Performing Cache Timing Attacks from the Reconfigurable Part of a Heterogeneous SoC—An Experimental Study

Lilian Bossuet* and El Mehdi Benhani

Laboratoire Hubert Curien UMR 5516, CNRS, Jean Monnet University, 42000 Saint-Etienne, France; benhani.elmehdi@gmail.com
* Correspondence: lillian.bossuet@univ-st-etienne.fr

Abstract: Cache attacks are widespread on microprocessors and multi-processor system-on-chips but have not yet spread to heterogeneous systems-on-chip such as SoC-FPGA that are found in increasing numbers of applications on servers or in the cloud. This type of SoC has two parts: a processing system that includes hard components and ARM processor cores and a programmable logic part that includes logic gates to be used to implement custom designs. The two parts communicate via memory-mapped interfaces. One of these interfaces is the accelerator coherency port that provides optional cache coherency between the two parts. In this paper, we discuss the practicability and potential threat of inside-SoC cache attacks using the cache coherency mechanism of a complex heterogeneous SoC-FPGA. We provide proof of two cache timing attacks Flush+Reload and Evict+Time when SoC-FPGA is targeted, and proof of hidden communication using a cache-based covert channel. The heterogeneous SoC-FPGA Xilinx Zynq 7010 is used as an experimental target.

Keywords: cache timing analysis; side-channel attack; covert channel; heterogeneous SoC security

1. Introduction

Due to the need to reduce the technology and to meet market demand, the heterogeneous system-on-chip (SoC) is becoming increasingly complex as it integrates more and more functionalities including processor cores, memory, third-party hardware IPs, and reconfigurable hardware (i.e., FPGA) for hardware acceleration. This has raised awareness of the need to protect the SoC from security failures particularly when the SoC is shared in a cloud and even when software protections are available on the SoC [1]. Indeed, some parts of the SoC or some software applications running on the SoC may be malicious and try to perform inside-SoC attacks by exploiting the two threats called side-channel analyses and covert channel communications.

Side-channel analyses are passive attacks widely used in cryptographic engineering. They make it possible to retrieve secret information (such as cipher secret keys) with relatively few physical measurements sometimes even using inexpensive equipment. Side-channel analysis works even when the algorithm is shown to be robust against algebraic cryptanalysis. Most of the dynamic characteristics of both hardware and software implementations of cryptographic primitives can be used for side-channel analysis: computation time, cache and memory access time, power consumption, electromagnetic radiation, optical radiation, etc. These physical quantities are thus widely exploited during side-channel analysis aimed at understanding the behavior of the circuits (or in order to discover the secret information they contain, such as the secret keys required by the encryption/decryption process) [2]. Many recent works, [3–6], suggest embedding an information leakage sensor inside the SoC to be able to perform physical side-channel analyses without the need for external measurements. These works use the sensitivity of the SoC power distribution network to the power supply fluctuations [3–6]. Figure 1 is a conceptual view of inside-SoC side-channel analysis. In this figure, the attacker (i.e.,
the malicious process/IP) is depicted using a stethoscope to show that performing a side-channel analysis required a diagnosis of the information leak. Indeed, the attacker first measures the physical side-channel information before analyzing it and locating the secret information. In this paper, we address the possibility of performing intra-SoC analysis of the access time of share cache memory (also called cache timing analysis).

Figure 1. Conceptual view of an inside-SoC side-channel analysis.

Covert channel communications allow data to be transferred between two entities (software applications, processor cores, memory, hardware IPs, etc.) that are not authorized by the security policy or by design, to communicate and/or exchange secret information. In general, covert channel communication involves a sender process that transfers valuable information to a receiver process that decodes it and uses it for malicious purposes. Most often, physical, logical or software separation/isolation avoids direct access between the sender and the receiver. The sender has access to the secret information via authorized or unauthorized access. Figure 2 is a conceptual view of inside-SoC covert channel communications. In this figure, the sender and the receiver use the same code, the receiver can use a precise measurement of the information in the covert channel before decoding it to obtain the secret information sent by the sender. Many methods to create inside-SoC covert channels can be found in the literature, and most use previously presented physical side channels including the SoC power management and power distribution systems [7–9]. In this paper, we address the possibility of performing unauthorized intra-SoC communication using the access time of share cache memory (also called cache-base covert channel communication).

Figure 2. Conceptual view of an inside-SoC covert channel.

To perform a cache timing attack (using a cache timing analysis or cache-based covert channel communication), an attacker (malicious third-party hardware IPs or software applications) has to fulfill two main conditions. The first condition is to distinguish a cache miss from a hit to understand if targeted data or an instruction are present in the cache memory or not [10–13]. Indeed, as presented in Figure 3, the attacker wants to know if the victim process (or the sender process in the case of a covert channel) has access to specific information stored in the main memory. To be able to distinguish a cache miss from a hit, the attacker can measure access time to the targeted data in the memory system or use a performance counter unit, like the performance monitor unit (PMU) in ARM processors. The second condition is to be able to evict (flush) cache lines. Indeed, the
attacker has to periodically flush part of the cache to be sure to detect specific access to main memory data or an instruction by the victim process (or the sender process in the case of a covert channel).

Modern heterogeneous SoC-FPGAs (such as Xilinx Zynq or Intel Cyclone V) are equipped with a cache coherency port called the accelerator coherency port (ACP) that connects the master interfaces of the hardware accelerator with the cache memory system. This article presents the methods used to fulfill the two previously mentioned conditions required to perform a cache timing attack from the programmable logic part of an SoC-FPGA.

This paper starts, in Section 2, by reviewing related works. Section 3 presents the technical background required to understand and implement the attacks presented in the article. Section 4 presents a method to measure the time needed to access consistent data from the programmable logic part of an SoC-FPGA and a method to evict a cache line from the same part. Mastery of these two methods is the sine qua non-condition for the implementation of a cache timing attack. Finally, Section 5 provides proof of the practicality of the cache attacks.

2. Related Works in SoC-FPGA

Some 7010 reports of the malicious use of cache coherency protocols in the case of SoC-FPGA can be found in the literature but it is very limited. Kim et al. [14] used cache coherency between the programmable logical part and the processing system of the SoC to slow down the execution of a CPU program. They used a hardware Trojan that continuously injects memory transactions, which increases the miss rate in the L1 data cache. Chaudhuri [15] presented three possible types of attack (direct memory access attacks, cache-timing attacks, and Rowhammer attacks) that can exploit the optional cache
coherency between the programmable logic part and the processing system. Like [14] and [15], in the present work, we make malicious use of the optional cache coherency in an SoC-FPGA. Then [16] presents an application of cache attack at the NoC level. For the first time, we rely on the AXI bus signals to distinguish a cache miss from a cache hit from the programmable logic part using the ACP presented in the following section. Moreover, for the first time, our work targets SoC-FPGA protected by ARM TrustZone technology.

3. Technical Background

This section presents the technical background required to understand and implement the attacks presented in the rest of this article. The experimental platform used in this work is a Xilinx Zynq-7010, but the concept presented is compatible with all enabled-TrustZone SoC-FPGA.

3.1. Experimental Platform and Design

The Xilinx Zynq-7010 is an SoC-FPGA that includes a dual-core ARM Cortex-A9 processor, a 4-way L1 set-associative cache for instructions, and one for data, each 32 KB. The Xilinx Zynq-7010 also has an 8-way set-associative L2 cache (512 KB in size) and a cache line length of 32 bytes. The L2 cache is the last level cache (LLC) for the Xilinx Zynq-7010 and is shared between the two ARM Cortex-A9 cores and the master interface in the hardware accelerator connected to the accelerator coherency port (ACP) presented in the following sub-section.

Figure 4 shows the experimental design implemented in the Xilinx Zynq-7010 SoC [17] for this work, and Figure 5 shows the memory hierarchy of this SoC. The hardware IPs of the programmable logic part of the SoC-FPGA are partitioned in two: secure IPs (in green in Figure 4) and non-secure IPs (in red in Figure 4), using the advanced extensible interface (AXI) functionality. Both IPs have direct access to the memory using the ACP. In the processing system, each core of the ARM processor is dedicated to a world, the secure ARM core (in green in Figure 4) runs critical applications, the non-secure ARM core (in red in Figure 4) runs normal applications. The external memory is also partitioned into a secure region (in green in Figure 4) and a non-secure region (in red in Figure 4) using the TrustZone configuration registers (called TZMA). The secure region of the external memory stores critical applications and the non-secure region contains the rest of the applications. More details about this implementation can be found in [18].

Figure 4. Experimental design (secure memory region/core/IP are in green, non-secure in red) implemented in a Xilinx Zynq SoC-FPGA.
3.2. Accelerator Coherency Port (ACP)

In a heterogeneous SoC-FPGA such as Xilinx Zynq, the ACP is defined as a slave interface. It is used by hardware accelerators to access the external memory with optional cache coherency. Figures 4 and 5 show that the ACP is connected to the snoop control unit (SCU) [17] that controls cache coherency between the master interfaces of hardware accelerators, the L1 and L2 cache. From a system point of view, the ACP allows the FPGA to compete with the ARM cores for memory access using the following process:

- During an ACP write request issued by a master interface, the SCU checks the existence of the targeted data in the different levels of the cache memory. If they are present, the SCU cleans and invalidates the appropriate cache line and sends a request to update the data in memory.
- During an ACP read request, if the data reside in the cache memory, whether they are invalidated or not, the data are returned from the cache memory to the master interface. Otherwise, the data are transmitted directly from the external memory to the master interface.

In general, a master interface connected to the ACP can read coherent memory directly from the L1 and L2 caches but cannot write directly to the L1 cache. The coherency of an ACP request is controlled using $\text{AxCache}[3:0]$ and $\text{AxUser}[4:0]$ signals of the ACP that are detailed in the following section.

3.3. $\text{AxCache}[3:0]$ and $\text{AxUser}[4:0]$ Signals

The $\text{AxCache}[3:0]$ and $\text{AxUser}[4:0]$ signals (for each signal $x = R$ for read or $x = W$ for write) of the ACP control the coherency access of a request. The $\text{AxUser}[4:0]$ signal is composed of the following bits, Shared bit $\text{AxUser}[0]$ and the $\text{AxUser}[4:1]$ bits, which control the write strategy (write-back, write-through, etc.) adopted by the request. The
AxCache[3:0] signal also controls the write strategy adopted by the request. It is composed of the following bits: in-buffer bit AxCache[0], cacheable bit AxCache[1], read allocate bit AxCache[2] and write allocate bit AxCache[3].

According to Xilinx recommendations [13], AxUser[0] and AxCache[1] bits must be set for a coherency request. Other import signals are the protection signals ARPort[2:0] and AWPort[2:0]. These signals are important for the security of the system and are detailed in the following section.

3.4. AxPort[2:0] Signal

The AxPort[2:0] signal (where x = R for read and x = W for write) is an access authorization signal that protects the slave interfaces from malicious requests. It is composed of the following bits, privilege level access bit AxPort[0], request security status bit AxPort[1] and access type bit AxPort[2].

To carry out our cache attacks, we only focus on the request security status bit AxPort[1]. In a system that incorporates ARM TrustZone technology, this bit is used to propagate in the AXI bus the security status of the request that is fixed by the request issuer (a master interface of a hardware accelerator or the processing system). The arbiter of the AXI bus uses the AxPort[1] bit to protect secure IPs from non-secure requests by rejecting the communication request and rising errors in the bus. The security status of the request according to the value of AxPort[1] bit is:

- **AxPort[1] = '1':** The request is non-secure and can only access non-secure system resources.
- **AxPort[1] = '0':** The request is secure and can access all system resources.

In an enabled-TrustZone SoC-FPGA, a master interface of a hardware accelerator exploited by a hardware Trojan represents a major threat to the entire system [1]. Moreover, this scenario is particularly credible when the slave interfaces (including the ACP) of the processing system part are not configured to deny access to secure regions of the main memory from the programmable logic part, which is often the case.

In the rest of the article, we presume that AxUser[0] and AxCache[1] bits are set, and that the hardware accelerator of the programmable logic part has access to secure memory regions.

4. Elements of the Attacks

This section presents the methods based on the AXI bus signals used to fulfill the two main conditions of a cache attack from the programmable logic part.

4.1. First Condition: Be Able to Differentiating a Cache Miss and Hit from the Programmable Logic Part

The method presented in this section uses the AXI bus signals connecting the master interface and the ACP to measure the access time and then distinguishes between a cache miss and hit. The rest of this section presents the AXI bus channels that leak information about the presence or absence of data in the cache and the method used to measure the access time.

In most SoC-FPGA, the AXI bus uses five channels to connect a master interface and a slave interface. These five channels are the read address channel, the read data channel, the write address channel, the write data channel, and the write response channel. Each channel uses a VALID and READY handshake signal pair to signal when the receiver is ready to process data, and to signal when valid data are ready on the bus. The scenarios for a coherency request issued by a master interface are as follows:

- For a coherency write request, the master interface starts by sending the targeted address in the write address channel followed by the data to be written in the write data channel. Once the data are received by the ACP, this port sends back a response to the master interface using the write response channel.
• For a coherency read request, the master interface starts by sending the address to be read in the read address channel. Then, the ACP sends the data back to the master interface using the read data channel.

In order to find out which AXI bus channels to use, we performed the following experiment. We issued some read and write requests from the master interface to measure the time elapsed between the launch of the request and the reception of the response. From time to time, we evicted the address targeted by the request. As a result, we observed that for a write request, the time elapsed between two handshakes of the write address channel and the write response channel does not vary with the presence or absence of the data in the L1 or L2 cache. Therefore, it is not possible to distinguish a failure from success during a write request. For a read request, we observed that the time elapsed between two handshakes of the read address channel and the read data channel depends on whether or not the data are present in the L1 or L2 cache.

In order to validate our observation, we used Xilinx Vivado hardware debug tools. Figure 6 shows that the time elapsed between the handshake of the read address channel (\(AXI\_ARVALID == \text{'1'} \& \& AXI\_ARREADY == \text{'0'}\), blue line in Figure 6) and the handshake of the read data channel (\(AXI\_RVALID == \text{'1'} \& \& AXI\_RREADY == \text{'1'}\), purple vertical line in Figure 6) is shorter, if the data are present in the cache, and is otherwise longer.

Figure 6. Read request, cache miss followed by a cache hit captured using Xilinx Vivado hardware debug tools.

Figure 7 is a histogram of a number of read requests when the targeted address is evicted one time out of two. The histogram shows that we can set a threshold that distinguishes the access time of a cache miss and the access time of a cache hit from the programmable logic part using our method. The histogram includes no errors because we used a standalone application for our experiments that did not cause a big miss rate.

Figure 7. Histogram of cache miss and hit of a consistent read request (master interface frequency = 250 MHz).
To sum up, to distinguish between a cache miss and hit in the programmable logic part, an attacker issues a read request and measures the time that elapses between two handshakes of the read address channel and the read data channel.

4.2. Limitation of the Proposed Method of Differentiating a Cache Miss and a Cache Hit

In the experiment we conducted on Xilinx Zynq-7010, we observed that the time elapsed between two handshakes during a read request depends not only on the presence or absence of the data in the cache but also on the frequency applied to the master interface.

Figure 8 shows that during a read request, the number of clock cycles between two handshakes decreases with the frequency. For an experiment setup with the processing system running on 650 MHz, the number of clock cycles decreases from 4 cycles for a frequency of 250 MHz to one cycle for a frequency of 100 MHz. The number of clock cycles between the two handshakes is zero for all frequencies below 55 MHz, so for Xilinx Zynq-7010, our method is limited to frequencies above 55 MHz. This limit has to be considered to use this method in an attack scenario, and a profiling step is needed to determine the threshold to use.

4.3. Second Condition: Evicting a Cache Line from the Programmable Logic Part

The second condition for a successful cache attack is being able to evict a cache line from the cache. This section presents the method we used to fulfill the second condition required for a cache attack.

As mentioned above, a coherent write request forces the L1 cache memory to invalidate the cache line containing the address of the request if the coherent data are present in the L1 cache. Therefore, sending a coherent write request is sufficient to evict a cache line containing the targeted address. However, in order to not modify the content of the address the signal WSTRB[4:0] of the ACP must be equal to \( b'0000' \). The WSTRB[4:0] signal controls the number of valid bytes in the WDATA[31:0] signal that must be updated in the memory. Then, to fulfill the second condition for a cache attack, we used a write request issued by the master interface with WSTRB[4:0] = \( b'0000' \).

4.4. Limitation of the Proposed Method of Cache Line Eviction

This method has also a limitation. It fails to expel cache lines if the region containing the target address has a write strategy other than Write-back.

5. Experimental Proof of Two Cache Timing Attacks

Now, as we have fulfilled the two main conditions of cache attacks, we can use them to implement cache attacks from the programmable logic part. This section presents three cache attack scenarios that exploit cache coherency: two cache timing attacks (Flush+Reload [10], Evict+Time [11]) and a cache-based covert channel attack.
Note: All the attacks presented in this section are performed using a standalone application. Therefore, the experimental results are obtained with a low level of noise, contrary to what we could obtain with an operating system. For all the experiments, the processing system is running at 650 MHz and the programmable logic part is running at 250 MHz.

5.1. Cache Timing Side-Channel Attacks

This section describes the implementation of two cache timing attacks, the _Flush+Reload_ attack and the _Evict+Time_ attack. This attack targets the symmetrical encryption algorithm _AES-128_ (Advanced Encryption Standard) running in the processing system, it is a standard encryption use for many applications and is currently implanted in crypto-processors [19]. We use the specific implementation call _AES-128 T-table_ presented in the following section.

5.1.1. AES-128 T-Table

The _AES-128 T-table_ implementation is a performance-optimized implementation of the _AES-128_. This implementation combines the three functions of an _AES_ round (SubBytes, _ShiftRows_ and _MixColumns_) in a single step using four pre-calculated look-up tables _T₀_, _T₁_, _T₂_ and _T₃_ of 1 kb (256 elements each comprising 32 bits) for the first 9 rounds of the algorithm. The last round also uses a pre-calculated look-up table _T₄_ but for this table, the _MixColumns_ operation is excluded. The _AES-128 T-table_ uses a 16-byte plaintext _p_ = (₀, ₁, . . . , ₃) and a 16-byte key _k_ = (₀, ₁, . . . , ₁₅) as input. Equation (1) presents the structure of a round _i_ (1 ≤ _i_ ≤ 9) that uses the 16-byte intermediate state _Sᵢ_ = (₈, ₉, . . . , ₁₃) and the 16-byte round key _kᵢ_ = (₀, ₁, . . . , ₁₅) as inputs. The round _i_ output is the intermediate state _Sᵢ₊₁_. The intermediate state _S₁_ uses an _xor_ between the plaintext _p_ and the key _k_ as input.

\[
S_{i+1} = \begin{cases} 
T₀[S₈] ⊕ T₁[S₉] ⊕ T₂[S₁₀] ⊕ T₃[S₁₁] ⊕ \{k₀, k₁, k₂, k₃\}, \\
T₀[S₉] ⊕ T₁[S₁₀] ⊕ T₂[S₁₁] ⊕ T₃[S₁₂] ⊕ \{k₄, k₅, k₆, k₇\}, \\
T₀[S₁₀] ⊕ T₁[S₁₁] ⊕ T₂[S₁₂] ⊕ T₃[S₁₃] ⊕ \{k₈, k₉, k₁₀, k₁₁\}, \\
T₀[S₁₁] ⊕ T₁[S₁₂] ⊕ T₂[S₁₃] ⊕ T₃[S₁₄] ⊕ \{k₁₂, k₁₃, k₁₄, k₁₅\} 
\end{cases}
\]  

(1)

The _AES-128 T-table_ is targeted by most of cache attacks [12,15,16,18,19]. These attacks exploit the fact that the intermediate state _S₁_ depends directly on the plaintext _p_ and on the key _k_. If an attacker knows the byte _pᵢ_ of the plaintext and the elements (addresses) of the look-up tables (_T₀_, _T₁_, _T₂_ and _T₃_) used during the encryption process, he/she can easily deduce the byte _kᵢ_ of the key. This section uses this weakness in the _AES-128 T-table_ to demonstrate the feasibility of a cache attack originating in the programmable logic part.

5.1.2. Threat Model of the Side-Channel Attacks

Figure 9 shows the threat model used for the _Flush+Reload_ attack and the _Evict+Time_ attack. In the programmable logic part of Xilinx Zynq-7010, direct memory access (DMA) IP transmits the data encrypted by the secure ARM core to an I/O device. The processing system uses a software implementation of the _AES-128 T-table_ that is vulnerable to time-based cache attacks. The IP DMA has no information about the secret key used by the cipher. The master interface of the IP DMA includes a hardware Trojan that exploits the cache coherency to recover the encryption key.

For the _Flush+Reload_ attack and the _Evict+Time_ attack, we provide the master interface with the physical addresses where the four tables _T₀_, _T₁_, _T₂_ and _T₃_ are located. Our purpose is to demonstrate the possibility of a threat represented by a malicious master interface that exploits cache coherency. In a real case scenario, the master interface would have to scan the whole main memory looking for the first four elements of each table to locate them before launching the attack.
5.1.3. Flush+Reload

The Flush+Reload attack targets the first round of the AES-128 T-table running in the secure world of the processing system. Before implementing the attack, we perform a pre-profiling step to define the threshold to use for each frequency in order to distinguish a cache miss from a cache hit.

The main purpose of a Flush+Reload attack on the AES-128 T-table is to determine which index of the $T_0$ is accessed by the encryption process in order to recover the key. To do so, the proposed Flush+Reload attack scenario uses three main steps:

- **Step 1:** The malicious master interface evicts the cache line containing one of the 256 elements of the table $T_0$.
- **Step 2:** The master interface triggers encryption by sending a plaintext with a fixed pi byte.
- **Step 3:** Once the master interface receives the ciphertext, the interface issues a read request targeting the address of the element evicted in step 1 which counts the number of clock cycles elapsed between the handshake of the read address channel and the handshake of the read data channel. If the number of clock cycles is below the threshold, the master interface can deduce that the elements of $T_0$ evicted in step 1 have been accessed by the cipher.

In order to find the byte $k_0$ of the key for the Flush+Reload attack, we used the technique presented in [12]. This technique helps find the upper five bits of the byte $k_0$ and reduce the key search space to 48 bits.

The cache access patterns in $T_0$ table presented in Figure 10 are created by running the three steps presented above 256 · 256 time in order to scan the whole $T_0$ table (256 elements) and try all possible values of the byte $p_i$ (256 possible values for a byte). In Figure 10a, the diagonal pattern of black squares reveals that $k_0 = 0 \times 00$. The diagonal pattern is due to the fact that the intermediate state $S1$ of the AES-128 T-table has accessed the value $T_0$: $[0 \times 00 \oplus p_i] = T_0[p_i]$.

![Figure 9. Threat model used for the Flush+Reload and the Evict+Time attacks.](image)

**Figure 9.** Threat model used for the Flush+Reload and the Evict+Time attacks.

![Figure 10. Cache access pattern to $T_0$ table using a Flush+Reload attack from the programmable logic part, (a) frequency = 250 MHz, (b) frequency = 100 MHz.](image)

**Figure 10.** Cache access pattern to $T_0$ table using a Flush+Reload attack from the programmable logic part, (a) frequency = 250 MHz, (b) frequency = 100 MHz.
Figure 10b shows the cache access pattern to table $T_0$ with a frequency of 100 MHz applied to the master interface and $k_0 = 0 \times 0F$. This pattern shows that it is possible to implement cache attacks even with only one clock cycle to distinguish between a cache miss and a cache hit.

5.1.4. Evict+Time

The Evict+Time attack we implemented only targets the first round of the AES-128 $T$-table. In this scenario, we also performed a pre-profiling step to find a threshold that differentiates between the execution time of a plaintext encryption process with the $T_0$ elements present in the cache and without. The main steps of our scenario of Evict+Time attack are as follows:

- **Step 1:** The malicious master interface triggers the execution of plaintext encryption in the processing system. Like in the first attack, all plaintexts use a fixed $p_i$ byte. This first step loads the elements of $T_0$ table necessary for the encryption onto the cache.
- **Step 2:** The master interface evicts the cache line containing an element from the $T_0$.
- **Step 3:** The master interface again triggers the encryption of the same plaintext as that used in step 1. During this step, the cipher only loads the cache with the missing elements needed to perform the encryption. If the element of table $T_0$ evicted in step 2 is needed during the encryption, the cipher algorithm will load it from the external memory. This load operation adds some clock cycles to the execution time of the encryption process.
- **Step 4:** This step is executed at the same time as step 3. The master interface measures the time between the initiation of the encryption and the reception of the ciphertext. If the encryption time is above the threshold fixed during the pre-profiling step, the master interface can deduce that the element evicted in step 2 was used during the encryption process and then find the byte $k_i$ in the key.

Like in the Flush+Reload attack, the four steps need to be run 256 · 256 times to get the byte $k_0$. Figure 11 shows the cache access pattern of the $T_0$ table using Evict+Time from the programmable logic part. The pattern in the figure reveals that $k_0 = 0 \times 51$.

![Figure 11](image-url)  
**Figure 11.** Cache access pattern to $T_0$ table using a Evict+Time attack from the programmable logic part, frequency = 250 MHz.

5.2. Covert Channel Attack Exploiting Cache Coherency between the Programmable Logic and the Processing System

In this section, we introduce for the first time a covert channel attack between the programmable logic and the processing system of an SoC-FPGA. The secure world of
the processing system includes a spy process that uses Flush+Reload attack software to communicate with the receiver process, i.e., the malicious master interface used previously. The malicious interface uses our method to distinguish between a cache miss and cache hit. In this scenario, we assume that the spy and the receiver process are not allowed to communicate directly. The two processes use a shared memory address located in a secure region of the external memory to communicate. This address is only readable by the receiving process but is both readable and writeable by the spy process.

The spy process uses Algorithm 1 to send a logical ‘0’ and ‘1’. Algorithm 1 uses the flush technique presented in [13]. To transmit a logical ‘1’, the spy process flushes the shared address from the cache and sleeps for a period Time_1. To transmit a logical ‘0’, the spy process flushes the shared address from the cache and sleeps for a period Time_2. Time_1 is longer than Time_2. Between two successive bits sent, the spy process loads the data in the cache and sleeps for a period, Time_3. The choice of the period Time_1, Time_2 and Time_3 has a significant impact on the bandwidth and the error rate of the covert channel.

Algorithm 1: Encoding of data by the spy process

Input: address, data_to_transfer

For i=data_to_transfer_size To 0 Do
   If (data_to_transfert[i] = 1) then
      Flush(address)
      Usleep(Time_1)
   Else
      Flush(address)
      Usleep(Time_2)
   End if
   Access(address)
   Usleep(Time_3)
End For

To decode the transmitted data, the receiving process continuously issues a coherent read request targeting the shared address to measure access time. The receiving process counts the number of successive cache misses at the same time. If the number is small, a logical ‘0’ is received. If the number is big, the logical ‘1’ is received. The receiver process uses the detection of a cache hit as an initialization signal of the number of successive cache misses.

Figure 12 shows an example of decoding a “hi” message. The wide symbols in the figure are decoded to a logical ‘1’ and the narrow symbols are decoded to a logical ‘0’. Figure 9 shows an example of an ad hoc protocol with a start and an end: the first four bits (0 × 5) indicate the start of the message, and the four last bits (0 × 5) indicate the end of the message. Figure 12 also shows the presence of some errors (red circles) during the decoding process. These errors can be avoided by choosing longer periods for Time_1 and Time_2.

Figure 12. Decoding a “hi” message (h = 0 × 68 and i = 0 × 69).
We do not provide the error rate or the binary rate of this covert channel because our focus is on the applicability of such a channel and not on its best performance.

6. Conclusions

In this paper, we present the malicious use of cache coherency between the processing system and the programmable logic part of the modern SoC-FPGA. We describe a method based on an AXI bus signal to distinguish between a cache miss and a cache hit originating the programmable logic part. We prove the feasibility of two cache timing attacks, *Flush+Reload*, *Evict+Time* and a covert channel attack. Such attacks could have dramatic consequences for system security, and the designer who wishes to develop sensitive applications on SoC-FPGA must therefore take them into consideration.

**Author Contributions:** Writing—original draft, E.M.B.; Writing—review & editing, L.B. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the French “Agence Nationale de la Recherche”, in the frame of Archi-Sec project grant number ANR-19-CE39-0008-03.

**Conflicts of Interest:** The authors declare no conflict of interest.

**References**

1. Benhani, E.M.; Bossuet, L.; Aubert, A. The Security of ARM TrustZone in a FPGA-based SoC. *IEEE Trans. Comput.* 2019, 68, 1238–1248. [CrossRef]

2. Kamoun, N.; Bossuet, L.; Gazel, A. SRAM-FPGA Implementation of Masked S-Box Based DPA countermeasure for AES. In *Proceedings of the IEEE International Design and Test Workshop (IDT)*, Monastir, Tunisia, 20–22 December 2008; pp. 74–77.

3. Schellenberg, F.; Gnad, D.R.E.; Moradi, A.; Baradaran Tahoori, M. An inside job: Remote power analysis attacks on FPGAs. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, 19–23 March 2018; pp. 1111–1116.

4. Ramesh, C.; Patil, S.B.; Dhanuskodi, S.N.; Prowelengios, G.; Pillement, S.; Holcomb, D.; Tessier, R. FPGA Side Channel Attacks without Physical Access. In *Proceedings of the FCCM*, Boulder, CO, USA, 29 April–1 May 2018.

5. Zhao, M.; Suh, G.E. FPGA-Based Remote Power Side-Channel Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 21–23 May 2018; pp. 229–244.

6. Prowelengios, G.; Holcomb, D.; Tessier, R. Power Wasting Circuits for Cloud FPGA Attacks. In *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (FPL)*, Gothenburg, Sweden, 31 August–4 September 2020; pp. 231–235.

7. Islam, M.N.; Kundu, S. PMU-Trojan: On exploiting power management side channel for information leakage. In *Proceedings of the ASPDAC*, Jeju, Korea, 22–25 January 2018; pp. 709–714.

8. Alagappan, M.; Rajendran, J.; Doroslovacki, M.; Venkataramani, G. DFS covert channels on multi-core platforms. In *Proceedings of the IEEE VLSI-SoC*, Abu Dhabi, United Arab Emirates, 23–25 October 2017; pp. 1–6.

9. Benhani, E.M.; Bossuet, L. DVFS as a Security Failure of TrustZone-enabled Heterogeneous SoC. In *Proceedings of the IEEE ICECS*, Bordeaux, France, 9–12 December 2018.

10. Yarom, Y.; Falkner, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack (SEC’14)*, USENIX Association, Berkeley, CA, USA, 20 August 2014; pp. 719–732.

11. Osvik, D.A.; Shamir, A.; Tromer, E. Cache attacks and countermeasures: The case of AES. In *Proceedings of the Cryptographers’ Track at the RSA Conference*, San Jose, CA, USA, 13–17 February 2006; pp. 1–20.

12. Gruss, D.; Maurice, C.; Wagner, K.; Mangard, S. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, San Sebastián, Spain, 7–8 July 2016; pp. 279–299.

13. Lipp, M.; Gruss, D.; Spreitzer, R.; Maurice, C.; Mangard, S. Armageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, USA, 10–12 August 2016; pp. 549–564.

14. Kim, M.; Kong, S.; Hong, B.; Xu, L.; Shi, W.; Suh, T. Evaluating coherence-exploiting hardware trojan. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Lausanne, Switzerland, 27–31 March 2017; pp. 157–162.

15. Chaudhuri, S. A security vulnerability analysis of SoC-FPGA architectures. In *Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, 24–28 June 2018; pp. 1–6.

16. Reinbrecht, C.R.W.; Susin, A.; Bossuet, L.; Sigl, G.; Sepulveda, M.J. Side Channel Attack on NoC-based MPSoCs are practical: NOC prime+probe attack. In *Proceedings of the 29th IEEE Symposium on Integrated Circuits and Systems Design (SBCCI)*, Belo Horizonte, Brazil, 29 August–3 September 2016.
17. Xilinx. Zynq-7010 All Programmable SoC Technical Reference Manual, UG585 v1.11. 2016. Available online: https://www.xilinx.com/ (accessed on 25 June 2021).

18. Benhani, E.M.; Bossuet, L. Design a TrustZone-Enable SoC using Xilinx VIVADO CAD Tool; Technical Report; University of Lyon: Lyon, France, 2017.

19. Gaspar, L.; Fischer, V.; Bernard, F.; Bossuet, L.; Cotret, P. HCrypt: A Novel Reconfigurable Crypto-processor with Secured Key Management. In Proceedings of the International Conference on ReConFigurable Computing and FPGAs, Cancun, Mexico, 13–15 December 2010; pp. 280–285.