Cache-Based Side-Channel Attacks in FPGA SoCs

School of Computer and Communication Sciences (IC)
Parallel Systems Architecture Lab (PARSA)
January 26, 2023

Master semester project report
by

Matteo Oldani

Under the supervision of

Dina MAHMOUD
Dr. Mirjana STOJILOVIĆ

Cache-Based Side-Channel Attacks in FPGA SoCs

Matteo Oldani

EPFL, Switzerland, matteo.oldani@epfl.ch

Abstract

This report explores the possibility of cache-based attacks on heterogeneous platforms such as the AMD-Xilinx Zynq 7000 which combines a field-programmable gate array (FPGA) with an ARM central processing unit (CPU). We built a partial key recovery attack on the advanced encryption standard (AES), which reduces the key search space from 128 to 48 bits, considering two threat models: FPGA-to-CPU and CPU-to-FPGA. In the first threat model, we assess the work done by Bossuet et al. [BB21], and we explore a new side channel, showing that a write transaction can also be used to detect timing differences between data inside and outside of the L1 cache. Finally, the attack is tested in the opposite direction, from the CPU to the FPGA, and the results are found to be consistent with the expectations. Our work thoroughly tests the advanced extensible interface (AXI) protocol and analyzes the five channels to identify timing differences. We propose further work to lower the minimum clock speed that can still detect a timing difference.

Contents

1 Introduction

System-on-chips (SoCs) are becoming more and more useful, especially due to the increasing demand for specialized hardware, where multiple chips are needed to complete a task. The paradigm of a heterogeneous SoC composed of a normal CPU coupled with an FPGA is nowadays considered a common solution to provide the end user with a high degree of parallelism while allowing customization. As mentioned in *The Zynq Book* [CEES15], heterogeneous SoCs will benefit applications from the automotive and aerospace industry to the medical one which includes fast and precise image and video processing. However, resource sharing poses some challenges in terms of the security of the whole platform. Side channels have been demonstrated, for example by Liu et al. [LYG+15], to be a critical vulnerability in the CPU-only world. In this work, we explore the possibility of cache-based attacks on heterogeneous platforms such as the Xilinx Zynq 7000 [Xil21] present on the ZedBoard. In the first part, we focus on FPGA-to-CPU attacks, and we assess the work done by Bossuet et al. [BB21]. In the second part, we expand the results with an additional side channel. Furthermore, we evaluate the security of the opposite scenario, showing a cache attack from the CPU to the FPGA.

In the rest of this report, we will give some background on cache attacks and AES, focusing on the T-Table implementation (Section 2), then present the experimental setup (Section 3) and the threat models (Section 4). We will then move on to the attacks and our findings (Section 5). Finally, we will discuss related and further work (Sections 6 and 7).

2 Background

2.1 Cache Attacks

Cache attacks are a family of attacks based on side channels widely known and explored in the CPU-only world. Bernstein et al. [Ber05] and Osvik et al. [OST06] were among the first to demonstrate cache-based side-channel exploits against cryptographic primitives. Moreover, these basic but critical attack techniques were used as building blocks for more complex and influential attacks. Meltdown [LSG+18] and Spectre [KGG+18] are examples of attacks that employ cache-side channels to exploit speculative execution vulnerabilities in Intel CPUs.

To build a cache attack, at least two primary conditions must be met. Firstly, a side channel is needed, which usually is a difference in the time required to complete a specific operation that indicates the presence or absence of some data in the cache. Secondly, the attacker needs to be able to flush or evict cache lines. The ability to control the presence of some data in the cache is required since the attacker will gather information by observing which part of memory is used by the victim and, thus, brought into the cache. More details of how those abilities can be used to mount a practical attack will be presented in Section 5.3 while explaining the attack done during the project.

Cache attacks can be classified into two categories based on shared memory between the attacker and the victim. If there is a shared memory region, the attacker can easily control the data in the cache either with flush instructions, if available, or by creating an eviction set to evict a specific portion of the cache. An eviction set is usually a set of addresses mapped to the same cache line we would like to flush, accessed with the only goal of forcing the data contained in that specific cache line out of the cache. Under this category, we find attacks such as FLUSH+RELOAD, EVICT+RELOAD, and FLUSH+FLUSH. In the other case, if the attackers do not have direct control over a specific portion of memory, they have to rely only on the eviction. However, the absence of a shared memory region poses another constraint to the attack. Indeed, the cache hierarchy of the architecture considered needs to be inclusive, and the attacker must have access to the same last-level cache used

by the victim. If both these requirements are satisfied, attacks such as PRIME+PROBE can be mounted. If the conditions are not fulfilled, the attacker will lose the ability to remove data in the cache and, in the end, lose the ability to mount the attack. A more detailed and comprehensive explanation for the attacks above can be found in work by Lipp et al. [LGSM15].

2.2 AES T-Table Implementation

The Advanced Encryption Scheme (AES) is a symmetric block cipher with a block size of 128 bits and a key that can be 128, 192, or 256 bits long. The key is expanded to build "round" keys used during the ten rounds of operations that comprise the encryption. Each round is made of four stages, namely:

1. Add round key
2. Sub-Byte
3. Shift-Row
4. Mix-Column

Fig. 1 presents a simplified overview of the steps done by the AES encryption algorithm.

Performance-optimized versions of the AES encryption algorithm, such as the T-Table implementation used in this project, use transformation tables to compute the aforementioned operation using lookup tables. In the T-Table implementation of AES, all the operations can be reduced to four lookup tables and a round is then evaluated with only 16 lookups. The following formulas will show how the AES algorithm uses the encryption tables.

$$\begin{aligned}
(x_0^{r+1}, x_1^{r+1}, x_2^{r+1}, x_3^{r+1}) &\leftarrow Te_0[x_0^r] \oplus Te_1[x_5^r] \oplus Te_2[x_{10}^r] \oplus Te_3[x_{15}^r] \oplus k_0^{r+1}, \\
(x_4^{r+1}, x_5^{r+1}, x_6^{r+1}, x_7^{r+1}) &\leftarrow Te_0[x_4^r] \oplus Te_1[x_9^r] \oplus Te_2[x_{14}^r] \oplus Te_3[x_3^r] \oplus k_1^{r+1}, \\
(x_8^{r+1}, x_9^{r+1}, x_{10}^{r+1}, x_{11}^{r+1}) &\leftarrow Te_0[x_8^r] \oplus Te_1[x_{13}^r] \oplus Te_2[x_2^r] \oplus Te_3[x_7^r] \oplus k_2^{r+1}, \\
(x_{12}^{r+1}, x_{13}^{r+1}, x_{14}^{r+1}, x_{15}^{r+1}) &\leftarrow Te_0[x_{12}^r] \oplus Te_1[x_1^r] \oplus Te_2[x_6^r] \oplus Te_3[x_{11}^r] \oplus k_3^{r+1}.
\end{aligned} \tag{1}$$

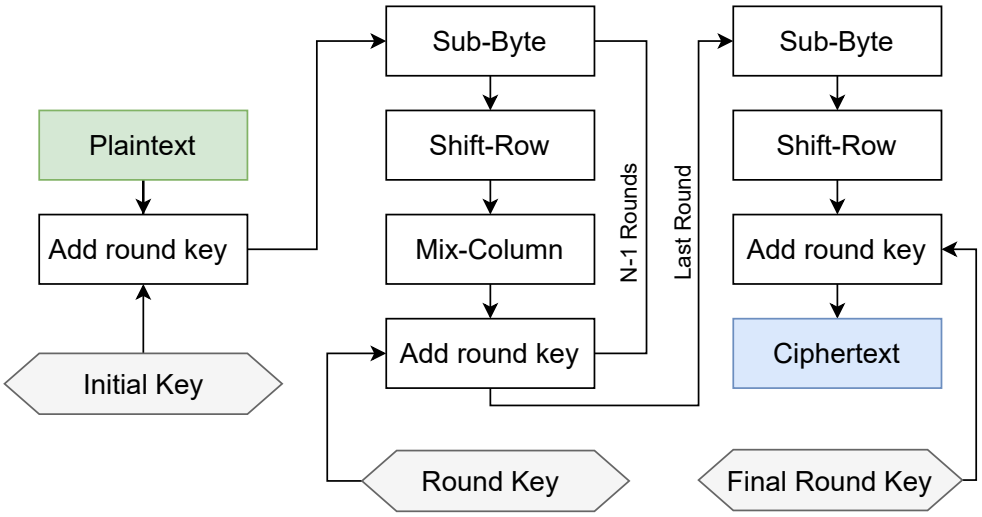

Figure 1: Simplified AES encryption steps.

Where Te_0, Te_1, Te_2, and Te_3 are the transformation tables, r represents the round and

$$x_i^0 = Plaintext_i \oplus key_i. \tag{2}$$

The T-Table implementation of AES is vulnerable to side-channel attacks, which can lead to a key recovery attack. As Gruss et al. [GMW15] explain, the lookups done in the first round of encryption depend only on the plaintext and the key. If the attacker controls or knows the plaintext, then the key bytes can be inferred by analyzing which parts of the T-Tables are in the cache. A detailed explanation of the attack implementation is given later in Section 5.3 where we will present the findings of the analyzed (and reimplemented) paper.

Before discussing the details of the attack on the T-Table implementation of AES, it is important to understand why we chose this target. While it may not be common to have a shared software library between a trusted and an untrusted application in our threat model, the T-Table implementation has become a standard benchmark for evaluating the possibility and efficiency of side-channel attacks. It is worth noting that this attack is not a result of a flaw in the implementation of AES but rather a weakness inherent in the architecture being used. The side channels used in this key recovery attack can also be adapted to target other libraries whose security may be compromised if the attacker can identify their access patterns.

3 Experimental Setup

3.1 ZedBoard

We used the ZedBoard as an experimental platform throughout the project to explore possible cache attacks in a heterogeneous SoC. The ZedBoard is a development board for the Xilinx ZYNQ-7000. The ZYNQ-7000 combines two ARMv7 Cortex-A9 cores, the processing system (PS), and an FPGA, the programmable logic (PL). The cores have a 64 KB private L1 cache each, divided into 32 KB for data and 32 KB for instructions. They share an L2 unified cache of 512 KB. By the ARMv7 specs, the two cache levels use a *non-inclusive* policy which inherently reduces the attack space as explained in Section 5.3.

Communication between the PS and PL is achieved with the AXI protocol, which is briefly explained in Section 3.2. The PS part has two master and two slave general-purpose interfaces, four slave high-performance interfaces, and one slave ACP port whose purpose is described in Section 3.3. Additional and more complete information on the board can be found in the ZedBoard Reference Manual [Avn14] and the ZYNQ-7000 Technical Reference Manual [Xil21].

The whole project is developed using the standalone operating system provided by Xilinx.

3.2 AXI Protocol

AXI stands for Advanced eXtensible Interface. The AXI protocol, defined by ARM as part of the AMBA standard, is the standard to connect different IPs. We will focus on the fourth version of the protocol, which presents three versions:

- AXI4: this is the full protocol interface and is suitable for high-performance memory-mapped devices.
- AXI4-lite: this version is used for simple memory-mapped communication.
- AXI4-Stream: this version is used for high-speed streaming data.

We are mainly interested in the standard AXI4 protocol. The protocol relies on five channels; two are used for "Read" transactions, while the remaining three are for "Write" transactions. Those channels are:

- Read Transactions:
  - Read address channel
  - Read data channel
- Write Transactions:
  - Write address channel
  - Write data channel
  - Write response channel

Each channel is a collection of signals. Every channel has a pair of VALID/READY signals, which are used to do a handshake between the master and the slave interface, which starts the transaction. Those handshakes will play a fundamental role in our side channels.

3.3 ACP Port

The Accelerated Coherency Port (ACP), as mentioned before, is one of the slave interfaces that the PS part of the ZYNQ-7000 has. As the name suggests, it is used to share data between the PS and PL parts of the board while preserving the coherency between the processor and the caches. In particular, this port is directly connected to the Snoop Control Unit, which enables cache-to-cache data transfer between different processors while ensuring cache coherence.

When an IP is connected with the ACP port, some parameters can be set as explained by the ARM Developer Guide at the following link. Among them, the most important signal, which will specify which policy to enforce, is the AxCACHE[3:0] where X can either be R or W depending on the transaction type. From the MSB to the LSB, we find: *Bufferable*, *Cachable*, *Read allocate*, and *Write allocate*. Among all the possible configurations, the two that were relevant for this project were b'0010' and b'1110'. The first one, which encodes a *cachable* but *not-allocate* transaction, is used for FPGA-to-CPU attacks. The second one, which allows the allocation policy, is fundamental in the CPU-to-FPGA scenario.

To conclude with the ACP port description, it is helpful to underline that a "write" transaction will write only to the L2 cache if issued from the PL part of the ZYNQ-7000. In this case, the SCU will take care of the coherency and invalidate eventual cache lines in the L1 caches of the CPU.

4 Threat Model

In this work, the threat model is based on multitenancy in a heterogeneous FPGA-CPU SoC. We assume that the architecture shares part of the memory hierarchy between the two parts: in our case, the L2 cache in the CPU can be coherently accessed from the FGPA. We believe that this is a real word assumption since sharing the last level cache is a practice widely adopted both by Intel [Int21] and by ARM [Lim15] platforms. Moreover, attackers need to control part of the platform, as we will discuss later, and the compromised resources they are using can be shared temporally or spatially. We focused on both FPGA-to-CPU and CPU-to-FPGA attackers. In the FPGA-to-CPU scenario, shown in Fig. 2, we assume that one of the two cores of the Cortex-A9 CPU is executing a security-relevant service. The attacker is in control of the FPGA side of the board, which needs to be controlled by

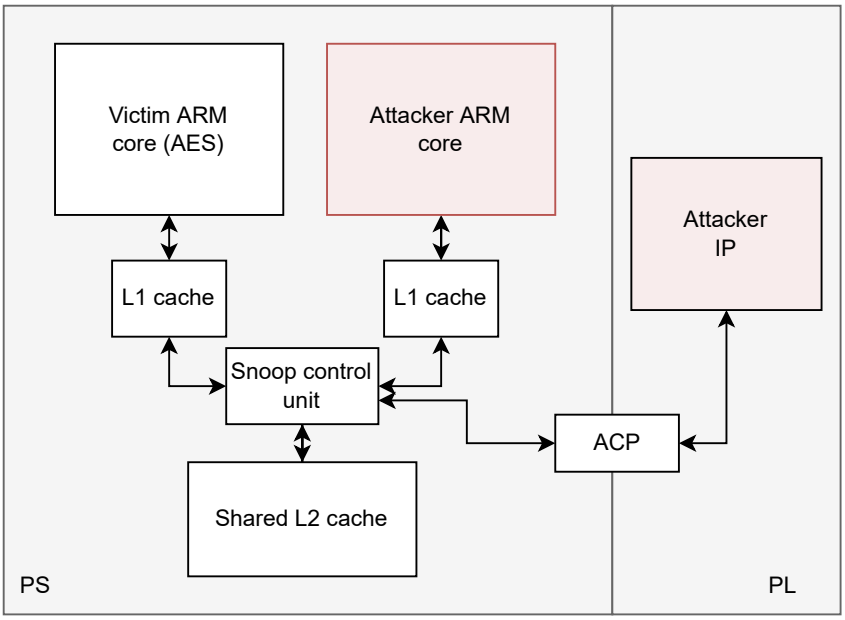

Figure 2: Threat model representing the FPGA-to-CPU attacker

the PS side of the SoC. However, despite the presence of the attacker on one core in the PS part, the attack is entirely operated from the FPGA side. Thus, CPU-implemented mitigations will fail in recognizing the attack.

In the second part of the work, we consider the CPU-to-FPGA scenario. As presented in Fig. 3, we assume a security-relevant service implemented on the FPGA side, while the attacker controls the CPU side of the board. In both cases, the cache-based side channel is exploited, assuming the victim service exposes a library with a memory shared with the attacker. In particular, we focus on a T-Table implementation of AES, assuming that the T-Tables used during encryption are shared between the victim and the attacker.

5 Exploration of Cache Attacks

In the first part of the project, we focus on evaluating and assessing the work done by Bossuet et al. [BB21]. In their work, they exploited a side channel in the AXI protocol to build a cache attack against a T-Table implementation of AES.

5.1 Timing Differences

To test for a side channel, Bossuet et al. [BB21] issued read and write transactions on addresses arbitrarily placed in or out of the cache. They then measured the time elapsed between the issuing of the transaction and the receiving of the response. They found that only the read transaction was showing the difference in timing. In particular, the number of clock cycles between the first handshake for the address exchange `(ARVALID == 1 && ARREADY == 0)` and the second handshake to signal the presence of data `(RVALID == 1 && RREADY == 1)` depends on the position of the data in memory. We tested this theory on our board and found a reliable timing difference. However, the handshake condition is partially incorrect: the proper one requires `ARREADY` to be equal to 1. Fig. 4 shows the proper handshake. For the correct handshake, we confirmed that there is a timing

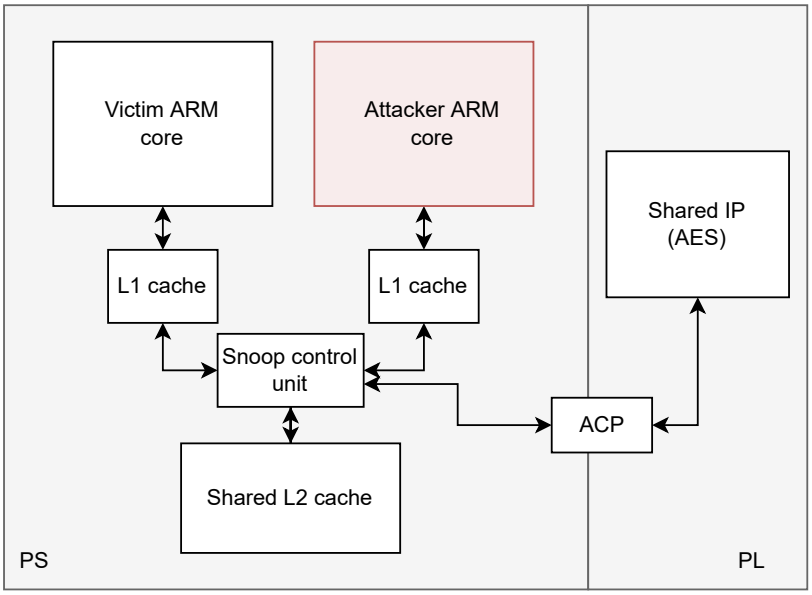

Figure 3: Threat model representing the CPU-to-FPGA attacker

difference not only between data in the cache and out of the cache but also between data in the L1 and L2 caches. In particular, as they stated in the paper, even if the difference depends on the clock speed of the PL part, starting from the default clock of 100 MHz, there is already a reliable timing difference of one clock cycle. In our case, a read done with the address in the L1 cache leads to a time of 12 cycles, while a read done with the address in the L2 cache leads to 13 cycles. It is important to specify that if the clock is below 100 MHz, the timing difference is not visible anymore. After confirming the timing difference with the integrated logic analyzer (ILA), we built a custom timer to count the number of clock cycles between the two handshakes so that our attacker IP could use the side channel to build the attack. Once the timer has detected and timed a complete read, it will set the valid signal to '1' and the counter signal to the number of clock cycles counted. The attacker IP, after recording that the valid signal is asserted high, will read the value in the counter signal. The relevant part of the timer code is shown below.

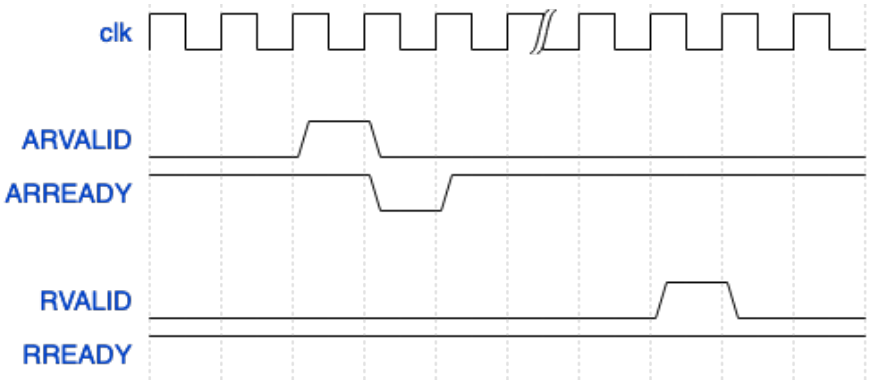

Figure 4: AXI read handshake

```
if rising_edge(Clk) then
    counter_valid <= counter_valid_i;
    counter <= std_logic_vector(ticks);

    if awready = '1' and awvalid = '1' then
        ticks <= (others => '0');
        counter_valid_i     <= '0';
        countable           <= '1';
    elsif rready = '1'and rvalid = '1' then
        counter_valid_i <= '1';
        countable <= '0';
    end if;

    if countable = '1' then
        ticks <= ticks + 1;
    end if;

end if;
```

5.2 Eviction

To fulfill the second condition for a cache attack, Bossuet et al. [BB21] found a way to evict an address from the L1 cache. Since the ACP port can only write to the L2 cache, they noticed that issuing a write from the PL to an address present in the L1 cache causes the invalidation of that cache line from the L1 cache. However, to not change the value of the address in memory, they set the `WSTRB` signal to b'0000'. That signal specifies which byte of the write transaction should be considered valid. Setting the signal to 0 defines a write in which all bytes are invalid. This eviction method was also presented by Kim et al. [KKH+17]. After testing that the eviction method was working by running simple experiments where we checked with the ILA to see a timing difference between a read done before the eviction and one done after, we developed a simple evicter. To build a working attack, we needed to be able to dynamically set the `WSTRB` so that we could both make actual writes and eviction writes (which means with the `WSTRB` set to b'0000'). We decided to implement a multiplexer that, based on the attacker's IP choice, could set the

`WSTRB` to the actual value that the AXI protocol would have used or to b‘0000’.

5.3 The Attack

The attack presented in the paper is an already explored partial key recovery attack on AES. Even if a full key recovery has been proven possible, as shown by the work of Briongos et al. [BMGM19], the one used in the paper only reduces the key search space from 128 to 48. As will be clear later, the attack finds the five uppermost bits of every byte that composes the key. The attacker will craft 16-byte long plaintexts and focus only on one byte at a time while randomizing the others. The attacker aims to find which value of that byte always brings into the cache the first entry of the T-Table used during the encryption. Indeed, following Equation 2, the result of the XOR between the byte considered and the relative key byte will be the T-Table index. If the T-Table is accessed in position 0, then the value of the plaintext byte is equal to the value of the key byte. However, the best we can do is to find the set of plaintexts that will give a value belonging to the same cache line as the first element of the table (the one corresponding to the value 0). Since every element of the table is an uint32_t and the cache line in our architecture is 32 bytes long, the set of plaintexts is composed of 8 possible candidates. In particular, we will have eight contiguous values with the five most significant bits fixed, which explains the key space reduction from 128 bits to 48. Before moving to the actual algorithm steps, it is worth mentioning that it is required to use all four tables to recover the whole key. The first encrypting round makes 16 table lookups: four in each table using always a different key byte. The algorithm’s steps to find the set of plaintexts containing the key’s values are as follows:

1. Consider a plaintext of 16 bytes. For every byte, denoted as ptxt[i], test all the possible byte values while setting the remaining bytes in the plaintext at random.

2. Every time, check if the table responsible for the current key byte has its first 8 bytes loaded in the cache. If so, this plaintext value is kept in the candidates. Otherwise, it is removed.

3. Iterate the process until, for every key byte, the set of candidates is reduced to only eight values.

5.4 Attack Results

We tried and mostly managed to replicate the results proposed in the reference paper. However, before moving to the actual results, we will explain the data collection method and what those results will display. The results will only exhibit part of the attack described above since we are now only focusing on the first byte of the plaintext: ptxt[0]. This means that only the recovery of the first byte of the key is demonstrated. For each value for the first byte of the plaintext, we have tested all the possible indices of Te_0. For each byte value, we issued 256 encryptions and, after each encryption, checked which indices of the T-Table were in the L1 cache, thus, accessed. These steps are presented in Algorithm 1. Fig. 5, which has on the x-axis all the possible plaintext values and on the y-axis all the T-Table indices, displays the results. The colors in the figure give an indication of how many times that particular address was found in the L1 cache during data collection. The darkest color represents 256 accesses, which is the maximum possible according to Algorithm 1. The diagonal pattern shows that the first byte of the key was set to 0. Indeed, each address of the first T-Table (Te_0) is consistently brought into the cache when the plaintext is set to the same value as the index of the address. Recall that the table is accessed in position $key[0] \oplus ptxt[0]$ in this case. Our results are consistent with the ones presented by the paper, as can be seen by comparing Fig. 5 to Fig. 6.

Algorithm 1: Data collection code for the plots.

Input: The key for the encryption
Input: The results array which will be filled by the FPGA
Output: Prints the matrix to be plot
foreach *value in range(256)* **do**
  foreach *j in range(256)* **do**
    plaintext[0] = *value*
    plaintext[1 :] = random(256)
    ciphertext = encrypt(*key*, *plaintext*)
    results = FPGA_time_and_evict()
    foreach *i in range(256)* **do**
      if *results*[*i*] == *12* **then**
        attack_results[i][value] += 1
print(*attack_results*)

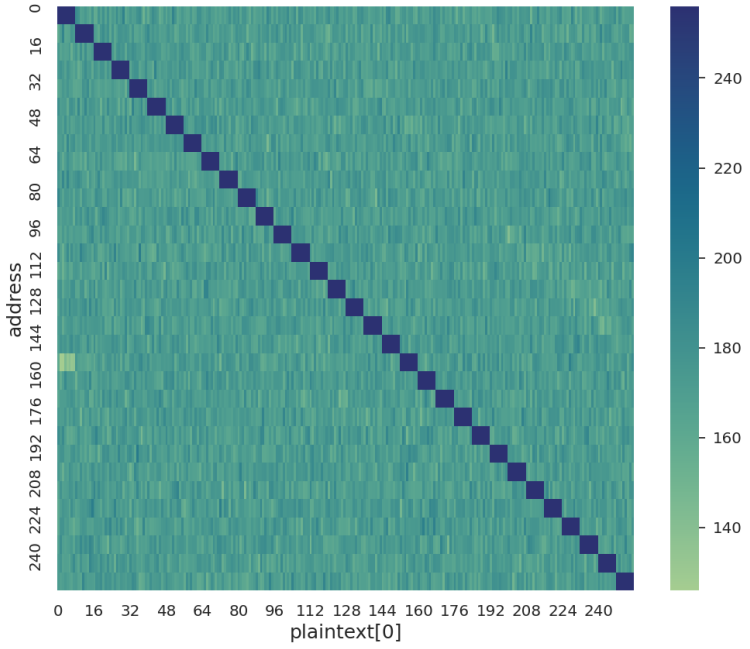

Figure 5: Results of our implemented attack with key set to 0

To conclude the analysis of the results, we have to point out that we did not manage to replicate the plot presented in Figure 11 in the reference paper [BB21]. The authors claimed that the first byte of the key was set to `0x51`. However, according to our experiments, the image will be correct only if we assume one of their axes has values in the opposite order. For example, if we consider the bottom-left corner, we should find "P_0" and "Address" both either set to 0 or set to 255. When the value of the first byte of the plaintext is set to 0, then the address brought in the cache should be the one at index `0x51` since tables are consistently accessed by XORing the key with the plaintext. However, this behavior is registered when the plaintext is set to 255. We believe that the Y-axis is wrongly labeled

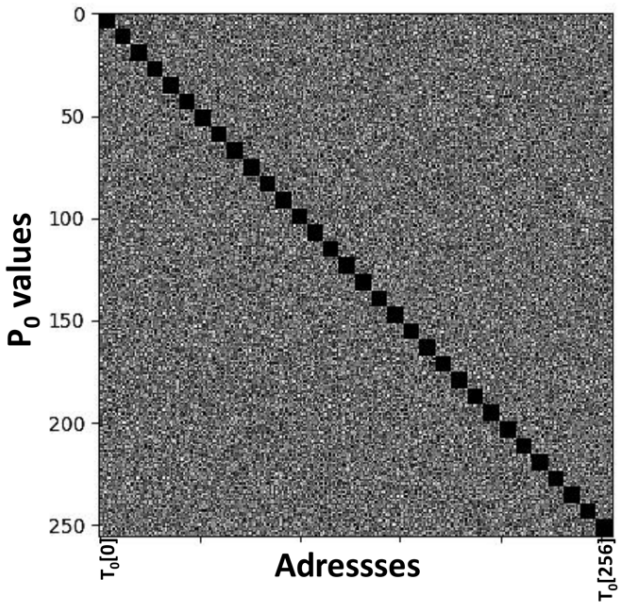

Figure 6: Results reported by the reference paper [BB21] with key set to 0

in the reference paper [BB21].

5.5 New Side Channel

In the second part of the project, we focused on expanding our knowledge of the board and finding new attacks. Despite the reference paper explicitly stating that only the read transaction showed timing differences, we decided to test the AXI protocol again. In particular, we focused on the write transaction. The idea was that something similar to a FLUSH+FLUSH attack might be possible. Indeed, in that attack, the timing difference is in the execution of the flush instruction, which is faster if the data does not need to be flushed. Since a write issued from the PL has to invalidate the line in case data is in the L1 cache, we thought it would have taken more time than a write on data not in the L1 cache. This has been proven true; tests showed that writes to addresses in the L1 cache are slower. We managed to put the threshold, with a PL clock speed of 125 MHz, to 14 cycles: we consider an address inside the L1 cache if a write to that address takes more than 14 clock cycles; it is in L2 or out of cache otherwise. However, the signal to be monitored to time the write transaction differs from the one used for the read transaction. The first handshake starting the timer is now composed by `AWVALID == 1 && AWREADY == 1` and the second handshake, indicating the end of the transaction is made by `BVALID == 1 && BREADY == 1`. Those last two signals are taken from the response channel and not from the data channel as it was done with the read transaction handshake. The same timer used for the previous side channel was deployed, only with different input signals. With this new side channel, we built the same attack to see if we could get the same results. We replicated the same setting as for Fig. 5, and as can be seen from Fig. 7, the attack worked as expected. For this new side channel, we also implemented an optimized version of the attack. Instead of reading all the addresses, we only tested one address for each cache line. This will reduce by a factor of 8 the number of reads required. Moreover, for every single iteration of the IP, the whole T-Table is checked, resulting in 255 fewer

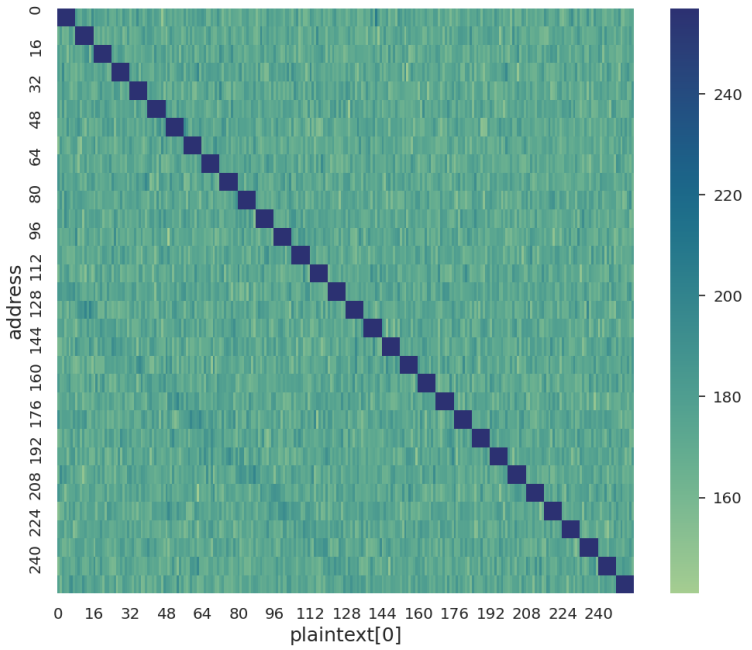

Figure 7: Results of our implemented attack with key set to 0 using the write side channel

iterations needed for each possible value of the plaintext in position 0. Fig. 8 shows the result obtained from the optimized version of the attack, which clearly still shows the same pattern. To conclude, with this attack, we also tested the eviction without setting the `WSTRB` to b'0000'. We demonstrated that also "normal" write will evict the cache line; however, to not overwrite the correct value present in the T-Table, we had to read it in advance and issue a write with the value previously read.

5.6 CPU-to-FPGA

In the last part of the project, we explored the possibility of mounting the attack in the opposite direction. This possibility was already discussed by Weissman et al. [WTM+19] in a slightly different setting. Indeed, we do not have a cache dedicated to the FPGA part. Instead, the FPGA still uses the L2 cache through the ACP port. In particular, we developed a custom IP that implements AES encryption using T-Tables. Those tables are stored in the DRAM memory of the ZYNQ-7000 and later brought into the cache. As before, we need a way to (1) flush the cache and (2) measure a timing difference for data in and out of the cache. To achieve the first goal, Xilinx has implemented multiple library functions to flush the caches. The library we are referring to can be found in the standalone folder of the platform under the name `xil_cache.h`. In particular, we used the `Xil_DCacheFlush` function to ensure that all the T-Tables entries were out of the cache before the encryption. To fulfill the second requirement, we developed three different timers, with increasing precision, after we noticed that the one already present in the standalone libraries was not working as expected. As explained in the Cortex-A7 MPCore Technical Reference Manual [Lim13], we can build a timer using a co-processor counter which is incremented at clock speed. The code of the final timer is showed below:

```
static __inline__ unsigned time_read_mem(unsigned char * addr){
```

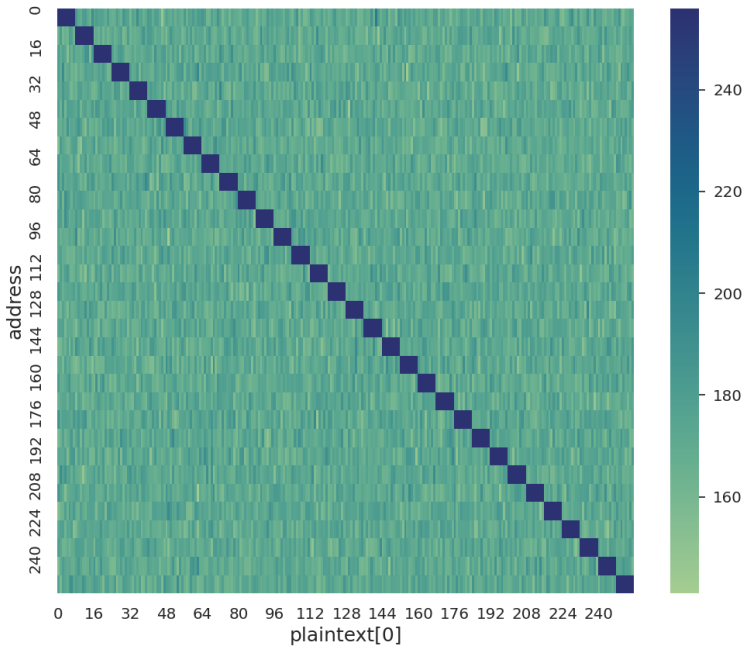

Figure 8: Results of our implemented optimized attack with key set to 0 using the new side channel

```
        unsigned cc;

        __asm__ __volatile__("mcr p15, 0, %0, c9, c12, 2" :: "r"(1<<31));
        __asm__ __volatile__("mcr p15, 0, %0, c9, c12, 0" :: "r"(5));
        __asm__ __volatile__("mcr p15, 0, %0, c9, c12, 1" :: "r"(1<<31));
        __asm__ __volatile__("ldr r9, [%0]" :: "r"(addr) : "r9");
        __asm__ __volatile__("mov r8, r9" : : : "r9");
        __asm__ __volatile__("mrc p15, 0, r10, c9, c13, 0");
        __asm__ __volatile__("mov %0, r10" : "=r"(cc) : : "r9");

        return cc;
}
```

The increased precision of our last iteration of the timer help us find a reliable threshold between data in and out of the cache. We noticed that a read operation on the CPU side would take less than 40 clock cycles only if the address was already present in the L2 cache. We recall that the FPGA cannot write to the L1 cache. As a reference, we found that accesses to the L1 cache require around 15 clock cycles. Fig. 9 reports the attack results, proving that the attack was successful in the CPU-to-FPGA case. This confirms the CPU's ability to distinguish between data brought into the L2 cache by the FPGA and data not in the cache. As mentioned in Section 3.3, we had to change the cache policy of our custom IP to make the attack successful. In particular, we enabled read and write allocation. To conclude the attack analysis, it is worthwhile to note the differences between Fig. 5 and Fig. 9's results. The picture looks noisier, and the pattern does not highlight only the diagonal. This is explained taking into account the work of the prefetcher. In particular, the prefetcher loads into the cache not only the cache line containing the address read from the CPU but also the cache line right after it. This is

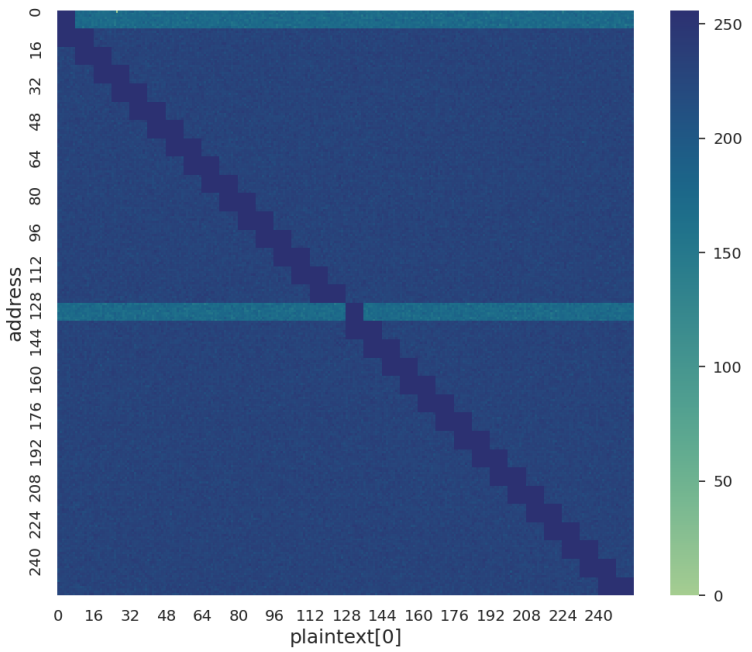

Figure 9: Results of our implemented attack from the CPU to the FPGA with key set to 0

reflected in the picture as if two consecutive rows are accessed while only the first one is used. This hypothesis is confirmed by noticing that the first row is not affected by this behavior since, to be prefetched, it would require the line before it to be accessed. However, no addresses before the T-Table is read. Moreover, the brighter line at index 128 shows that the prefetcher does not work across pages. During the data collection for Fig. 9, Te_0 had a starting address of `0x103E00`, thus, after 128 `uint32_t` values, the address is exactly `0x104000`.

Before moving to the next section, we present a particular pattern discovered due to a bug. We ran the attack without the read allocation policy enabled; thus, data was not brought into the cache. We then timed the accesses to the memory location we were interested in and noticed that some were constantly slower. Fig. 10 displays this pattern. In particular, we timed each address ten times and summed the time required to access it. The darker dots present in Fig. 10 are the slow addresses. Unfortunately, due to time constraints and being out of the project scope, we did not have time to investigate further. However, we thought it was an interesting pattern that deserved to be reported.

6 Related Work

Besides the reference work done by Bossuet et al. [BB21] analyzed in the first part of the project, the main works done on this topic have been accomplished by Weissman et al. [WTM+19] and by Sepúlveda et al. [SGZS21]. In particular, Weissman et al. [WTM+19] present an analysis of the security on Intel-based architectures which provide inclusive caches, opening the possibility to PRIME+PROBE attacks. Sepúlveda et al. [SGZS21] further improves the efficiency of cache attacks against AES by exploiting contention on the bus-based SoC communication to execute the attack exactly after the first round of

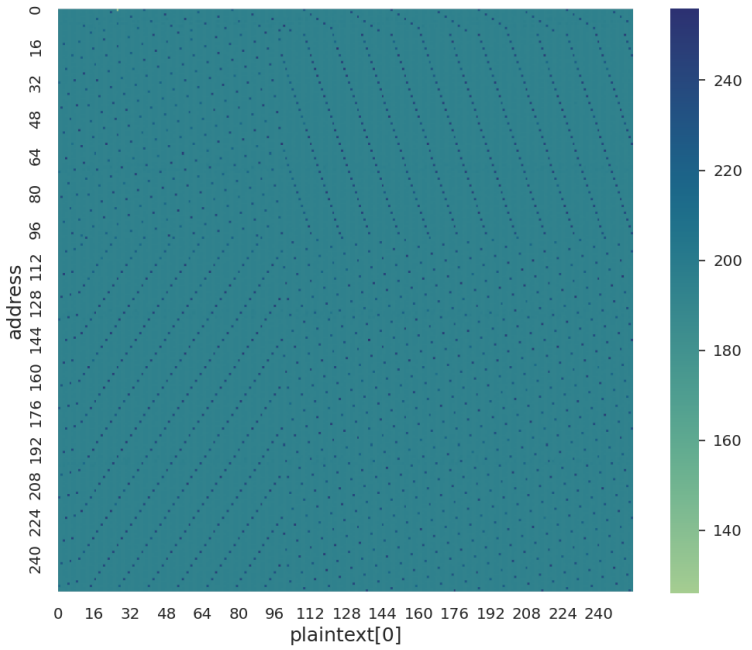

Figure 10: Memory pattern showing that some addresses are constantly slower than others

encryption.

7 Conclusion & Future Work

This work explored the possibility and feasibility of cache attacks on heterogeneous platforms. We assessed and replicated the results presented by Bossuet et al. [BB21]. We further expanded their work on the AXI protocol finding a new side channel that proved helpful for building a cache attack. Moreover, we tested another threat model, building an attack from the CPU to the FPGA. We believe that the ACP port, which uses the AXI protocol, has now been thoroughly tested and no further work can be done to find a new side channel. In particular, all five channels have now been analyzed and, on both read and write transactions, a side channel has been found. This work can be extended by lowering the minimum clock frequency at which a timing difference can still be detected. In particular, if after the invalidation of the cache line from the L1 cache, the attacker IP causes an eviction from L2 (building an eviction set), then a read transaction issued from the PL will take even longer because it will be located entirely out of the cache.

References

[Avn14] Avnet. Zynq™ evaluation and development hardware user's guide, June 2014. Available: https://files.digilent.com/resources/programmable-logic/zedboard/ZedBoard_HW_UG_v2_2.pdf.

[BB21] Lilian Bossuet and El Mehdi Benhani. Performing cache timing attacks from the reconfigurable part of a heterogeneous SoC—an experimental study. *Applied Sciences*, 11:6662, July 2021.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. April 2005.

[BMGM19] Samira Briongos, Pedro Malagón, Juan-Mariano Goyeneche, and José Moya. Cache misses and the recovery of the full AES 256 key. *Applied Sciences*, 9:944, March 2019.

[CEES15] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book*. Strathclyde Academic Media, August 2015.

[GMW15] Daniel Gruss, Clémentine Maurice, and Klaus Wagner. Flush+flush: A stealthier last-level cache attack. arXiv, 2015.

[Int21] Intel. Developer guide, 2021. Available: https://www.intel.com/content/www/us/en/develop/documentation/tcc-tools-2021-3-developer-guide/top/real-time-configuration-and-optimization-tools/cache-allocation/cache-architecture/cache-in-11th-gen-intel-core-processors.html.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. arXiv, May 2018.

[KKH+17] Minsu Kim, Sunhee Kong, Boeui Hong, Lei Xu, Weidong Shi, and Taeweon Suh. Evaluating coherence-exploiting hardware trojan. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 157–162, March 2017.

[LGSM15] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Armageddon: Last-level cache attacks on mobile devices. arXiv, 2015.

[Lim13] Arm Limited. Cortex™ - A7 MPCore technical reference manual, 2013. Available: https://developer.arm.com/documentation/ddi0464/latest/.

[Lim15] Arm Limited. ARM cortex-A series programmer's guide for ARMv8-A, 2015. Available: https://developer.arm.com/documentation/den0024/a/Caches.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. arXiv, January 2018.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006*, August 2006.

[SGZS21] Johanna Sepúlveda, Mathieu Gross, Andreas Zankl, and Georg Sigl. Beyond cache attacks: Exploiting the bus-based communication structure for powerful on-chip microarchitectural attacks. *ACM Trans. Embed. Comput. Syst.*, 20(2), March 2021.

[WTM+19] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient rowhammer on heterogeneous FPGA-CPU platforms. arXiv, December 2019.

[Xil21] Xilinx. Zynq-7000 SoC technical reference manual, April 2021. Available: https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM.