Remote Sensors Project - Final Report

Submitted by:

Tal Rotshtein	213041098	talrotshtein@campus.technion.ac.il
Lior Malka	212594881	lior.malka@campus.technion.ac.il
Yonatan Rosen	213041098	yonatanrozen@campus.technion.ac.il

Introduction

Side channel attacks is a cyber attack where an attacker uses unintended information leakage caused by the system's architecture and implementation to infer sensitive data, instead of attacking the implementation of the underlying system or algorithm itself. A common example is an attack that infers cryptographic key bits according to behavioural characteristics of the system such as the timing of certain actions and power consumption. For instance, a key with 100 '1' bits can result in heavier encryption operations than a key with only 10 '1' bits.

However, most side channel attacks face a major challenge: physical access to the victim is required, since leaked information such as precise power consumption, in most cases, cannot be gathered remotely.

Our interest lies in exploring the possibility of performing a side channel attack based on data about power consumption of CPUs, without physically accessing the device, but rather using software tools, and **performance counters** in particular.

Performance counters are software/hardware tools that gather general information about the system's performance. Many systems support many kinds of performance counters, in order to provide low-level insight to the execution of programs, which can help optimize their performance. Common examples of hardware metrics that can be measured using builtin hardware performance counters are CPU instruction and cycle counters, cache misses, memory access and more. Software performance counters can be used to measure/approximate stats such as CPU usage percentage, context switches, page faults and so on.
Whether the performance counters are software or hardware based, information from both can be gathered by applications, and do not require direct physical access to the machine. Meaning code execution ability on the machine is enough to utilize the performance counters present in it, and obtain the information they harvest.

The idea of **Our Project** was to utilize the above-mentioned performance counters, which are present in most systems, and see how accurately we can approximate the power consumption of a machine using them. If such accurate approximation can be acquired, it could open the door for new possibilities of power side channel attacks that

will not require physical access to the machine. Specifically, we aimed to achieve this approximation on ARM CPUs, which are widely used in mobile devices, IoT devices, embedded systems and even servers (a similar work exists for Intel CPUs[1]) The plan is to create machine learning algorithms, which given a measurement of the performance counters at a certain small time interval, will yield an approximation for the power the machine consumed at that time. The model can be trained using data we will gather from performance counters, and actual physical power measurements using a dedicated tool such as an oscilloscope. This way we can obtain labeled data: the features are the metrics from the performance counters, and the label is the actual power consumption measured in the lab.

The Project's Process

We divided the project into 3 parts: we started in research and reading about side-channel attacks in general, the technical components required for our experiments, and the specific system we would like to explore. Afterwards, we conducted experiments and gathered power consumption data and other data. Finally, we analyzed the data, trained a model and evaluated the results. We will now elaborate on each of the parts and the process.

Research and Reading

First, we learned about the different methods of power analysis in the context of side-channel attacks, such as Differential Power Analysis and Correlation Power Analysis[2]. These methods are a form of side-channel attacks in which the attacker studies the power consumption of a cryptographic hardware device and extracts vital information such as key and plaintext bits. The purpose of this phase was to understand how power analysis works in general for precise power measurements (measured with dedicated hardware), in order to imitate this process using the performance counters. Also, we read about methods that incorporate Machine Learning techniques into these specific side-channel attacks[3][4] as this was a core part of our project.

After that, we began exploring the technical components of the projects: the machine we chose for our project and that we will later perform experiments on, is a Raspberry Pi 4 Model B which has an ARM base architecture, supports linux, is very configurable and easily paired with external power monitoring tools, and also it is affordable so we could afford more devices in case something went wrong. In addition ARM processors are very prevalent these days in various systems, including mobile phones, IoT devices and embedded systems, which further motivated us to explore them.

We started with the configuration of the performance counters. At first we investigated whether there exists a performance counter tool which can directly evaluate or approximate real time power usage of the system, since such a metric can be very useful

to us. Unfortunately, we did not find such utility for the ARM architecture which can sample and display the power consumption in a high enough frequency.
After some further research and a search for an appropriate tool which can measure in high frequencies, we settled on the performance counters that are used through the perf utility in linux distributions. Perf is a powerful Linux profiling tool used to measure and analyze system performance. It provides insights into how programs interact with the hardware and the kernel, and leverages dedicated hardware components such as the Performance Monitoring Unit (PMU) in order to acquire and display accurate information. In some ARM devices perf's default configuration only includes basic metrics such as CPU cycles and number of instructions. In order to extend the available metrics and counters and to fully utilize perf to gather more specific data such as memory loads or cache/branch misses, we had to change the system's configuration[5].
Afterwards, we researched how to properly use perf and what metrics can be useful for our goals. We came to the conclusion that metrics such as number of cycles, retired instructions, cache-references and misses, and memory and bus accesses are commonly used to model power consumption.

Finally, we had to think of how to simulate different workloads on the Raspberry Pi board, so that later we can measure the above mentioned metrics, and get data which we can train a machine learning model on effectively. We found a linux tool called "stress-ng" which can simulate various kinds of workloads and target many CPU-level metrics. This, in combination with the "perf" utility, enabled us to create varied simulations which target the metrics we are interested in, as we will explain in detail in the following section.

Experiments and Data Collection

As we explained earlier, In the experiments we aimed to measure real time statistics like CPU cycles, instruction and cache misses, using the perf utility, and then combine the results with the actual power consumption (measured by the oscilloscope), in hope to find a correlation between the two.

On a high level, the process was:

1. Running different programs on the Raspberry Pi that simulate different events in the system, mainly using the "stress-ng" utility, and simultaneously measure system stats using perf.
2. run the same programs and measure real time power consumption in the lab.
3. Data pre-processing: normalizing the power consumption data and joining it with the data from perf.
4. Learning: using the data from perf as features, and the power consumption values as labels, and training a model in hope to find a correlation.

data collection using perf

Like we mentioned in the research part, the statistics we decided to measure were CPU cycles, instructions, cache references, cache misses, memory accesses, and bus accesses.
For the programs, we used the linux stress-ng utility. Stress is a lightweight Linux utility used for stress testing a system by applying configurable workloads on various hardware resources. It allows running programs that generate high CPU usage, consume memory, perform disk I/O operations, and spawn multiple processes, allowing users to assess system performance, stability, and other parameters under different workloads.
Stress is used via command lines in linux, and receives different arguments that will define the nature of the program it will run[6]. For example, the "--cpu-method" flag determines which type of computation the CPU workers will perform to stress the processor. Different methods target different specific CPU operations. Further details about the flags and command line arguments provided can be found in the stress-ng manual.

Below are some examples of the programs we ran using stress:

```
# Cycle through all available CPU stress methods
stress-ng --cpu 1 --cpu-method all --timeout 10s
stress-ng --cpu 2 --cpu-method all --timeout 10s
stress-ng --cpu 3 --cpu-method all --timeout 10s
stress-ng --cpu 4 --cpu-method all --timeout 10s

# stress cache accesses
stress-ng --cache 1 --timeout 10s
stress-ng --cache 2 --timeout 10s
stress-ng --cache 3 --timeout 10s
stress-ng --cache 4 --timeout 10s

# Simulate memory accesses
stress-ng --vm 1 --vm-bytes 64M --timeout 10s
stress-ng --vm 2 --vm-bytes 64M --timeout 10s
stress-ng --vm 3 --vm-bytes 64M --timeout 10s
stress-ng --vm 4 --vm-bytes 32M --timeout 10s

# Simulate accessing heap memory
stress-ng --bigheap 1 --timeout 10s
stress-ng --bigheap 2 --timeout 10s
```

```
stress-ng --bigheap 3 --timeout 10s
stress-ng --bigheap 4 --timeout 10s


# Combined:
stress-ng --cpu 2 --bigheap 2 --cache 2 --vm 2 --vm-bytes 256M
```

As we can see, we ran many different programs with various parameters, to collect a more varied data, which will later contribute to a more robust model.
We concatenate each stress command with perf in the following way:

```
perf stat -I 100 -o out.txt -e cycles, instructions,
cache-references, cache-misses, branch-misses, mem_access,
bus_access stress-ng --cpu 2 --cache 2 --vm 2 --vm-bytes 256M
--timeout 10s

# value after -I defines measuring resolution
```

We later parsed Perf's output from a "out.txt" into a csv file.

Measurement With the Oscilloscope

Initially, we observed that direct measurement of the Raspberry Pi's power consumption using an oscilloscope yielded highly noisy waveforms. This was due to the lack of a dedicated pin capable of measuring the total current flowing through the device. Consequently, we determined that directly connecting the oscilloscope to the Raspberry Pi was not a viable approach.

To address this, we modified the Raspberry Pi's power supply by introducing a resistor into its power cable. This allowed us to measure the entire current drawn by the device by connecting the oscilloscope in parallel with the resistor. While this method enabled current measurement, the recorded waveforms remained noisy. Upon investigation, we concluded that the excessive noise was caused by the length of the cables used to connect the oscilloscope. Long cables can act as antennas, introducing electromagnetic interference and degrading the signal quality.

To mitigate this issue, we shortened the cables, which significantly reduced the noise in the measurements. However, additional inconsistencies persisted. We hypothesized that the noise was due to variations in the Raspberry Pi's power consumption caused by background processes and services running on the operating system. For instance, services such as Wi-Fi and Bluetooth communication contributed to power fluctuations in addition to electromagnetic interference, resulting in unreliable data.

To minimize the impact of these background processes, we disabled Wi-Fi and Bluetooth and reduced the OS interface to a simple terminal. Despite these efforts, our results still lacked the desired precision due to residual power consumption from connected peripherals and OS-level services.

Finally, we developed a solution to streamline the measurement process: We configured the Raspberry Pi to execute our performance monitoring script automatically upon boot, and disconnected peripherals like HDMI, mouse, and keyboard, which were identified as additional power consumers. By registering the script to the system's startup registry, we eliminated the need for external peripherals and cables during operation. This approach significantly improved the accuracy of the measurements and provided the most reliable data we had achieved thus far.

Data Analyzing and Learning

After collecting data in the previous part, we started analyzing it before the learning phase: we took the data from the waveform, and converted each Perf time frame (the size in *ms* after the “-I” flag in the *perf* command) to the average of all the voltage values measured in that frame. This way, we could match each perf measurement with a label that represents the average voltage in the measurement time frame, and use a ML to approximate this average as accurately as possible. Then, we joined the data from the waveform with the data from perf and received one csv file with the perf metrics as features and the average voltage in each timeframe as labels. We then scaled all of the data so that our model won’t be biased to certain features. Finally we splitted the dataset into train and test sets.

For the model itself, we chose SVR which is like SVM for regression (our labels are continuous), and performed hyperparameters tuning with grid search. For most of the test set, the residuals (the squared distance of the real label from the predicted label) were very small, thus the model succeeded in predicting values that are close to the actual voltage. The results of the learning phase are described in detail in the notebook.

Further Steps & Suggestions

First, we suggest finding a way to improve the measurements with the oscilloscope and make them less noisy. In our experiment we tried to minimize noisy factors but further steps are required to ensure that the measurements are as clean as possible. For example, perhaps using the oscilloscope in an environment with as little transmitting devices as possible (Wi-Fi, computers, telephones etc) will improve the accuracy of the signals we receive. Also, it is recommended to connect the oscilloscope probes after the

Raspberry Pi's internal capacitors, since this measures the actual power consumption of the CPU more accurately (note that this requires further engagement with the hardware and deeper understanding). Finally, we suggest using stronger models or neural networks in the learning phase to learn more complex relationships in the data.

References

[1] The use of Performance Counters to perform Side-Channel attacks
[2] Power Analysis Methods
[3] Side Channel and Micro-architectural Attacks on Modern Cryptosystems
[4] ASCAD
[5] Enabling Raspberry Pi Performance Counter Support on Linux perf_event
[6] Stress testing real-time systems with stress-ng