École Polytechnique Fédérale de Lausanne

ARM Cache Paths
Analyzing program paths from cache timings

by Victor Faltings

Bachelor Project Report

Approved by the Examining Committee:

Dr. Marcel Busch
Thesis Advisor

Prof. Dr. sc. ETH Mathias Payer
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 8, 2022

Abstract

The *ARM CacheFuzz* project is an ARM fuzzer that aims to perform grey-box fuzzing without instrumentation. Instead, the fuzzer makes use of cache timing information to infer program path coverage.

Our intuition is that when looking at cache timings, we can extract a pattern of cache misses and hits corresponding to a program path.

To verify our intuition, we perform *Prime+Probe* timing attacks on some simple victim programs at semi-regular time intervals on every cache set and gather the results in a heatmap matrix.

From our results, we are not able to consistently identify an obvious pattern for a given program. We believe this to be due to excessive noise caused by other processes or hardware optimizations by the device. We then propose some interesting possible extensions to explore that could help to extract a signal from our cache timing matrices.

Contents

Chapter 1

Introduction

Fuzzing is an automated program testing technique which involves running a target program with many different inputs in an attempt to trigger unexpected behavior [2]. The most basic approach to fuzzing involves continuously running the target with completely random inputs. However, modern fuzzers use feedback from previously generated inputs in order to mutate the input in more interesting ways. In particular, a common interest is the path coverage of the program. Grey-box fuzzers maximize this coverage by using coverage information obtained by instrumenting the target binary. The most popular such coverage-guided fuzzer is AFL++.

The *ARM CacheFuzz* project is a fork of AFL++ that performs grey-box fuzzing on Android ARM smartphones. However we do not rely on instrumentation to obtain coverage information and instead use cache timing attacks, specifically *Prime+Probe*. This is helpful because it allows us to gain the benefits of coverage-guided fuzzing without needing to recompile the source code with instrumentation (which is not always possible). To transform the data obtained from the timing attacks into coverage information we make use of an *Analyzer*. Figure 1.1 shows how the different components of the fuzzer interact.

The analyzer is responsible for feeding information back to AFL regarding the program coverage of the previous input through a bitmap. AFL is particularly interested in knowing whether or not an input has covered a new program path or not. Note that AFL does not require a particular program path to be encoded in a specific way in the bitmap. Therefore, the job of the analyzer is fundamentally to be able to differentiate new program paths from previously encountered ones.

The goal of this project is to maximize the performance of the analyzer. We are interested in being able to differentiate program paths, while also aiming for a low runtime.

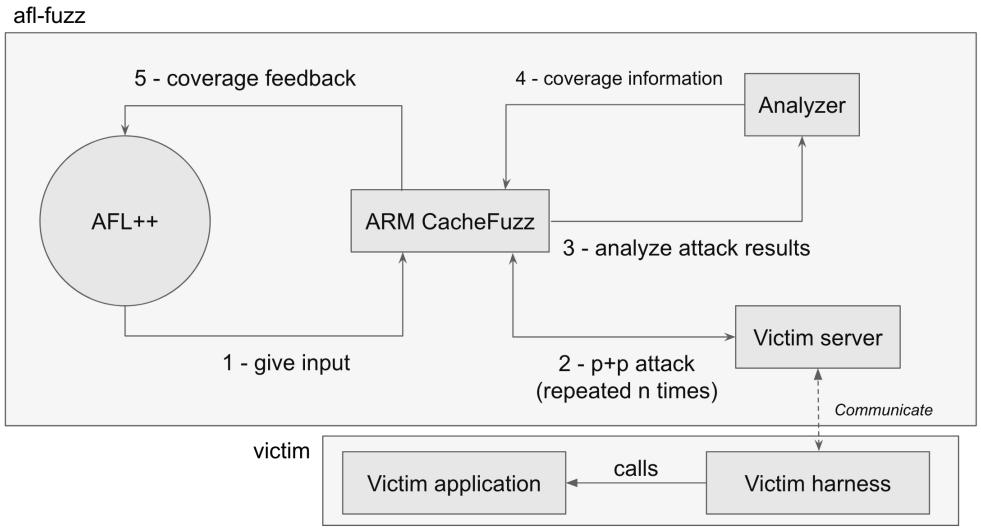

Figure 1.1: ARM-cachefuzz overview

Chapter 2

Background

Timing attacks are an information side-channel in which an attacker can analyze the time taken to execute certain actions [4]. In particular, cache timing attacks measure the access time of a given cache set in order to infer whether or not that set was used by a victim program. One such attack is *Prime+Probe* (P+P) which works as follows:

- The attacker primes a cache set X by filling it with it's own data (the actual contents of the data do not matter, the attacker only needs to remember what they put in)
- The victim program will then continue it's normal execution. However, if the victim needs to use cache set X for either a program instruction or program data it will find that the cache set is full. Therefore the cache will need to evict one of the pieces of data placed by the attacker to make room for the victim data.
- Finally, the attacker tries to reload all the data that was previously stored in the cache and measures the time it takes to do so. If some of the data takes particularly long to load then it is likely that that data was evicted from the cache (the long access time comes from the hardware needing to fetch the data from memory). In this case the attacker can infer that the victim made use of cache set X.

In their 2016 paper, "ARMaggedon: Cache Attacks on Mobile Devices" [3], researchers from the Graz University of Technology showed that ARM mobile devices are vulnerable to the same cache timing attacks as Intel x86 CPUs. Alongside the paper, the `LibFlush` library was published [1]. With this library, we are able to easily perform cache timing attacks, such as P+P, on Android ARM smartphones.

Using P+P we are able to measure the access time of a particular cache set and determine whether or not that set was used by a different process. This leaks information on the behavior of such other processes. In particular, program instructions are loaded into the cache so we can

gather information on what instructions a process is executing. By continuously performing these attacks in sequence over the course of a process' lifetime we can therefore in theory obtain information on the executed program path of the process.

We therefore introduce the notion of frame windows which are time slices corresponding to one P+P cycle. In each frame window, we collect the access time of every cache set of the system and group the results in a 2-dimensional matrix as shown in Figure 2.1. In this matrix, each row represents a frame window and each column represents a cache set. The numbers in each cell (i, j) represent the measured access time of cache set j in frame window i.

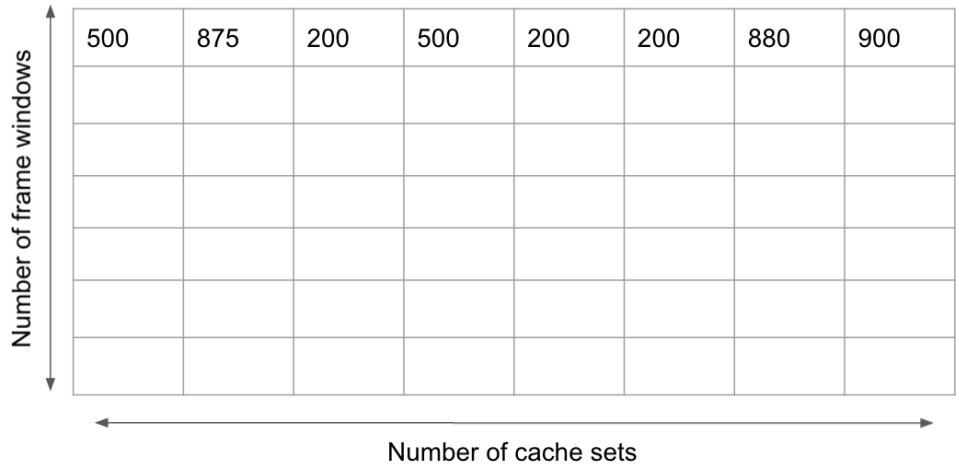

Figure 2.1: Cache timing matrix

Chapter 3

Challenges

Assuming a theoretical perfect-world case where we have timing matrices such that in a given frame window, exactly one instruction from the victim program is executed and we have no scheduling issues, we still have some challenges to deal with.

3.1 Reduced Address Space

The first obvious challenge is that the number of cache sets is far smaller than the number of different instruction addresses for most programs. Therefore, multiple different addresses will map to the same cache set; this is known as *congruent addresses*. Due to this, we cannot uniquely identify any address from its cache set. Furthermore, cache sets contain many bytes (depending on the cache line length) so consecutive instructions will also map to the same cache set.

Example 1 *We assume a direct-mapped cache with 16 sets, and 32 byte cache line length, byte-addressable memory and 64 bit instructions. If a process executes an instruction at address* `0x000`*,* `0x008` *or* `0x200` *they will all map to cache set 0. Therefore we cannot tell them apart in our timing matrix as shown in Figure 3.1.*

We believe that this issue will not be very significant when looking at the entire timing matrix. Although it is possible to have congruent addresses in a given binary it is unlikely we encounter these conflicting pairs in the same frame windows for two different program paths. In the rare case where this would happen, we don't expect the rest of the frame windows to closely resemble each other. In other words, we expect different program paths to leave different fingerprints in the cache. The exception is in the case of very small changes in the program path. For example if the two different possible destinations for a control flow jump are contained in the same cache line we cannot tell them apart.

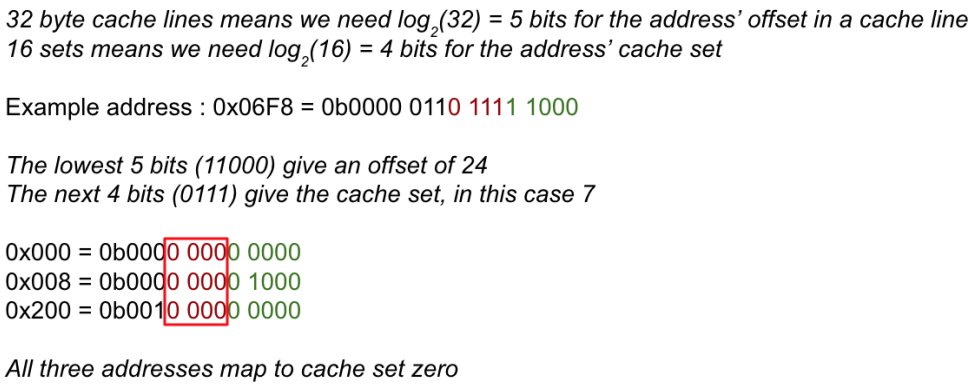

Figure 3.1: Congruent memory addresses

Example 2 *We assume a direct-mapped cache with 16 sets, and 64 byte cache line length, byte-addressable memory and 64 bit instructions. If we consider the following code snippet and example paths A and B we cannot differentiate them by looking at the cache sets.*

```
0x00    beq    r1, r2, instr2
0x08    instr1
0x10    br      end
0x18    instr2
0x20    instr3
0x28    end
```

- *Program path A:* `r1 == r2 : 0x00, 0x18, 0x20, 0x28`
- *Program path B:* `r1 != r2 : 0x00, 0x08, 0x10, 0x28`

Because of the line length of our example cache, all instructions will map to cache set zero. In both situations, our timing matrix contains four cache misses on set zero.

3.2 Cache pollution

The second challenge is that the timing matrices contain a lot of noise because of other processes running concurrently on the hardware. Because we are timing the accesses of the L2 cache we also have to deal with the noise introduced by other cores. From the timing information it is impossible to determine whether or not a cache set was used by our target *victim process* or a different process/the OS. We statistically mitigate this problem by running a program path multiple times and averaging out the noise.

Furthermore, there are additional problems when we consider the real-world situation.

3.3 Time shifts

The first is that P+P cycles are not atomic. This means that when analyzing the timing matrix we cannot determine the total order of cache sets accessed by the victim. At every frame window we can only identify a set of accessed cache sets (and therefore program instructions), which gives a partial order to the program path. Due to small timing differences, certain cache set accesses can also be shifted to the succeeding/preceding frame window.

This is illustrated in Figure 3.2. We can see example timing matrices from separate runs of the same program path. However some of the cache misses and therefore high access times get delayed by one frame window for the first and second cache set.

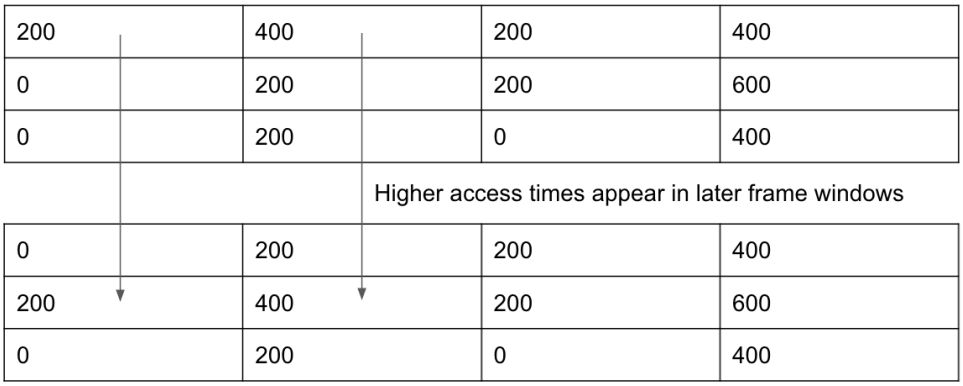

Figure 3.2: Timing matrix shifts

3.4 Descheduling

Secondly, we must deal with the phone's OS' scheduler. It is likely that our 2 processes, the victim process and the P+P process, will be descheduled and rescheduled by the OS during the lifetime of the victim process. When this happens 2 problems can arise. If the victim process is descheduled and the P+P process continues to run we will only be measuring noise in our timing matrix. If the P+P process is descheduled while the victim process continues to run we will miss out on instructions executed by the victim. In some situations this can also lead to a very high number of cache misses in a single frame window. These scheduling problems also lead to gaps and offsets between timing matrices of the same program path.

3.5 Probing instructions

Finally, there is some inherent loss of information with the implementation of our P+P thread. We must probe each cache set in a loop, which means the same instruction will be executed many times by the P+P thread in turn polluting the cache. This means that whichever cache set this instruction maps to will be accessed heavily in the cache. In turn, this means that if the victim is making use of this cache set, we will not be able to know that because we cannot differentiate between the victim making use of that set or our P+P thread making use of that set.

Chapter 4

Goals

Our goal is to consider as much context as possible when analyzing the timing matrices. In the scope of this project a very broad definition of context could be the number of frame windows that go into a comparison between 2 cache timing matrices. The minimum amount of context would therefore be if we do a row-wise comparison only looking at the same frame window in both matrices. The maximum would be achieved if both matrices can be compared to each other in their entirety.

The intuition here is that the same program path should not produce wildly different timing matrices. We have established that it is probable that some information can be offset to a later frame window due to scheduling/timing problems. However, if we are dealing with the same program path this information should likely show up somewhere in the timing matrix. If we limit ourselves to only considering a single frame window at a time we can easily be fooled by these problems but if we look for the information elsewhere in the matrix it is likely that we will find it.

We have also established that we would like to run the same program path several times in order to help eliminate some of the noise caused by other processes running on the hardware. It is therefore very important that we consider multiple frame windows when creating an "average" representation of a program path.

To achieve our goal we will aim to cluster the timing matrices into clusters representing program paths. We can then define a distance metric and compute a threshold using test data. Our first objective is to produce an easily human-interpretable representation of these timing matrices. The natural choice was to build heatmaps. Cells with higher timing measurements will appear as "hotter" and with high probability a cache miss. The next few sections will present the resulting heatmaps of basic test programs.

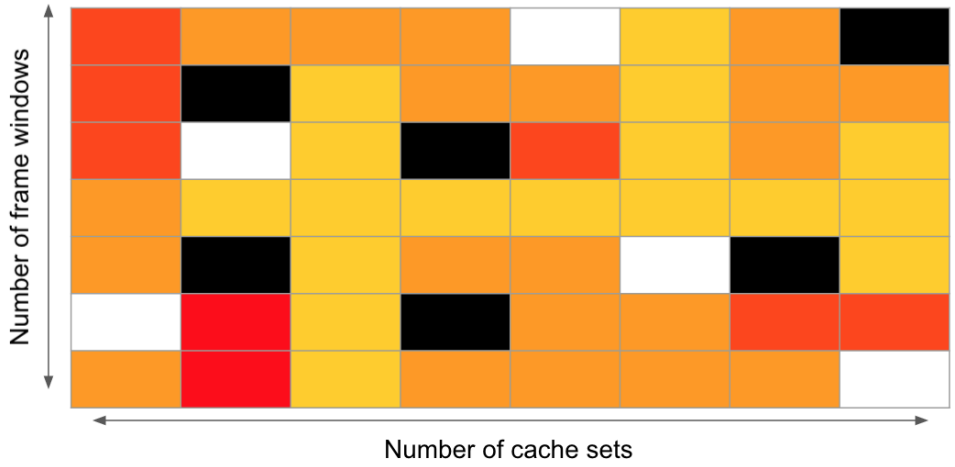

Figure 4.1: Example cache timing heatmap

Chapter 5

Experiments

5.1 Experiment platform

All the following experiments are performed on a Samsung S6 phone, which contains 8 cores spread over 2 boards. Note that cores 0-3 share a separate L2 cache from cores 4-7. The P+P thread runs on core 0 and the victim thread runs on core 1. The L2 cache configuration of the phone is 256 sets of 64 byte long cache lines. The experiments are launched as background processes by using ‘adb‘, ‘busybox‘ and ‘nohup‘. For every program, we perform 50 repetitions of the program.

5.2 Single-set

5.2.1 Program description

The most basic program we created was the *single-set loop*. This program allocates a single byte of memory on the heap. It then forks a child process. In this situation, the child represents a victim program. The victim continuously accesses the allocated byte of memory using the `libflush_access_memory` function provided by the `LibFlush` library. Meanwhile, the parent process allocates a 2d timing matrix and performs a given number of P+P cycles (the number of frame windows is a user-provided argument to the program). In total, the program constructs and saves `nb_reps` timing matrices using the same allocated byte of heap memory (where `nb_reps` is a user-provided argument).

5.2.2 Behavior

The expected behavior of this program is that a single set corresponding to the allocated byte of memory will appear as very hot on the resulting heatmaps. This set should also be the same across all different repetitions (different timing matrices).

The following figures show one example heatmap of a single repetition as well as a heatmap of the average of all repetitions. For the moment we average the matrices in the most basic way (a simple per-cell average).

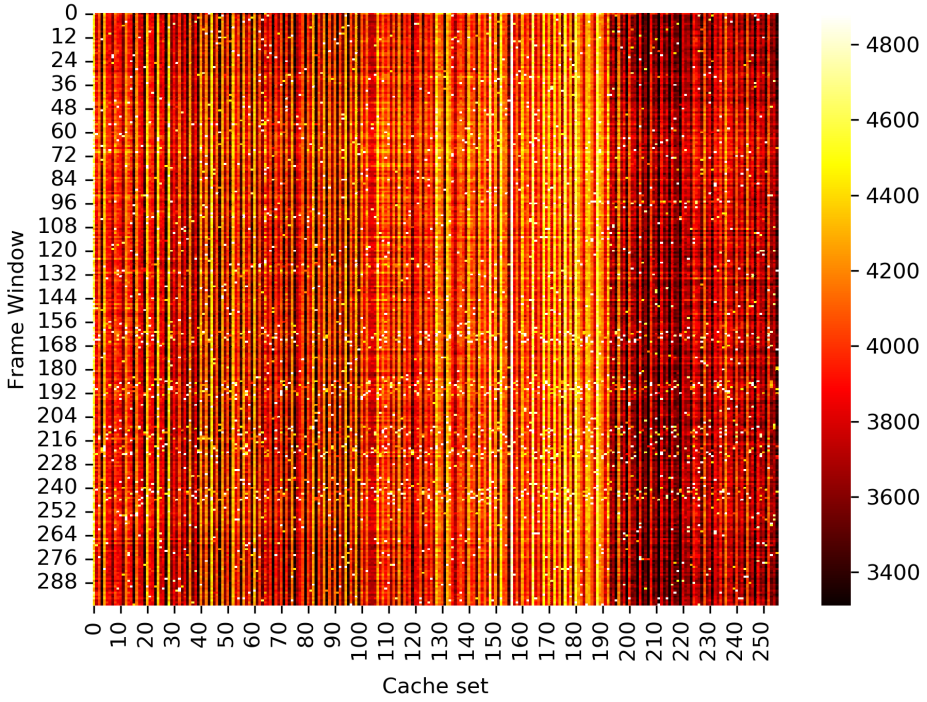

Figure 5.1: Single-set loop average (50 runs)

5.2.3 Discussion

As we can see from the heatmap of the average of all runs there seems to be some cache set which are accessed more often than the others. We can notice clear vertical lines of heat in some parts of the cache and very dark lines in other sets. However, we are not sure whether the white vertical line around cache set 155 corresponds to the byte of memory being accessed or our P+P instructions as discussed in 3.5.

Comparing Figure 5.2 to the average heatmap we can notice a clear resemblance in the colors of the cache set. The same cache sets that appear dark in the single run also tend to be dark in

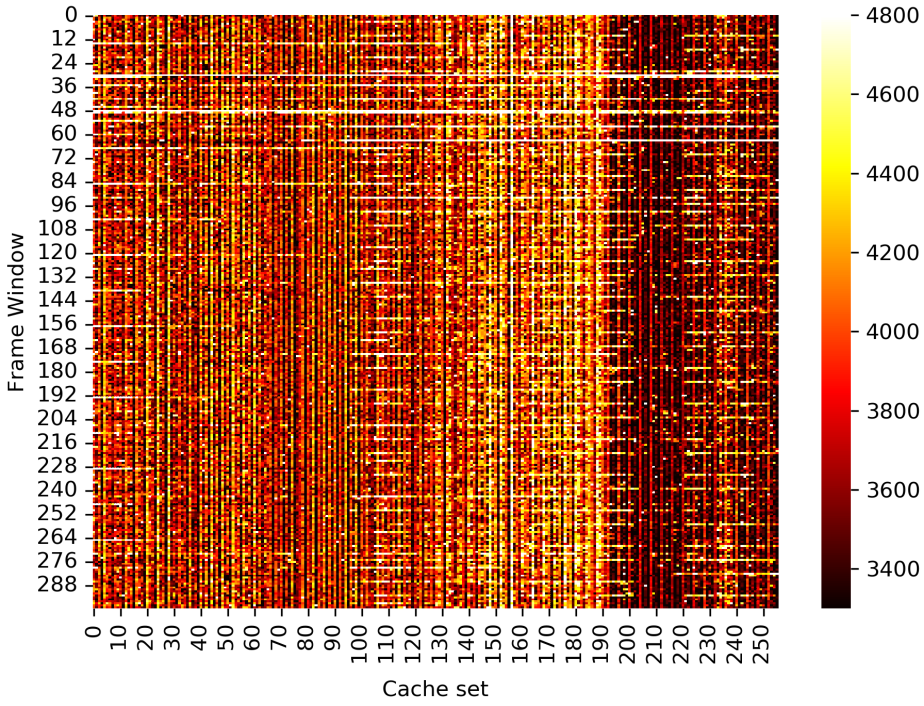

Figure 5.2: Single-set loop single run

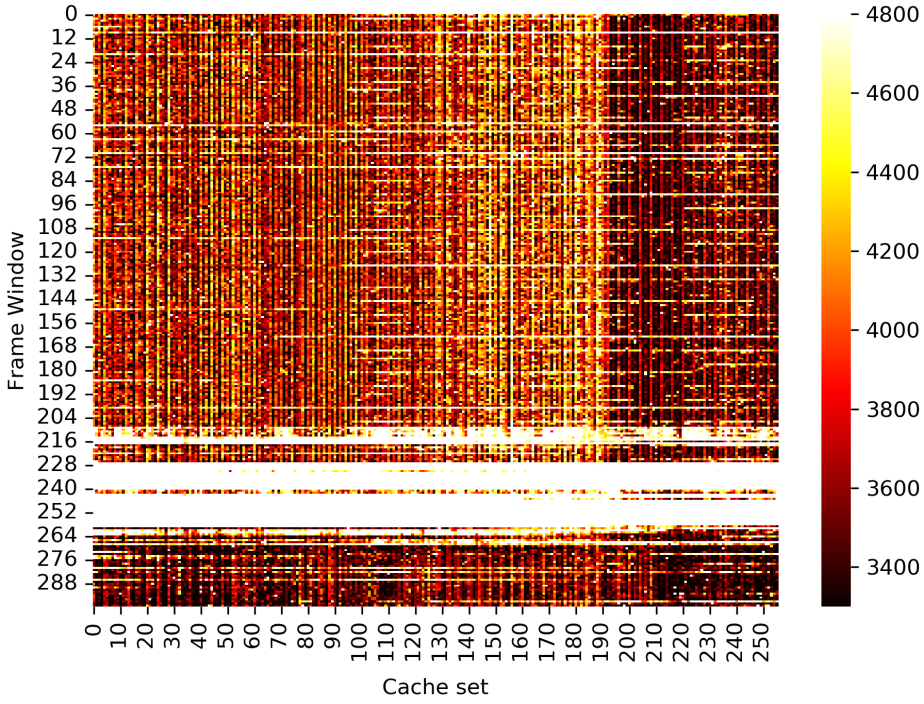

Figure 5.3: Single-set loop single run

the average, and likewise for the hot sets. We can also notice the occasional horizontal line of heat. We believe that this is due to descheduling by the OS as discussed in 3.4.

Looking at Figure 5.3 however we can see that certain runs can have a lot more noise than others. In certain runs, we notice a lot more of these horizontal lines, sometimes spanning several frame windows. In this situation, we believe it is possible that the OS was very busy with other system processes, which impacts our data greatly.

5.2.4 Disabling CPU cores

In an attempt to reduce some of the noise we also explored the possibility of deactivating certain cores of the hardware. We explored the deactivation of either only cores 2-3 as well as cores 2-7. The results are shown in Figures 5.4 and 5.5.

We cannot notice any significant reduction of noise when comparing the results being run on 6 cores (Figure 5.4) or all 8 cores (Figure 5.1). However, we can notice that Figure 5.5 appears much brighter than the other two heatmaps. We believe that this is because due to the phone only operating on two cores it must more frequently deschedule our processes, which leads to higher access times on average. On top of this, all activity from cores 4-7 (which usually make use of a separate L2 cache) now show up in our measurements.

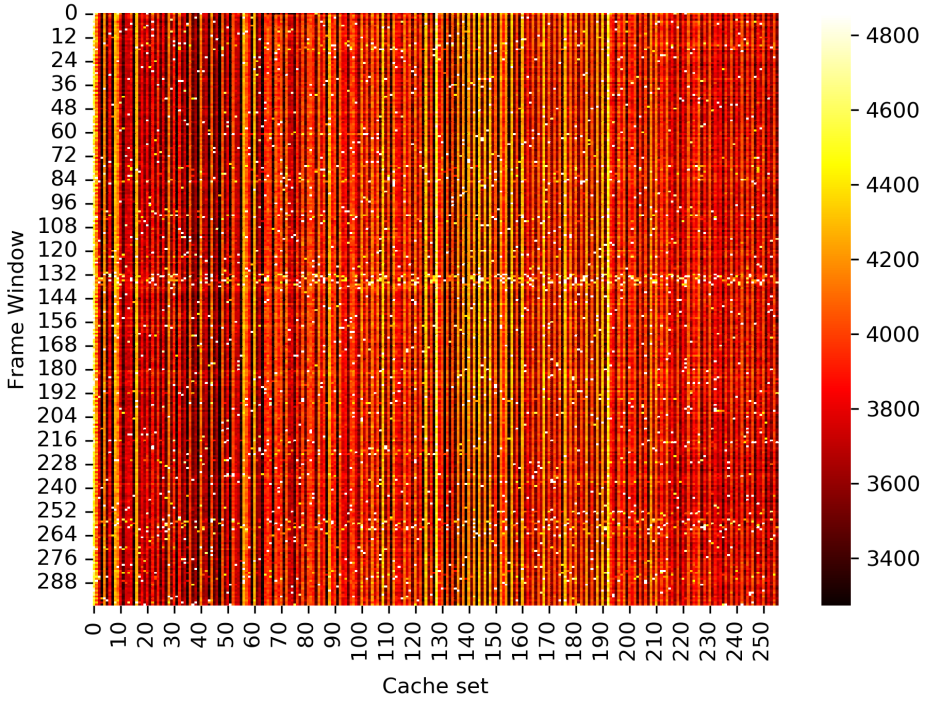

Figure 5.4: Disabling cores two and three

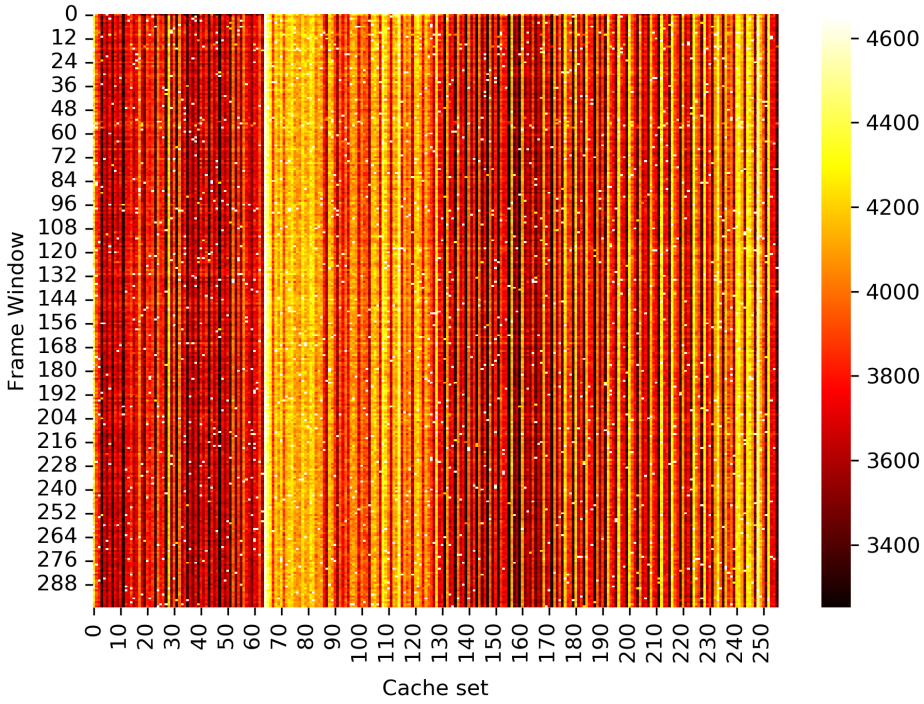

Figure 5.5: Disabling cores 2-7

5.3 Staircase

5.3.1 Program description

The next test program we present is what is referred to as the *Staircase* program. This program is very similar to the *single-set loop* with the main differences being the behavior of the child process as well as the allocation of memory at the start of the program. We now allocate a total of $n * k$ bytes of memory where n represents the number of cache sets of the phone and k represents the length in bytes of each cache line. On our S6 these numbers correspond to 256 sets and 64 bytes per line. The victim process now attempts the allocated array at increasing offsets. The goal is that the victim will access the subsequent cache sets at each subsequent frame window. In order to synchronize the incrementation of the accessed offset and the frame windows the child process is given a pointer to an offset variable that is incremented by the parent at every frame window.

5.3.2 Behavior

Our expectation for the resulting pattern is sets being accessed in a linearly increasing function of the frame windows. This pattern could aptly be described as a staircase.

Figure 5.6 shows the resulting heatmap of the average of all runs. Figures 5.7 and 5.8 show the resulting heatmap of two single runs.

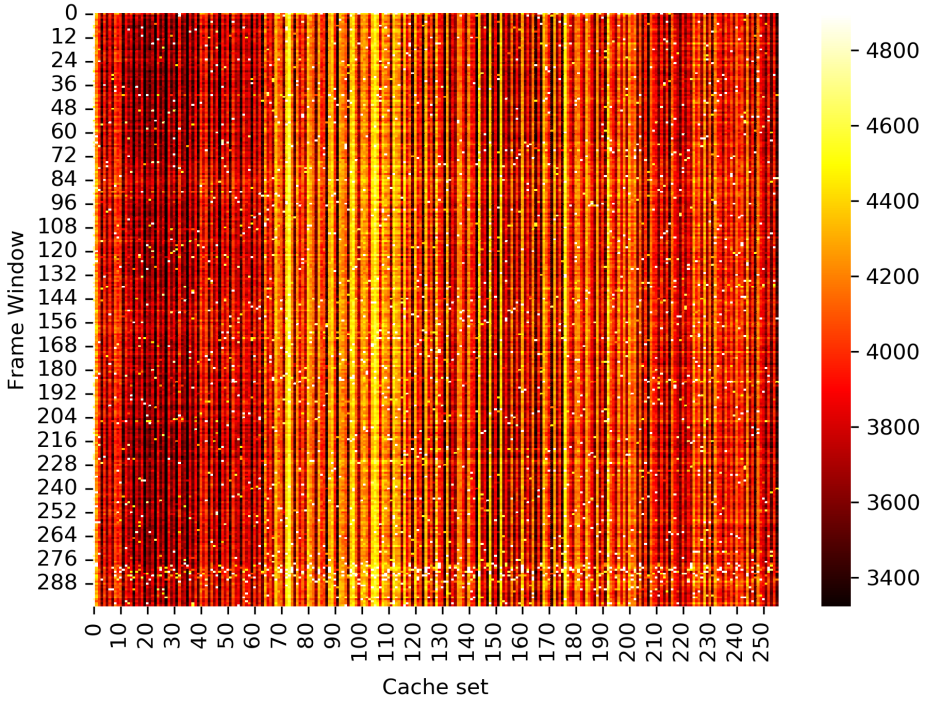

Figure 5.6: Staircase average (50 runs)

5.3.3 Discussion

Unfortunately we cannot identify a clear pattern resembling a staircase as was the goal. The heatmap is overall very bright and there does not appear to be a strong relation between the brightness of a cache set and the frame window (as should be the case because the accessed set changes each frame window).

Looking at the heatmaps of individual runs we notice the presence of a lot more brightly colored cells. In particular, we notice the appearance of partially filled horizontal lines. This leads us to believe that some of the behavior could stem from cache line prefetching or other hardware optimizations.

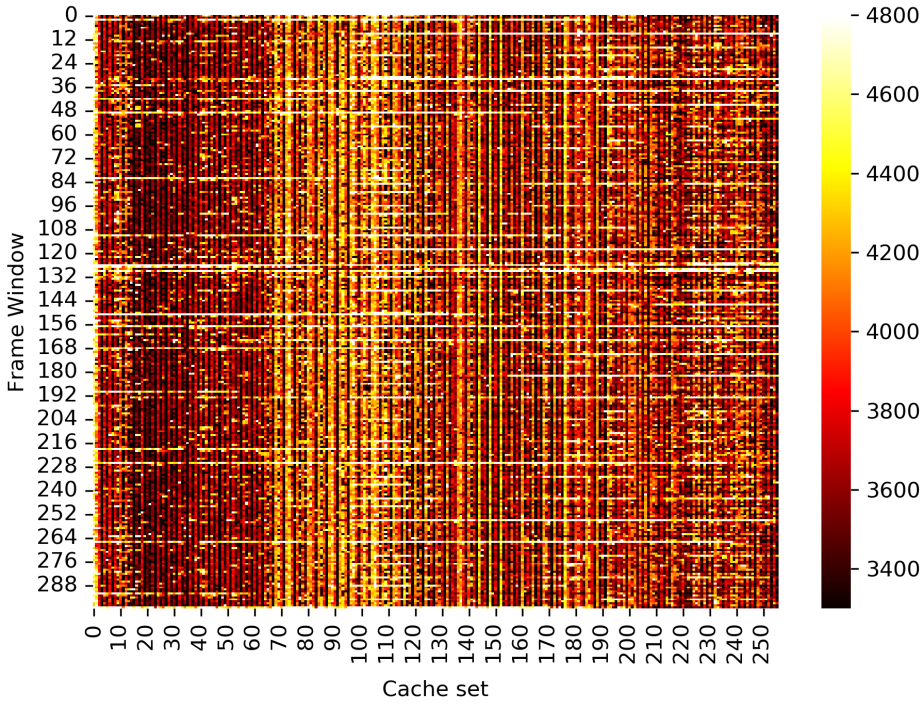

Figure 5.7: Staircase single run

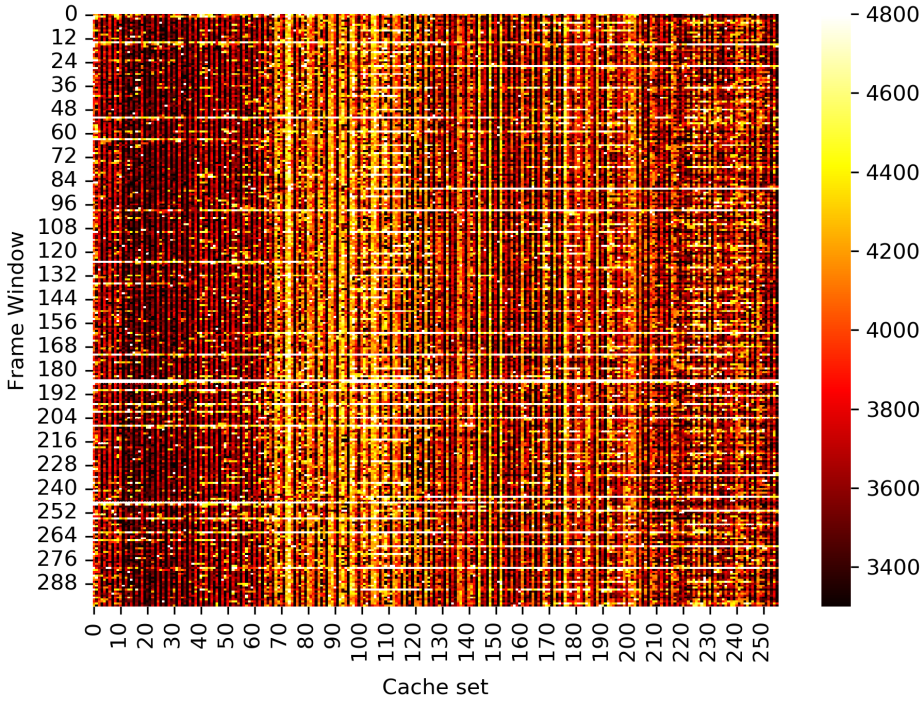

Figure 5.8: Staircase single run

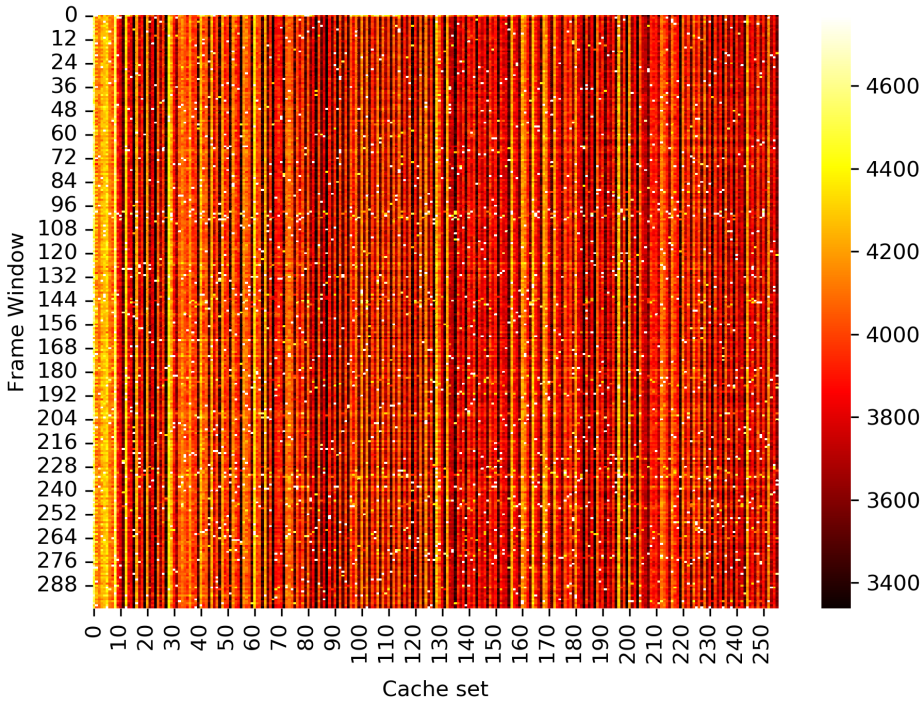

Figure 5.9: Disabling cores two and three

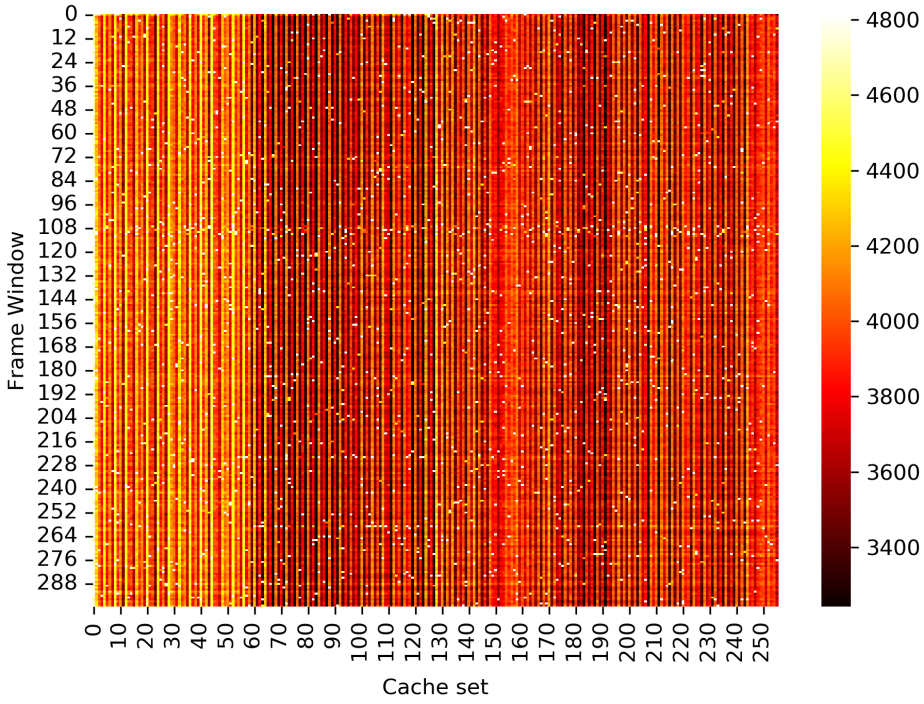

Figure 5.10: Disabling cores 2-7

5.3.4 Deactivation of cores

We again performed the experiments with some of the cores deactivated and present the resulting heatmaps in figures 5.9 and 5.10. As with the *single-set loop* there does not seem to be much benefit according to our current method of visualizing the matrices.

5.4 Dashed-line

5.4.1 Program description

Our next program is the *dashed-line*. Our goal now is to have the same program as the *single-set loop* with the difference being the victim now alternates between accessing the memory location and remaining idle for a given frame window.

5.4.2 Behavior

The behavior we expect to see for this program is similar to the 'single-set' program. However, as the victim will occasionally idle for a one or two frame window we expected the line to be alternating between high access times and low access times. On a heatmap this would show up as alternating between very bright and very dark spots.

Once again we show an average heatmap as well as a few heatmaps of a single run.

5.4.3 Discussion

Looking at the average heatmap we can see overall the presence of more heat, i.e. higher access times. We notice at least 2 very clear vertical lines which appear very hot. We believe these lines correspond to the instructions used for our P+P attacks. However, we unfortunately cannot see any noticeable dashed line. What we see instead is that the entire matrix seems to have a dashed look, implying that when the victim process is idle there are many less cache sets being accessed (not just the single memory location cache set).

This observation is even more clear when looking at the heatmaps of individual runs. We now see a lot of bright cells which seem to appear at semi-regular intervals. We are still not sure why the effect of sleeping during a couple frame-windows followed by being active for a couple frame-windows seems to cause an impact on so many different cache sets.

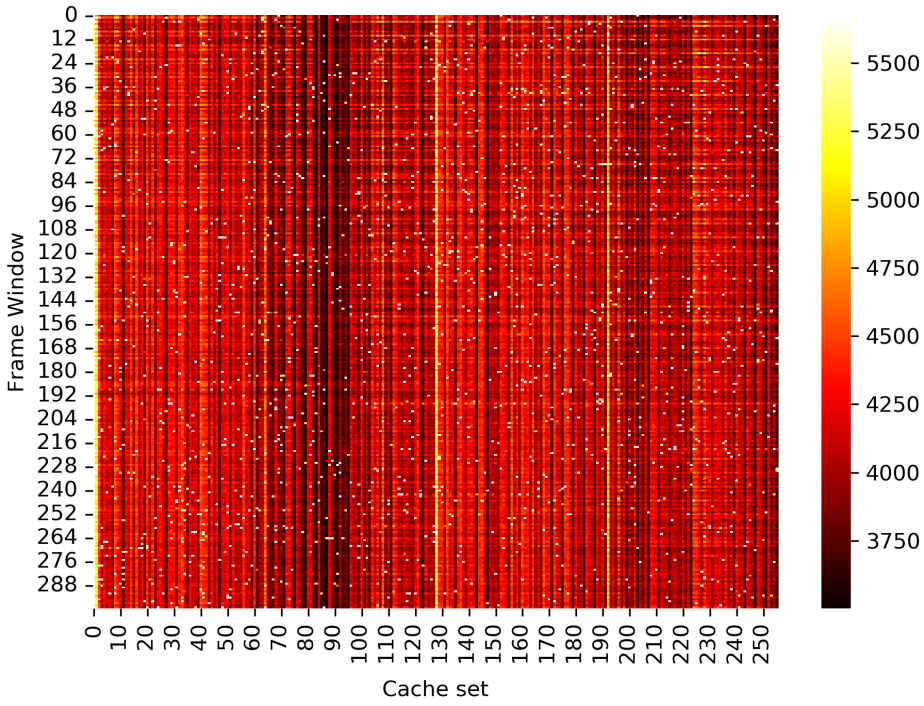

Figure 5.11: Dashed-line average (50 runs)

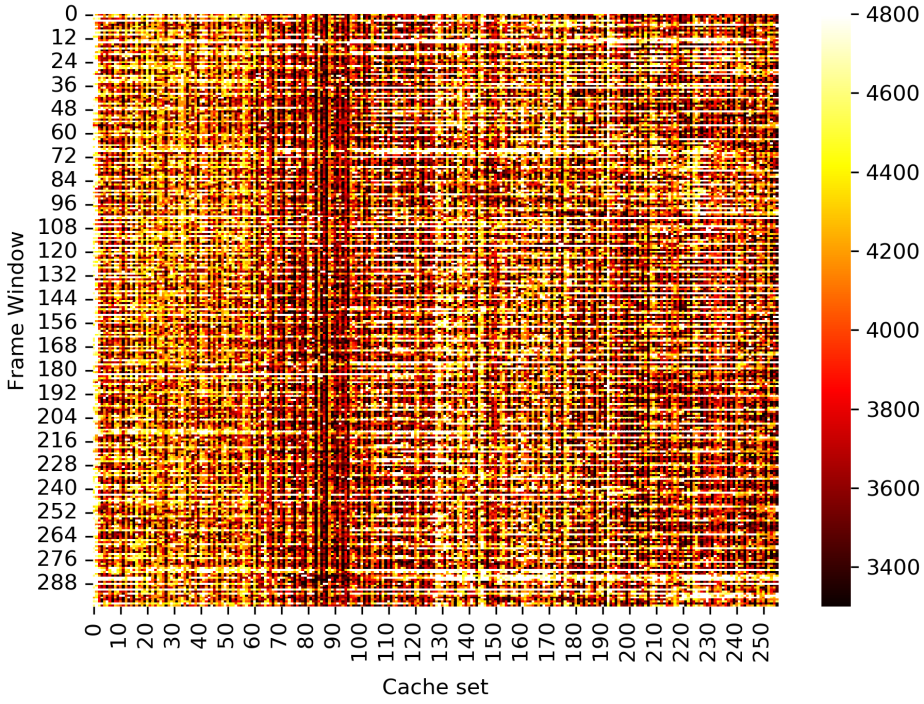

Figure 5.12: Dashed-line single run

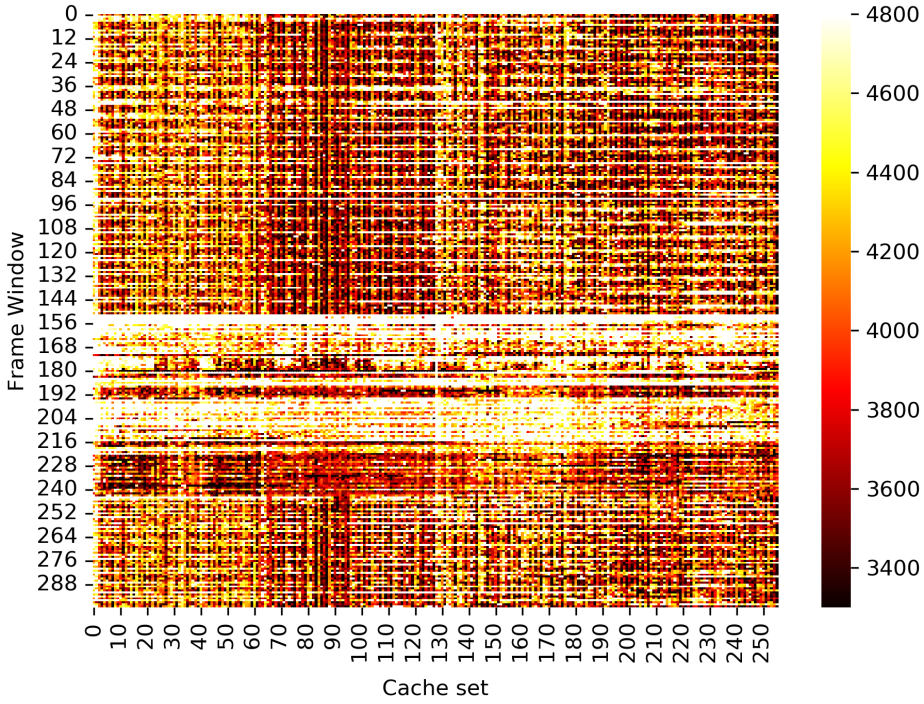

Figure 5.13: Dashed-line single run

5.4.4 Deactivating cores

Once again, we explored the possibility of deactivating certain cores and present the roles in figures 5.14 and 5.15

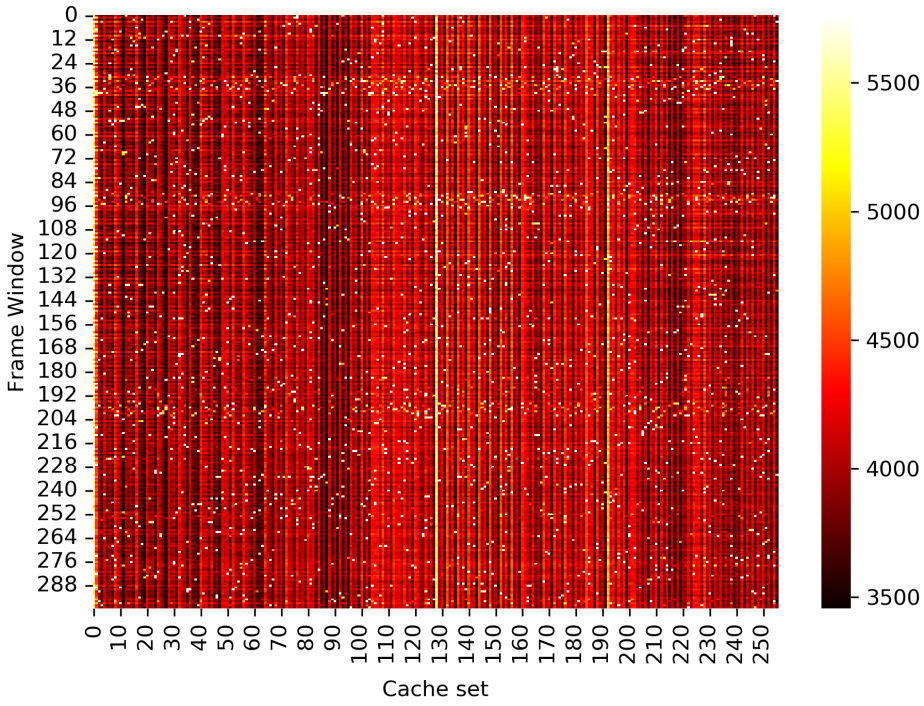

Figure 5.14: Disabling cores two and three

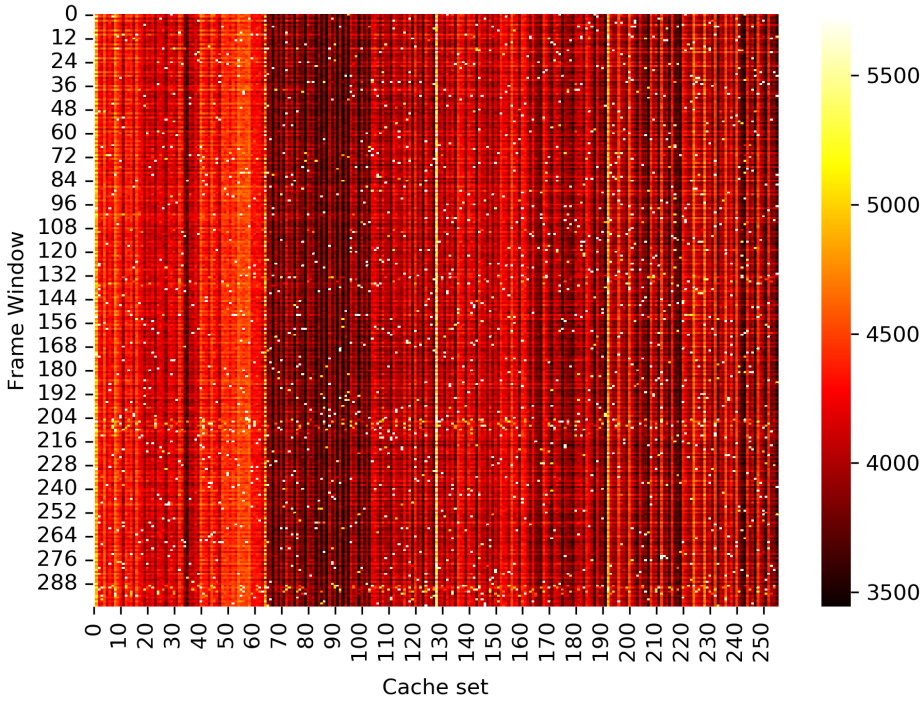

Figure 5.15: Disabling cores 2-7

Chapter 6

Next steps

6.1 Analyzing more complex victims

We are interested in performing more experiments on more complex victim programs. We have seen that the measured heatmaps from our experiments do not always align with our expectations given the behavior of the program. However, it is possible that the real signal produced by the program path is simply not readable for us in this form. All our programs so far are very basic and do not contain many instructions, this could be a reason why we rarely see any real variation over time when looking at the average heatmaps. A more complex program with highly varying program paths would hopefully show clearer differences.

We also believe that such an approach would be helpful because the cache activity would come from our victim loading program instruction rather than data. As mentioned in 5.3.2 we believe that it is possible that some hardware memory optimizations are interfering with our measurements, which leads to behavior different from what we expect.

6.2 Quantization of access times

Another interesting extension to explore is quantization. For the moment, we deal with timing matrices where each cell contains the access time of a cache set. However, these timings can vary greatly and we are vulnerable to outliers when computing an average. It would be interesting to explore the possibility of clustering these timings. In the extreme case we could produce binary matrices, where a zero represents a cache hit and one represents a cache miss.

Quantizing however does incur a loss of information so it would be interesting to try a range of different numbers of clusters instead of only producing binary matrices. We could then look at how this impacts the identifiability of the program paths' cache usage footprint.

Another benefit is that this extension has a low performance cost. We can easily train a k-clustering model on a given phone by creating a lot of training data. The learned thresholds do not depend on the program itself so they could then be used when analyzing the behavior of real victim programs.

6.3 Clustering timing matrices

We would like to be able to cluster timing matrices and, consequently, program paths. One possible approach is to create program patterns and use an image classification approach to identify program paths.

This would then require two phases. From the 40 repetitions we perform to eliminate cache noise we must first extract some sort of program pattern. Then, when evaluating a new fuzzer-generated input, we can compare it to the previously extracted program paths to determine whether or not we think this corresponds to a new program pattern, and therefore path.

This also requires defining a distance metric for our problem. We propose using the *Wasserstein distance* for this purpose.

To understand this metric, we can imagine a scenario in which we are given an arbitrary pile of dirt and the task of reshaping this pile so that it takes the shape of some target pile of dirt and a cost function that measures the effort required to move dirt around in a given way (i.e. c(x, a, b) is the cost of moving x amount of dirt from a to b). The optimal transport problem aims to find the cheapest way of achieving this and the Wasserstein distance metric is a measure of this solution [5].

In the scope of our project, the "piles of dirt" correspond to our timing matrices (where the height is now given by the access time itself). We can construct a cost function such that the cost of shifting values along the y axis (time) is very cheap but very expensive along the x axis (cache sets).The intuition here is that if we compare two timing matrices from the same program path, the total amount of cache misses should be roughly the same (and occurring in the same sets) however they are sometimes slightly offset. We would then find a relatively low cost of shifting these misses along the time axis to match our target pattern.

6.4 Cross-correlation

Another possibility is to perform column-wise comparison of the timing matrices instead of row-wise. The intuition here is that there is no strong relation between cache set A and cache set B in a given frame window. However, when looking at the entire lifespan of the victim program and focusing on a single cache set we expect to see similar behavior for the same program path.

Our idea is to view each individual cache set as a discrete-time function over time and perform cross-correlation between two different timing matrices to see how closely the behavior of each cache set matches.

Chapter 7

Conclusion

Our goal with this project is to be able to be able to efficientely and reliably differentiate program paths using only cache timing information. We believe that each program path leaves a particular footprint with its cache usage.

Our experiments and their results however show that such footprint is not as easily detactable as we hoped. The actual cache activity of our programs can wildly differ from our expectations. We believe this to be due to noise caused by other hardware activity or hardware optimizations affecting memory loads and stores.

We hope that the proposed extensions will help us in extracting a signal from our timing data.

Bibliography

[1] *Armageddon/libflush at master · IAIK/armageddon*. URL: https://github.com/IAIK/armageddon/tree/master/libflush.

[2] *Fuzzing*. Jan. 2022. URL: https://en.wikipedia.org/wiki/Fuzzing.

[3] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "Armageddon: Cache attacks on mobile devices". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 549–564.

[4] *Timing attack*. Nov. 2021. URL: https://en.wikipedia.org/wiki/Timing_attack.

[5] *Wasserstein metric*. Dec. 2021. URL: https://en.wikipedia.org/wiki/Wasserstein_metric.