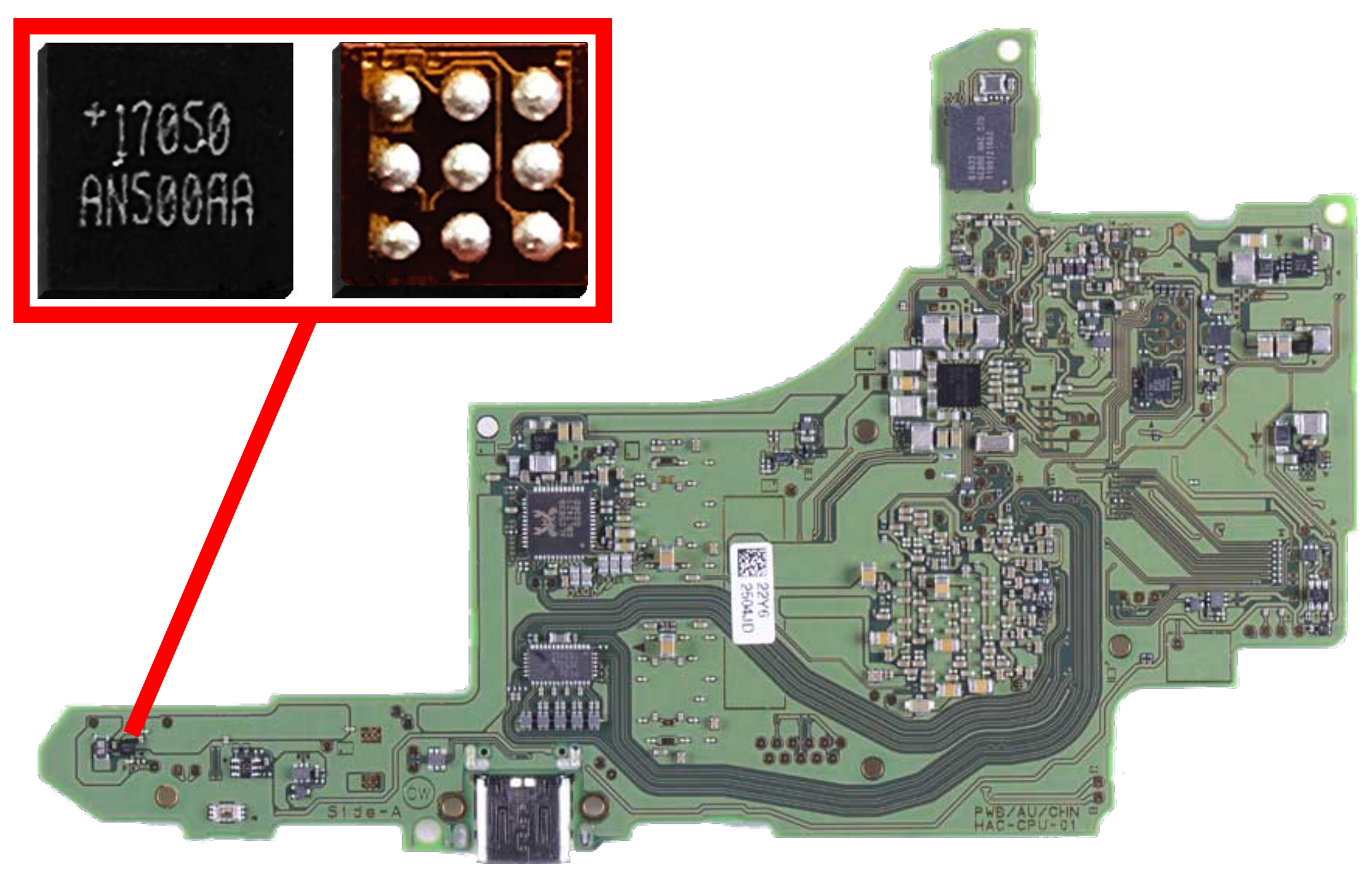
Bypassing Android isolation with fuel gauges: new risks with advanced power ICs

Vincent Giraud   David Naccache

DIÉNS, ÉNS, CNRS, PSL University, Paris, France   Ingenico



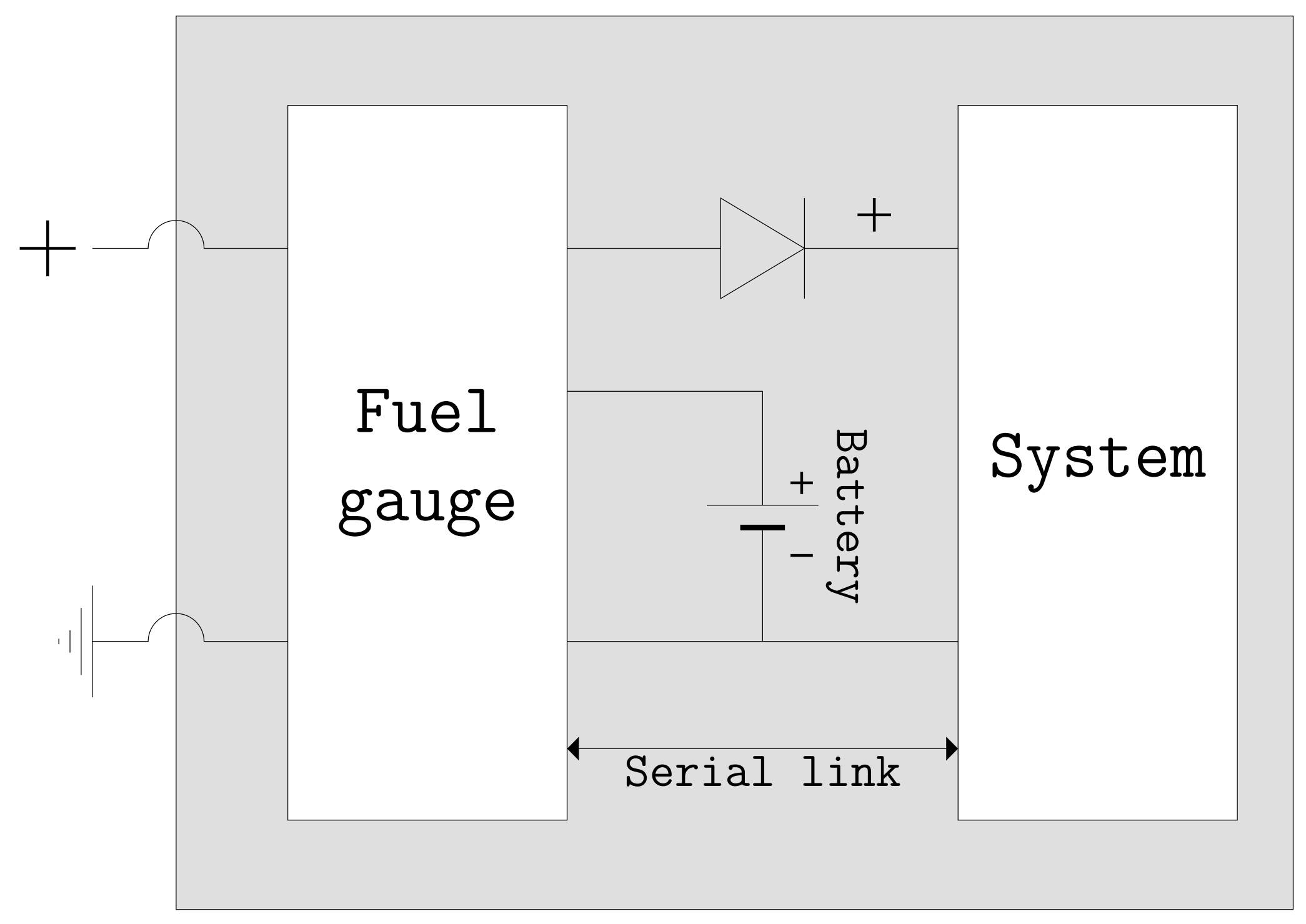
One of the boards in the Nintendo Switch. This device contains a fuel gauge, the MAX17050, to which we provide a magnified view.

The challenge of managing batteries

Lithium batteries offer many advantages for embedded devices. However, predicting and analyzing their behavior is hard, as it depends on many factors: the **voltage** at their terminals, the operating **temperature**, their **age**, the **lithium quality**, the **extracted load** since the last full charge…

To facilitate power management, many designers incorporate a fuel gauge in their embedded devices. It is an integrated circuit dedicated to analyzing and monitoring various metrics related to the energy source. They include the instantaneous current going in or out of the battery, along with the other aforementioned metrics.

However, an overlooked aspect of these components that is worthy of interest in the field of security is their remarkable accuracy. Can it pose a threat?

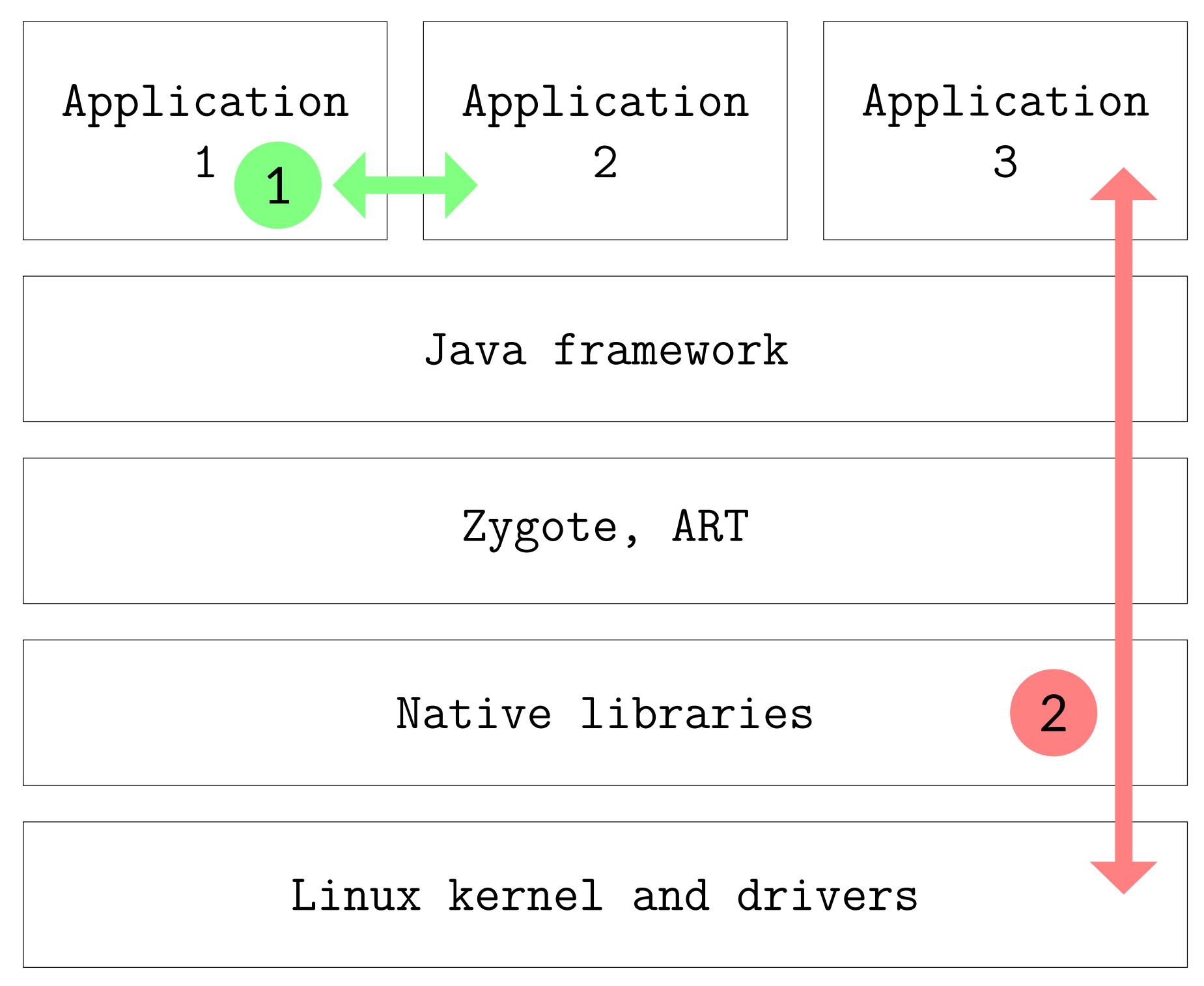


Typical integration of a fuel gauge in an embedded system. It is placed between the system and the power sources, so that all consumed current have to flow through it.

Integration in Android

While the Android security policy is clear and explicit regarding interactions between applications 1, it is less the case concerning interactions with the hardware 2, as it can vary over the type of requested resource, the type of access, or the operating system version, for example. Smartphones and tablets constructors can also further complicate their moderation rules.

We discovered that despite the potential hazards implied by fuel gauges, no moderation or security policy enforcement can be found in any of the abstraction layer in Android, especially in the Java framework and at the kernel level. The former is usually where all of these considerations are implemented, the latter features SELinux, a security module perfectly able to restrict abuses on hardware resources.



Layered representation of the Android operating system. To access hardware resources, requests from applications at the top must be relayed by all of the underlying components.

Identified risks

Given the lack of security policy concerning fuel gauge accesses, we identified three risks:

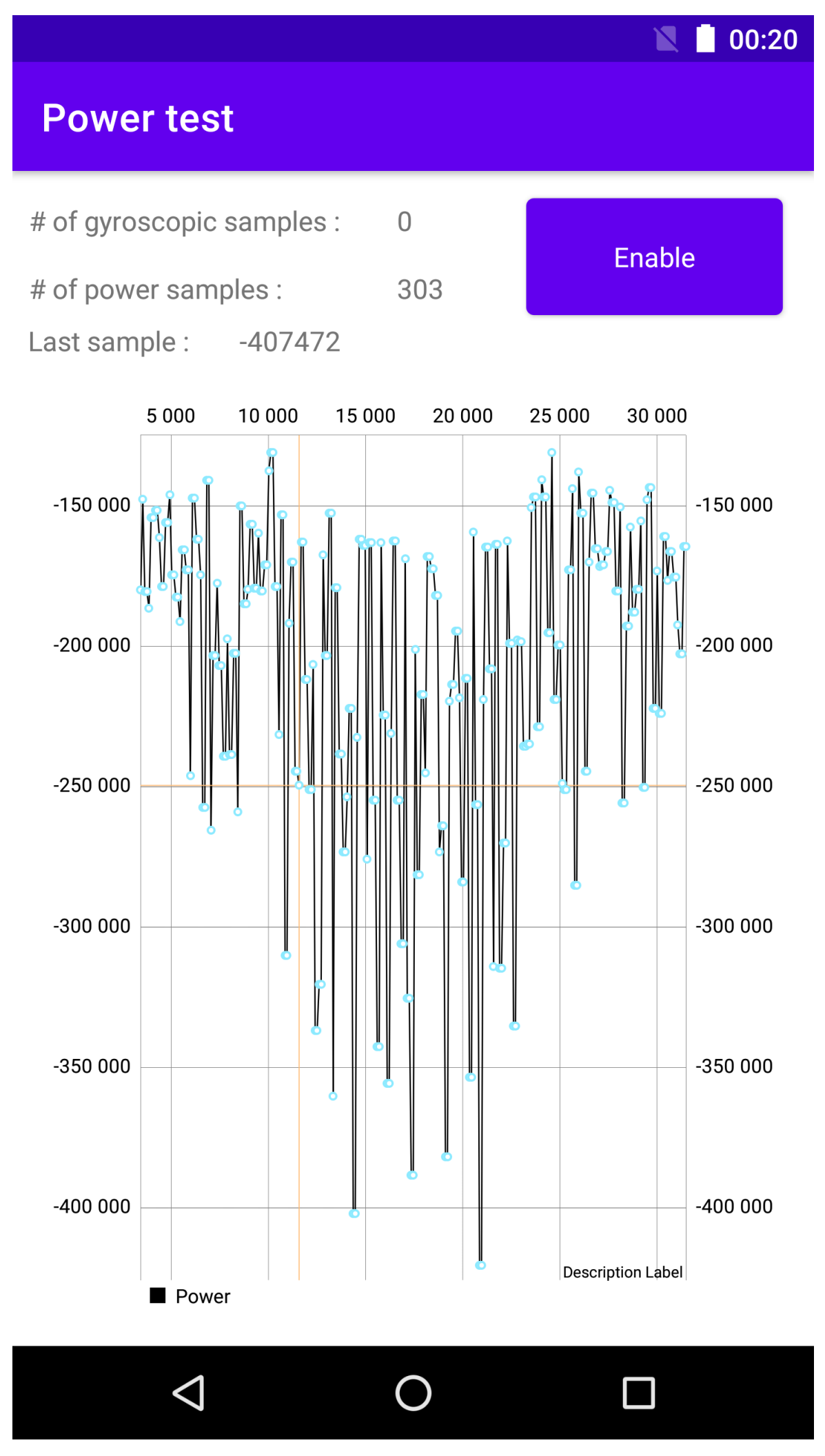
1. A privacy risk: a malicious application could monitor the consumption all day long without the user's consent, and log the activity. A lot of sensitive information could be deduced.
2. The risk of creating a hidden communication channel on the platform: one malicious application that has exclusive access to some data could secretly get it out by voluntarily causing more power consumption and modulating a signal on the real-time current measurement. Another application would simply have to read this signal by accessing the fuel gauge.
3. The risk of having a legitimate application being spied on by a malicious one, to get sensitive data. In these works, we focused on targeting a process ongoing at human speed: a PIN code entry.

Proof of concept: PIN code entry spying

Are fuel gauges precise enough and are their refresh rates high enough to spy a PIN code entry? To verify it, we developed an application probing these integrated circuits, and tested it on smartphones in Google's Nexus and Pixel lines. We confirmed that:

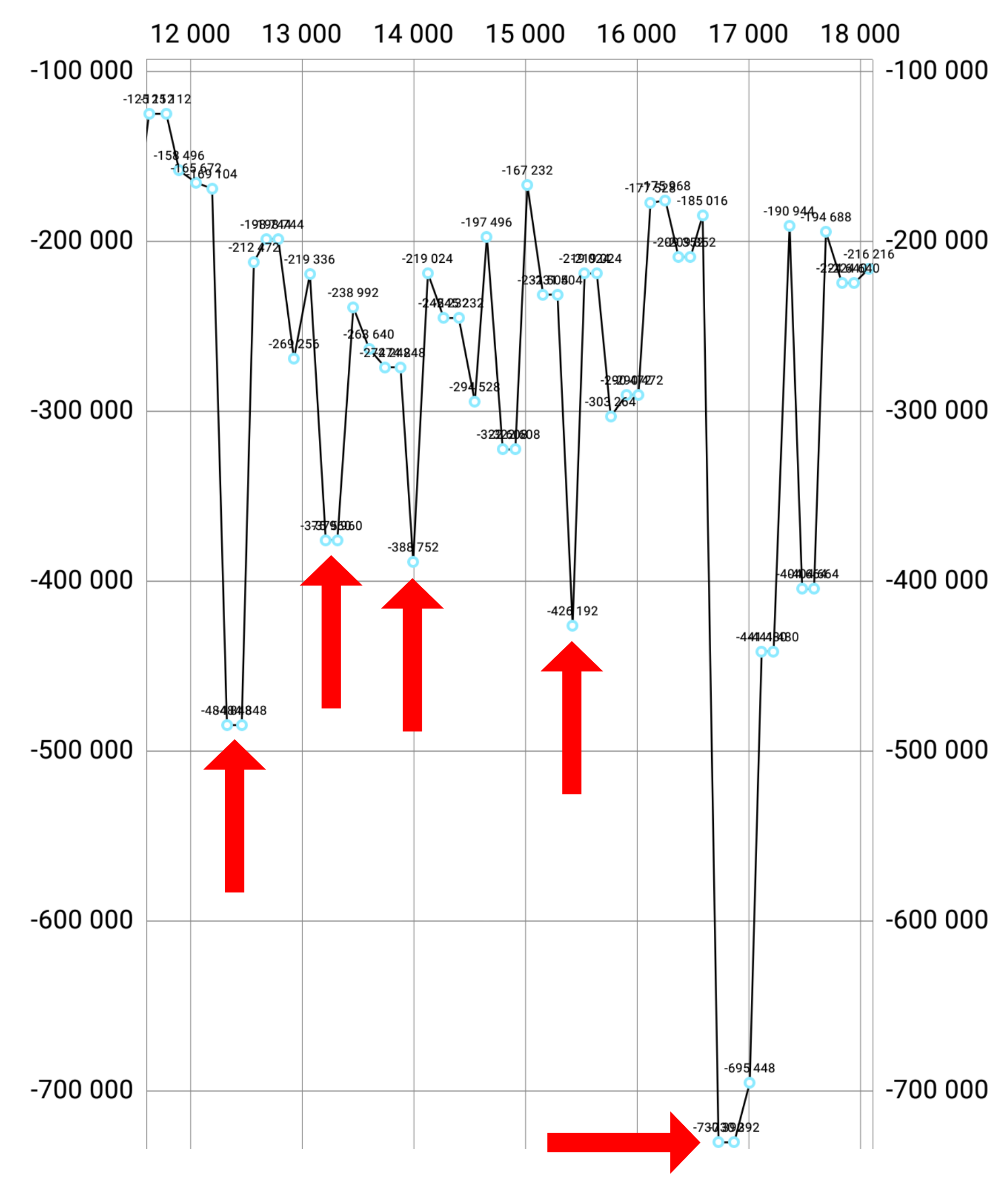
- their refresh rates are high enough to spy on processes at human speed, like a PIN code entry. However, they are too low to target any software-only process, including cryptographic implementations. This limitation is not induced by Android, which does not cap the request frequency, but by the refresh rates of the physical registers in the fuel gauges.
- they are precise enough to detect a touch on smartphones' screen. The overconsumption happening during a touch can have two reasons: the physical functioning of the touchscreen, and the software processing of the components responsible for the management of the input, notably the corresponding driver.

By observing the real-time current consumption trace obtained during a PIN code entry, we can notice downward peaks corresponding to key presses, without needing to do signal analysis. Attacks on these processes are thus possible. The platform does not need to be rooted: Android, in its default configuration, provides enough permissions for a malicious application to be able to spy.

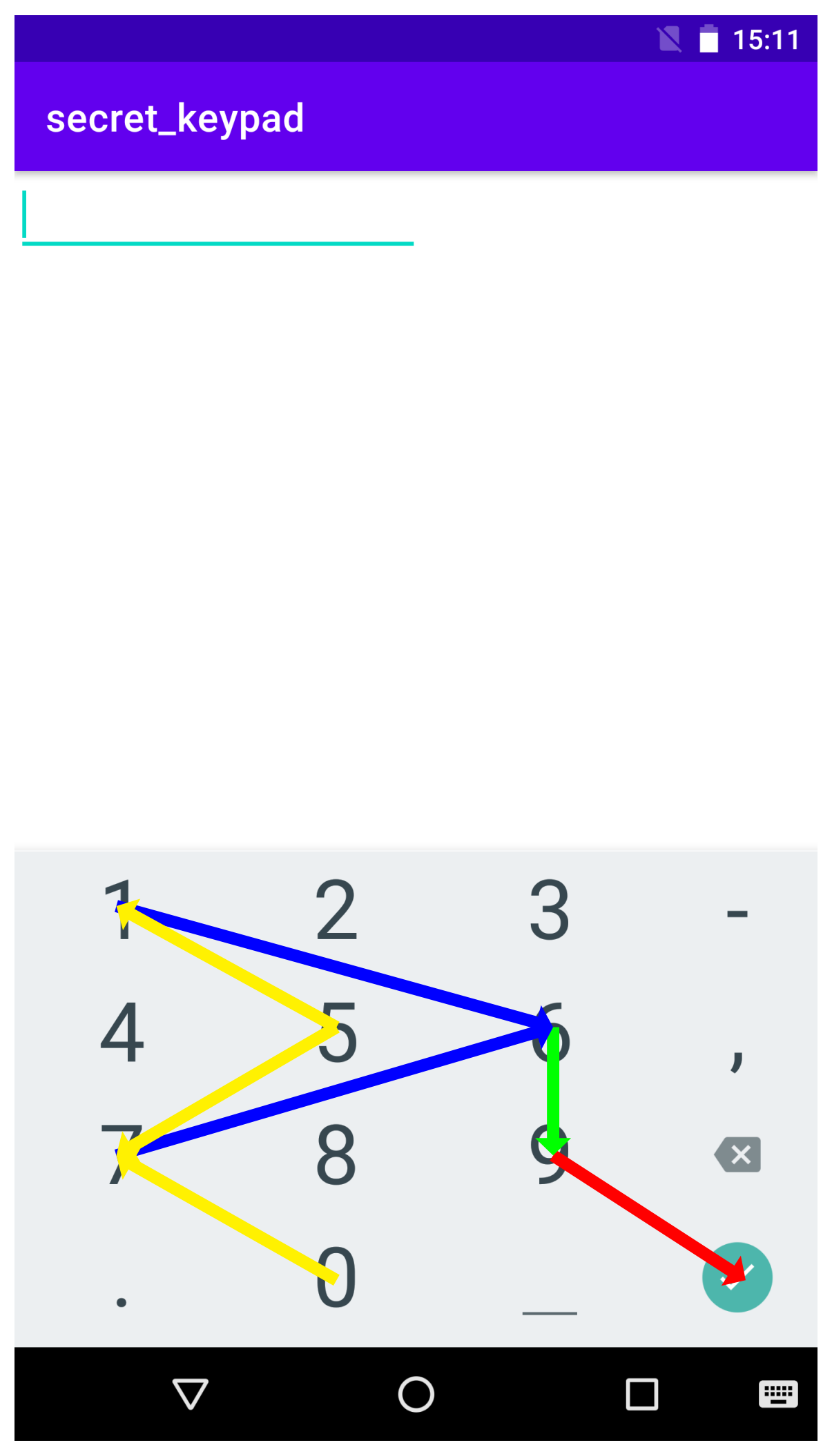


# of gyroscopic samples : 0
# of power samples : 303
Last sample : -407472

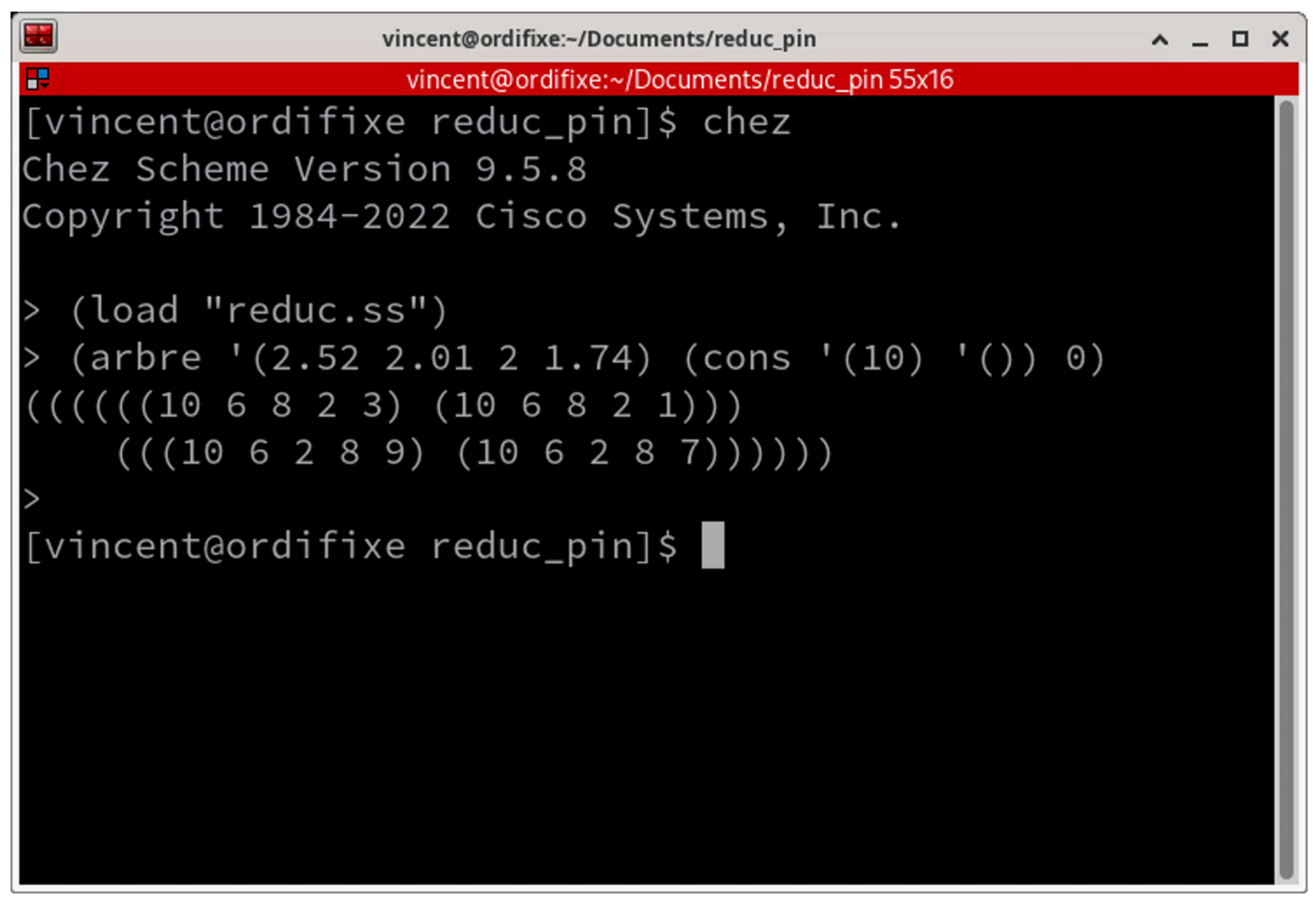
This capture shows the real-time current consumption trace obtained by our spying application during a screen touch, happening between sample number 12500 and sample number 22500. We can see a higher variance toward the bottom during this period: samples are negative because the current is flowing out of the battery.



This current consumption trace has been obtained during a PIN code entry. We can clearly see five peaks, each corresponding to a key press: the four numbers of the passcode, and the validation key.



Capture of a victim application displaying Android's native keypad. As the added arrows show, the tree of the most likely PIN codes starts from the validation key which is at a known location, and spawns new branches for each new possibility when approaching the beginning of the entry.



```
vincent@ordifixe:~/Documents/reduc_pin
vincent@ordifixe:~/Documents/reduc_pin 55x16
[vincent@ordifixe reduc_pin]$ chez
Chez Scheme Version 9.5.8
Copyright 1984-2022 Cisco Systems, Inc.

> (load "reduc.ss")
> (arbre '(2.52 2.01 2 1.74) (cons '(10) '()) 0)
((((((10 6 8 2 3) (10 6 8 2 1)))
    (((10 6 2 8 9) (10 6 2 8 7))))))
>
[vincent@ordifixe reduc_pin]$ █
```

Example use and output of the script deploying the tree according to four given temporal delays. The program has been configured with the geometrical data describing the shape of Android's native keyboard.

Attacking

We consider the four delays between the five key presses. An important fact we rely on is that the validation key the user needs to press in order to confirm his code is at a fixed, known location on the screen. Starting from the end, we can develop a tree by considering, at each step, the most likely previous key depending on the corresponding delay. Each branch of the resulting tree is one of the most likely PIN codes.

While this method provides good results, once the delays between the key presses are recovered, other techniques found in the state of the art can also be applied, such as Markov chains, artificial intelligence…

Defending

Removing this attack vector is easy if we control the system: patching Android completely suppresses this risk. In particular, the `BatteryManager` system service can be rectified very simply with just a few lines of code.

However, protecting an application is a difficult task if we cannot modify a system which we know is compromised. We are investigating solutions based on jamming, and fake sensitive signal simulation.

In the particular case of a PIN code entry, using a randomized keypad is a satisfying solution. However, it will break user accessibility, and will not protect against other power-related risks.

The authors would like to thank Guillaume Bouffard from the National Cybersecurity Agency of France (ANSSI) for his support throughout these works.