 the covert channel is definitely exploitable on Android devices. In this work much lower bit rates were achieved then in previous work of Bartolini et al. [1] on Ubuntu platforms. The hardware configuration of the Samsung Galaxy S5 differs a lot from the Odroid, it has a battery, a case and no fan. This changes the dynamics of the thermal covert channel. A case study should be designed to find out about the real capabilities of the channel on Android devices to confirm the security threat. For further experiments, the synchronization between source and sink should be improved. Either by modifying the existing data processing framework so that it can deal with a time shift. One option would be to implement UNIX signaling as process synchronization, which requires root privileges because of the Android Application Sandboxing. Another solution would be to use Android Binders as inter-process communication, where one has to design a direct interface between the Sink and SourceService to allow synchronization. Both solutions would imply direct communication between sink and source which does not make it interesting for a real attack, but to explore the channel on Android. It would be even more interesting, to find a synchronization scheme directly done on the covert channel between sink and source to remove the necessity of the Launcher application.For further work, the SourceService and the SinkService could be embedded into real applications. The source could be embedded into an application with file system access where it has access to sensitive data, for example a file explorer app. The sink into an app with Internet access to allow transmission over the web to an attacker, for example a browser application. This would make the framework even nearer to a real attack and would definitely confirm the security threat of such a covert channel, if data transmission is still possible.

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*.

[2] Android security tips. URL `https://developer.android.com/training/articles/secu`

[3] Android application fundamentals. URL `https://developer.android.com/guide/compon`

[4] Java native interface. URL `https://en.wikipedia.org/wiki/Java_Native_Interface`.

[5] Android debug bridge. URL `https://developer.android.com/studio/command-line/ad`