On Cache Timing Attacks Considering Multi-Core Aspects in Virtualized Embedded Systems *

Michael Weiß[1], Benjamin Weggenmann[1], Moritz August[2], and Georg Sigl[2]

[1] Fraunhofer Institut AISEC, Garching, Germany
{michael.weiss, benjamin.weggenmann}@aisec.fraunhofer.de
[2] Technische Universität München, Munich, Germany
{moritz.august, sigl}@tum.de

Abstract. Virtualization has become one of the most important security enhancing techniques for embedded systems during the last years, both for mobile devices and cyber-physical system (CPS). One of the major security threats in this context is posed by side channel attacks. In this work, Bernstein's time-driven cache-based attack against AES is revisited in a virtualization scenario based on an actual CPS using the PikeOS microkernel virtualization framework. The attack is conducted in the context of the implemented virtualization scenario using different scheduler configurations. We provide experimental results which show that using dedicated cores for crypto routines will have a high impact on the vulnerability of such systems. We also compare the results to previous work in that field and our visualization directly shows the differences between cache architectures of the ARM Cortex-A8 and Cortex-A9. Further, a non-invasive countermeasure against timing attacks based on the scheduler of PikeOS is devised, which in fact increases the system's security against cache timing attacks.

Keywords: Cyber-physical system (CPS), Virtualization, Trusted Execution Environment, Microkernel, AES, Cache Timing, Embedded Systems

1 Introduction

Former single-core real-time embedded systems used in the avionics and automotive industry are evolving to integrated ARM-based multi-core virtualized platforms, nowadays denoted as cyber-physical systems (CPSs). To save weight and costs of airplanes and vehicles, such systems run several user controlled applications beside security and safety critical applications side by side on the same physical system. Consider in-flight entertainment systems which provide users with the ability to connect their own untrusted devices, e.g., smart phones

* Parts of this contribution were supported by the German Federal Ministry of Education and Research in the project *SIBASE* through grant number 01IS13020.

and tablets. However, those systems also provide flight information which needs a connection to safety critical systems. That is why for instance the ARNIC-653 standard [3] demands for strict isolation and real-time constraints by statically configured partitions. A widely used real-time operating system framework in the avionics industry which provides partition separation according to ARNIC-653 is PikeOS [9]. It elaborates a microkernel and a user-space abstraction layer for this purpose.

However, none of those real-time operating systems have been examined for vulnerabilities to cache timing side channels circumventing the partition isolation considering influences of multi-core. Previous work mainly focuses on x86 systems in shared cloud scenarios.

Cache-based side channel attacks make use of a simple model to correlate the execution time of an algorithm with the state of the cache used by the CPU in charge. It is assumed that the execution time is lower if the data needed by the algorithm is already stored in a cache line (*cache-hit*). On the other hand, if the required data is not present in the cache and hence has to be loaded from the main memory (*cache-miss*), this will result in a longer execution time. This model is simple, but reasonable and only relies on the cache architecture of the CPU. Weiss et al. [23] provide a suitable attack scenario, however only on a single-core system using an academic real-time framework, focusing on mobile phone devices. We use this as base for our research on how actual CPSs running in the cockpits and cabins of airplanes are vulnerable to cache-based timing side channel attacks.

Our main contributions are:

1. We adapted the virtualization-based attack scenario from [23] to multi-core embedded system using the microkernel-based operating system framework PikeOS which has high relevance in avionic and also automotive industry.
2. We propose the discrete-time countermeasure which is based only on real-time configuration of the PikeOS scheduler as a drop-in update to existing systems and compare it to related approaches.
3. We elaborate different multi-core scheduler configurations and evaluate their vulnerability against time-driven cache attacks.
4. By comparing the attack results between single- and multi-core configurations, we are able to show that dedicated cores for crypto services leak the most information about the key.
5. Further, we compare our Cortex-A9-based setup against the Cortex-A8-based setup of [23], which leads to interesting patterns of key space reduction directly showing differences of the underlying cache architecture.

The rest of the paper is structured as follows: We provide background on cache based side-channels and related work in Section 2. In Section 3, the system architecture and attack scenario including the attacker model is described. We provide some more detailed background knowledge about the PikeOS scheduler in Section 4, before we describe the *discrete-time* countermeasure in Section 5. Experimental results of the attack performance under different scheduler configurations are evaluated in Section 6. Finally, the work is concluded in Section 7

2 Background and Related Work

Cache-based attacks can be divided into three different categories, each having a different attacker model. *Time-driven* attacks [4, 15, 14, 2, 5] make use of the cache model in a very general way as they only require timing data of entire runs of a cryptographic algorithm, e.g., an encryption using AES. This corresponds to an attacker who has only very limited or coarse information about the cache. *Trace-driven* attacks [1, 6] additionally require detailed information about the cache activity during single runs of the encryption, in particular the sequence of cache hits and misses caused by the memory accesses performed by the encryption algorithm. A trace can for instance be captured by profiling the power consumption while the encryption routine is running. This translates to an attacker, who has gained a substantial level of knowledge about the runtime cache behavior which in case of a power profile also requires physical access to the device. Finally, *access-driven* attacks [15, 8] assume to have knowledge about the cache-sets accessed by the algorithm. The underlying assumption is therefore that the attacker can control the cache runtime behavior. In the *Prime+Probe* attack [15], for example, those areas of the cache that also hold the lookup tables of the attacked algorithm are filled by a spy process with own data before the encryption is triggered (*Prime*). After the encryption, the spy process measures the access time to its own data to see which parts have been evicted from the cache by the encryption algorithm (*Probe*). Now the attacker can deduce which parts of the lookup tables were accessed by the encryption and from this infer some or all bits of the secret key. As can be seen from the above explanations, time-driven attacks are the most widely applicable class of attacks since they do not require a strong attacker with fine grained access to the cache.

In [4], Bernstein proposes a cache-based timing attack to recover the secret key of an AES encryption on a remote server. Bernstein's paper contained no thorough analysis of the attack and no explanation why the attack is successful. Neve et al. fill this gap in [14] by presenting a full analysis of Bernstein's attack methodology and explaining the correlation model. They argue that Bernstein's original technique cannot be used easily as a real remote-only attack where timings need to be measured by the attacker. Moreover, they improve Bernstein's attack by also considering second round information and thus lowering the number of required samples. To get accurate timings, Bernstein avoided the noisy network channel between the attacked server and the attacker by measuring the encryption time directly on the server, which is a rather unrealistic scenario since the server needs to be modified. In virtualization environments, however, the noise is negligible since local communication channels with only a small and almost constant timing overhead are used, as shown in [23].

Ristenpart et al. [16] consider side-channel leakage in virtualization environments on the example of the Amazon EC2 cloud service. They show that there is cross virtual machine (VM) side-channel leakage. They used the access-driven *Prime+Probe* technique from [15] for analyzing the timing side-channel. However, Ristenpart et al. are not able to extract a secret encryption key from one VM. In [23], Weiss et al. consider a virtualization-based system where the trusted

environment runs an AES server. Under the assumption that the untrusted environment could be hijacked by an attacker, they show that a man-in-the-middle attack via an adapted version of the cache-timing attack by Bernstein [4] is generally able to significantly reduce the key space, thus making brute-force attacks feasible. The impact of noise under realistic workloads is examined by Spreitzer and Plos [18], who evaluate time-driven attacks on conventional mobile devices (ARM Cortex A8 and A9). Unlike our approach, they consider noise induced by the Android operating system and applications running simultaneously on the device. However, they do so using a slightly unrealistic attacker model where the attacker captures timings in the very same process where the AES encryption routine is implemented and called, which likely reduces the effects of the OS and concurrent processes.

There are several ways to defend against time-driven cache timing attacks: One option is to switch to hardware-based implementations as provided by some processor manufacturers, e.g. Intel with its AES-NI instruction set [7], thus entirely avoiding cache-based attacks against the algorithm. If no hardware support is available, it is possible to change the implementation of the algorithm itself and get rid of the table lookups. While earlier software-based suggestions [13, 12] were generally slow compared to table-based implementations, Kasper et al. [10] present an efficient constant-time implementation based on bit-slicing that is suitable for stream and packet encryption.

Kim et al. [11] present a novel countermeasure against cache-based side channel attacks in a virtualization environment called STEALTHMEM. This countermeasure works at hypervisor level by assigning dedicated cache lines to each CPU in a group of CPUs with shared L3 cache. These so-called stealth cache lines are never evicted; therefore, sensitive data, such as S-boxes in AES, can be stored in these cache lines without introducing cache or timing side channels for an attack. Stefan et al. [19] propose instruction-based scheduling to prevent cache-based timing attacks on a single CPU. Instead of having a fixed amount of time, a process has a fixed amount of instructions it can execute before the next process is scheduled. The authors examine a simple timing attack and show that this attack is prevented by the proposed scheduler with negligible increase in the size of binaries and execution time. These countermeasures require considerable changes to the hardware, the hypervisor, or the cryptographic algorithms, whereas neither of which is necessary for our approach. Lately, Varadarajan et al. [20] have proposed a similar approach to our *discrete-time* scheduler scheme for cloud systems which they call soft-isolation. In contrast to our approach for real-time based schedulers, their approach relies on a feature of the Xen hypervisor scheduler called minimum run time (MRT) guarantee.

3 Attack Scenario and System Architecture

We assume a Trusted Execution Environment (TEE) which separates two compartments, a trusted environment which provides crypto services and an untrusted environment which runs user applications. The secret keys used for en-

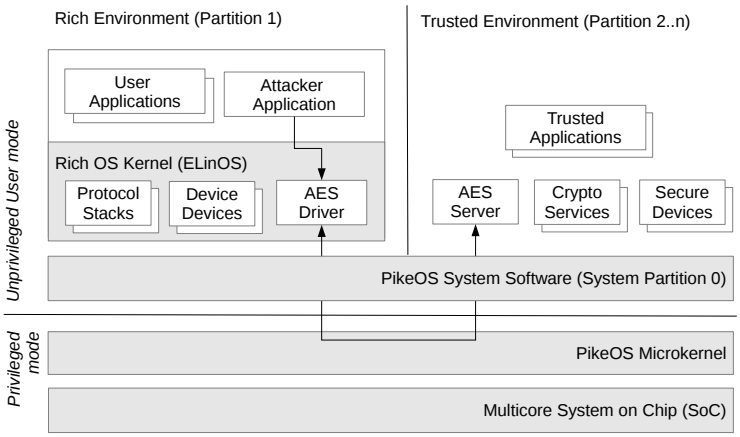


Fig. 1: Adapted virtualization based system security architecture

cryption have a high security value and thus are only accessible in the trusted environment. A viable usage scenario is, e.g., to establish a VPN tunnel. The network protocol stacks of a rich operating system kernel are used in the untrusted environment while the payload is encrypted by a driver using an encryption service inside the trusted environment. Hence, the secret session keys cannot be compromised by an attacker in the rich OS.

For our work we adapted the virtualization based security architecture of [23], which is a realization of a TEE to a microkernel system, to PikeOS. PikeOS distinguishes between resource and time partitions. A resource partition in PikeOS denotes a separate address space protected by the microkernel, while time partitions are used to assign computation time to threads. The microkernel itself only implements the basic mechanism for IPC, scheduling, and separation of address spaces in privileged processor mode. Device drivers, higher level abstraction for inter-partition communication as well as virtual memory management are implemented in the user-space abstraction layer, called PikeOS System Software (PSSW). Native device drivers for secure devices can be implemented in their own partition also in user-space. Figure 1 illustrates this architecture including the attack scenario. In our scenario, the architecture comprises a rich environment which runs the untrusted user applications in one partition as well as a trusted environment that hosts the security and safety relevant trusted applications each in their own partition. Both environments are allowed to communicate with each other using protocol messages transmitted via the virtualization layer, which in our case is the PikeOS microkernel and its user-space abstraction layer PSSW. To exchange data between the trusted and untrusted applications, shared memory is used. The user applications may use the trusted applications via special device drivers integrated into the rich OS kernel.

The concrete attack scenario now assumes that an AES encryption server runs in the trusted environment. To launch an encryption, a user application simply stores the plaintext in shared memory and calls the AES server through

IPC. The ciphertext is then written back to the shared memory. In this scenario, an attacker has compromised the rich OS and wants to determine the key used by the AES server. As he has full access to the rich OS in the untrusted environment, he is able to launch as many encryptions as he likes with chosen plaintexts. This he could do either by hijacking running processes or deploying own code that directly uses the kernel of the rich OS. The attacker is therefore able to launch a time-driven attack as it was discussed above.

4 Scheduling in PikeOS

PikeOS features a special scheduler that uses a combination of *time-driven and priority-based scheduling* to account for the different needs of the applications. To allow for deterministic real-time responsiveness, the scheduler uses a time-driven approach. Every real-time application is statically assigned to a time slot of a defined length. The length of these time slots can vary between applications but has to stay within a certain relation to the length of the other time slots. Every application is periodically scheduled for the length of the slot it is assigned to. As every partition gets assigned a defined amount of CPU time at defined points in time, they are able to schedule real-time processes themselves. This, so far, is a standard approach for scheduling real-time applications. To also support non real-time applications, a straightforward extension of this approach is to just create a new time slot and assign all applications without timing constraints to it. Within this slot, a standard round robin scheduling scheme can be applied. However, this approach is inefficient since it wastes a lot of CPU time. The PikeOS scheduler refines this approach to a more efficient strategy. It might occur that the processes of a real-time application finish before its time slot end or that it does not have any processes to run at all. As it would harm the temporal determinism, the scheduler cannot simply switch to the next application in this situation. Rather than wasting this time, the PikeOS scheduler uses this excess CPU time to schedule applications with no real-time constraints. For this purpose, it leverages priority-based scheduling. All real-time applications are assigned the same mid-level priority number while low priority numbers are assigned to the other applications. Now, the scheduler continues to schedule the real-time applications periodically but uses the excess time to schedule the low-priority non real-time applications in a round robin fashion. In this way, no computing time is wasted and the overall amount of time needed to execute all applications decreases drastically when compared to a standard RTOS scheduler.

5 Discrete-Time Countermeasure

One main pitfall of novel countermeasures is that some of them require changes to already established systems that are too substantial to be easily implemented, hence making these countermeasures practically irrelevant. The *discrete-time* countermeasure that is presented in the following therefore aims at making cache based time-driven attacks infeasible for attackers while demanding as few

changes and inducing as little overhead as possible. Assume the rich OS and the trusted environment are implemented as partitions in PikeOS and are hence handled by the scheduler. Now assume the attacker has compromised the rich OS and is able to launch the timing attack against the AES server that runs in a trusted partition. In order for the attacker to successfully carry out the attack, two conditions must be fulfilled:

1. He must be able to retrieve enough samples from the AES server, in the order of several hundred millions.
2. The samples must leak enough information for the correct hypothesis on the key to yield a higher correlation on average than all wrong hypotheses.

The *discrete-time* countermeasure aims at these two points. It works straightforward in that both applications, the rich OS and the AES server in the trusted domain, are treated as real-time applications such that each is assigned an own time slot. Note, that it is not necessary for either of the two applications to have any real-time time constraints in order for the scheduler to be configured as described above. Using this configuration of the scheduler the time measured by the attack for one encryption t_{enc} is now given by Equation 1.

$$t_{enc} = n \cdot t_{OS} + m \cdot t_{serv} \tag{1}$$

with t_{OS} being the length of the time slot of the rich OS and t_{serv} being the length of the time slot of the AES server. The two variables n and m represent the number of executions of the two time slots. Note that we ignore negligible timing quantities that are independent of the AES server, such as the remaining time in the slot of the rich OS after the encryption was requested and the time passing in the first slot of the rich OS after the encryption is done before the attacker's process is scheduled. As it can be easily verified the time is always a multiple of the two time slot lengths which gives rise to the countermeasure's name. This has two major effects on the attack. Firstly, as the scheduling for these two applications is strictly time-driven, the rich OS will be scheduled a number of times while still waiting for the encryption to finish and hence being idle. This will increase the time needed by an encryption in a way that, given carefully chosen values for t_{OS} and t_{serv}, a single encryption as it is needed for benign purposes can still be done without noticeable delay. However, a number of encryptions as needed for an attack will take a significantly larger amount of time. This already will make an attack time-wise more difficult. Secondly, as the information that can be gained by one sample is now very coarse-grained, there is only a very small correlation left between the timing information and occurring cache-misses or hits. This will make it very hard for the attacker to distinguish the correct key hypothesis from false ones and will increase the number of necessary samples. Therefore, the discrete-time countermeasure is a strong shield against the kind of attacks considered here. Furthermore, the countermeasure requires no change of any kind in the code and also causes arguably only little timing overhead. It is also straightforward to implement, can be extended to multiple applications and is most likely also applicable to

other RTOS schedulers working in a similar manner as the PikeOS scheduler. Although not in the focus of this paper, access-driven attacks can be prevented similarly by a simple configuration in the scheduler to flush the cache when switching partitions.

6 Evaluation

To practically analyze the scenario presented in Section 3, we elaborated the following testbed. The untrusted runtime is implemented using the para-virtualized Linux distribution ELinOS including the necessary code for the attacker to conduct the timing attack. The AES server in the trusted runtime is implemented as an application based on the native PikeOS API. Obviously, both applications have their own partition. To enable the communication between the two partitions, two unidirectional queuing ports and a shared memory page were set up. The rich OS and the AES server use these ports to communicate via a simple handshake protocol and use the shared page as buffer for plain- and ciphertexts. Queuing ports are unidirectional communication channels defined in the ARNIC-653 [3] standard that can be set up between two partitions statically at compile-time and then initialized at run-time by the applications.

As hardware platform, we chose the Freescale i.MX6 SabreLite board which comprises a Quad-Core ARM Cortex-A9 CPU with 1.2GHz. The cache architecture consist of a 32KB I- and D-Cache (L1) per Core and a 1MB shared L2 cache. The L1 cache is 4-way associative and has a cache line size of 32 byte. For precise timing measurements, the ARM CCNT register was utilized as stated in [23] and [18].

To analyze the success rate of Bernstein's timing attack, the effect of a broad range of parameters was examined. For the comparison between different values for these parameters, two criteria were used.

1. The number of different candidates for each key byte
2. The average position of the correct candidates in the ordered output lists

The first one directly gives information about how much the key space could be reduced by the attack. To quantitatively measure the effectiveness of the attack, this is therefore the best parameter. In the best case only one candidate, namely the correct one, remains for each byte and the key is hence revealed completely. But even only a significant reduction of the number of candidate bytes is already valuable to the attacker as he then can launch a brute-force attack in the reduced key space with the remaining possible values. However, this score does not use all information of the output of the attack. As the list of possible candidates for each key byte is ordered, it is interesting to know at which positions in these lists the correct values can be found. This is a measure for the ability of the attack to separate the correct hypotheses from the other remaining ones. In the best case, the correct value for each key byte always has the highest correlation and is therefore at first position in the list. That information is also of high interest

Table 1: Summary of results for different scheduler configurations

No.	Utilized Cores	Scheduler Configuration	Average Position	Remaining Key-space
1	2	1 Core dedicated each (Single)	4.0	2^{72}
2	4	4 Cores shared (Quad)	4.25	$\approx 2^{82}$
3	4	4 Cores Server, 1 core shared rich OS (Server)	4.0	$\approx 2^{82}$
4	4	2 Cores dedicated each	4.375	$\approx 2^{73}$
5	1	1 Core shared	4.3125	$\approx 2^{80}$

to an attacker as he can use this information to significantly speed up his brute-force attack. Since he knows the correlation of all remaining possible byte values, he can order the possible keys by the correlation and then test for candidates with higher correlation first. This will usually require much less than the average $\frac{n}{2}$ guesses, n being the number of key candidates. Another approache to reduce brute-force complexity could be to use recently proposed key-rank estimation procedures [22], [21] as shown by Spreitzer et al. [17].

For all the experiments summarized in Table 1, normal priority-based scheduling was used and the profiling and attack phase were done on the same device. This might not always be possible in a real-world setting, but was done to have an optimal setting for the evaluation. If not stated otherwise the attacked key was

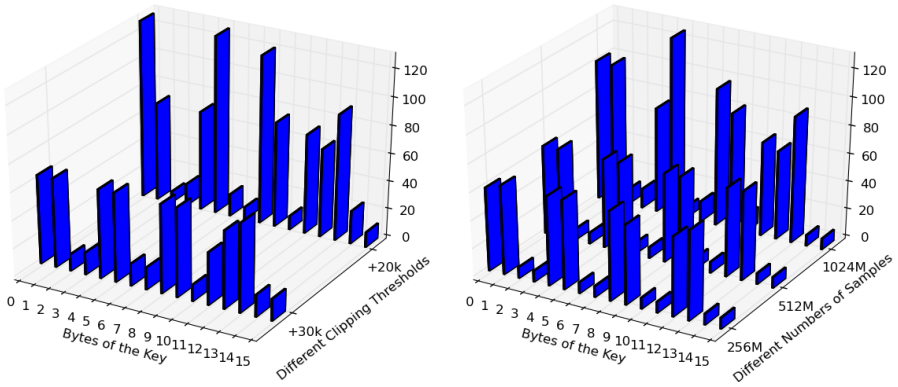
$$0x21\ 53\ fc\ 73\ d4\ f3\ 4a\ 98\ 17\ 33\ bb\ 3f\ 18\ 92\ 00\ 8b$$

and both profiling and attack phase were conducted with 512 million samples to have approximately 2 million samples for each possible key candidate.

6.1 Identifying and Tuning of Attack Parameters

To reduce the noise in the measurements, Bernstein disregards all measurements above a certain threshold. In the original code, this threshold was set to a value fitting the timing behavior of his implementation. This value was therefore changed in this implementation. To evaluate the effect of this clipping, two different thresholds were investigated both with 512 million samples for attack and profiling phase. The threshold that was initially set to about 30,000 clock cycles higher than the average of the timing samples was compared to the threshold 20,000 above average. The results are displayed in Figure 2a and Table 2a. The results clearly show that the lower threshold leads to a significant lower reduction of the key space. This implies that the timings lying in the interval between the two thresholds indeed contained information about the key. This also complies to the findings in [17], which shows that minimal timing attack of [**?**] does not leak any information on ARM.

One parameter that comes to mind very quickly when thinking about analyzing a side channel attack is the number of samples. One would assume that an increasing number of samples automatically results in a higher success rate



(a) Clipping Thresholds (b) Increasing Samples

Fig. 2: Histograms describing the numbers of possible candidates for all bytes of the key and for varying clipping thresholds and samples

as the noise gets averaged out more and more, leaving only the relevant information behind. To verify this assumption, we conducted the attack with 256, 512 and 1024 million samples. The results are displayed in Figure 2b. Table 2b shows the average position of the correct key byte candidates. As expected, increasing the number of samples does in fact also increase the success rate of the attack. However, the increase of the success rate shows a logarithmic behavior. This behavior is derived directly from the cache architecture. As only the upper k bits of a data word are used to index the cache lines, the timing behavior is independent of the lower bits. In the best case, the attack could therefore only reveal the upper k bits of each key byte. This explains the observed boundary of the reduction of the key space. It furthermore explains why the remaining number of possible values per byte is in almost all cases a power of 2. A similar behavior was described by Neve et al. [14]. This limitation only applies for aligned T-tables. In the case of disaligned T-tables, which is not the case in our setup, even more information might leak.

6.2 Single Core vs. Quad Core

The PikeOS scheduler allows the use of a CPU mask to specifically select the cores that shall run a partition. As each core has its own L1 cache but all cores share the L2 cache, it is interesting to examine how the success rate of the attack changes when only one or all cores are used. To do this, three different configurations were regarded. For the first one both partitions were run by a single core (configuration 1) while for the second one both partitions were run on all four cores (configuration 2). The third configuration involved the AES server running on all four cores (configuration 3) while ELinOS was assigned only one core. The results are depicted in Figure 3a and Table 1.

Table 2: The average position of the correct key byte candidates for the different clipping thresholds and numbers of samples

(a)

Clipping Threshold	Average Position
+30k	4.3125
+20k	4.375

Measurements conducted with 512M Sampales in configuration 4, see Table 1

(b)

Number of Samples	Average Position
256M	5.5625
512M	4.0
1024M	3.875

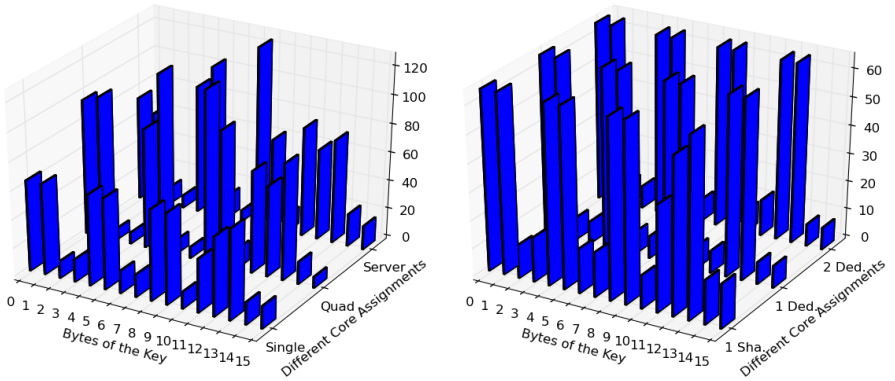
Measurements conducted in configuration 1, see Table 1

It can be seen that configuration 1 gave the best results for both criteria, and scenario 2 yielded the worst. This is understandable since in the first scenario, the T-tables are stored in a single L1 cache and the L2 cache, whereas in scenario 2 the T-tables are most likely scattered over the four L1 caches and the L2 cache. This decreases the signal to noise ratio with high certainty and thus lowers the success rate. Additionally, when both the rich OS and the AES server use the same core, their cache usage will interfere which also reduces the quality of the timing samples. This effect is visible in the difference between scenarios 2 and 3. Although the AES server uses four cores in scenario 3 as well, it only interferes with the other application in one of them which leads to an overall better success rate of the attack.

Using a dedicated core for the AES server might not be a good idea as it reduces the noise. Therefore, it was investigated how the success rate of the attack is affected when the two partitions have one or two cores for their own in comparison to the configuration where both partitions share only one core. The results are shown in Figure 3b.
The use of dedicated cores leads to a significantly better success rate in terms of the total number of remaining key candidates. The setup with one dedicated core also shows a slight decrease in the average position of the correct key byte values. As it can be seen, assigning one core to each partition (configuration 1) thereby results in a slightly better attack result than using two dedicated cores (configuration 4). This can be explained by the already discussed effect of using multiple L1 caches. However, the slight increase of the average position compared to the scenarios where 4 cores are utilized seems to be caused by measurement inaccuracies. In summary, when using an ordinary priority based scheduling scheme on a multi-core system without any countermeasures, it is not recommended to use a dedicated core for the cryptographic algorithm as this would reduce the noise significantly.

6.3 Comparison to Fiasco Setup

In [23], Weiss et al. present results for Bernstein's attack carried out in a very similar virtualization setting. In contrast to the hardware presented above, Weiss



(a) Shared Scheduler Configurations (b) Dedicated Scheduler Configurations

Fig. 3: Histograms describing the numbers of possible candidates for all bytes of the key and for varying scheduler configurations.

et al. used a beagleboard that is based on a Cortex-A8 with 720 MHz. To implement the virtualization scenario, the Fiasco.OC microkernel together with L4Re was utilized. Note that we use the same key as in [23] to provide comparable results. Weiss et al. report for the OpenSSL implementation of AES that they were able to reduce the byte value space of almost all bytes to 16 possibilities. For the fourth byte, no reduction was possible while the eighth and sixteenth byte could only be reduced to 32 possible values. This result was achieved with 2 million samples for each byte value, translating to the overall number of 512 million samples that was also used in this work. Data about the position of the correct key byte values in the output lists was not provided.

The best result achieved in terms of the reduction of the key space in this work draws a very different picture. For one dedicated core for the ELinOS and the AES partition respectively, the highest reduction found was a reduction down to 8 possible values for the bytes 3,4,7,8,11,12,15,16. For the remaining bytes a reduction was possible only down to 64 different values. This pattern is interesting in itself as every consecutive 2-byte tuple seems to be highly correlated in the reduction capability. However, it is also very different from the result stated above. For this implementation, the maximally achieved reduction is twice as high as for the implementation of Weiss et al. Nevertheless, only half of the bytes could be reduced that far while for the implementation of [23], nearly all byte spaces could be reduced to the respective minimum. Then again, in the implementation of this work all bytes could be reduced to at least 64 different values. This was not the case for the implementation using the Fiasco.OC kernel. Both implementations have in common that there seems to be a limit for the reduction of the key space that depends on the implementation. This was already mentioned above and is also stated in [23]. The two results are compared in Figure 4. The difference of the reduction pattern reflects the different cache architectures in terms of the cache line size. On the Cortex A8 with a 64 Byte

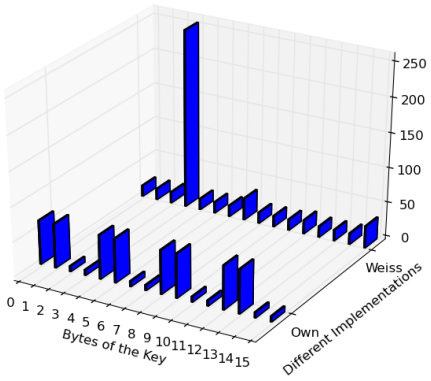


Fig. 4: Results for the implementation of this work compared to [23]

cache line size every fourth key byte is harder to reduce, while on the Cortex A9 with 32 Byte cache line size every first 2 bytes are harder to reduce. Both pattern repeat every 4 bytes, this is due to both caches are 4-way associative.

The total number of possible keys was reduced to 2^{72} for the worst PikeOS setup and to roughly 2^{70} for the Fiasco setup showing a slight advantage for the PikeOS setup. However, compared to the setup utilizing only one core with a reduction of key space to $\approx 2^{80}$ the PikeOS setup is ≈ 7 orders of magnitude harder to attack.

6.4 Evaluation of the Countermeasure

To evaluate the effectiveness of the discrete-time countermeasure, a range of different scheduler configurations was tested. The ELinOS and the AES server partition were assigned one time slot each and the length of these slots was then varied. It was quickly found that the length of both slots would have to be in a certain relation in order to ensure that the rich OS and the AES server work correctly. One configuration that led to a behavior of the system indistinguishable from the behavior with simple priority-based scheduling was found to be to set the slot length to 5 ticks for both partitions. The default duration of one tick was set to 1 ms. Using this configuration, the delay of single AES encryptions increases significantly by roughly about 70% while in contrast the encryption of a whole buffer with the size of a memory page may be conducted with only a small overhead of less than 30%, see table 3. This configuration was therefore chosen for the attack. Both partitions were assigned one dedicated core and the rest of the setup remained unchanged from previous experiments.

After running the profiling phase for one day we were able to retrieve ≈ 34 million samples. To capture the whole amount of 512 million samples for both phases, this means a total run-time of about one month for the above configuration of the scheduler. Remember that due to the different timing behavior induced by the countermeasure an even higher number of samples is needed in order to recover the key as good as possible. Therefore, it is reasonable to as-

Table 3: Performance comparison of the countermeasure

Scheduling Scheme	Average Clock Cycles per AES-block	
	one block (16 Byte)	one page (4 KByte)
Priority-Based	$\approx 125,000$	≈ 1770
Discrete-Time	$\approx 210,000$	≈ 2286

sume that for the attack to produce a useful output at least twice the number of samples and hence, with the overhead caused by our countermeasure, even more than twice the time is needed. Even if the attacker would do the profiling phase off-line, he would still need to be able to access the system for about one month. It is very unlikely that such a computational intensive attack would remain unnoticed for the entire time frame. Furthermore, depending on the actual use of the AES server, a rescheduling of the key might occur during that time, too. It can be seen from this that the proposed countermeasure indeed protects a device very well while simultaneously requiring almost no effort to be set in place. Also, the user experience does not change with the countermeasure which might be an important factor for the mobile device market. The different run-times of one encryption for priority-based scheduling and the countermeasure are shown in Table 3. For a more thorough evaluation of the discrete-time countermeasure, additional experiments need to be conducted.

Comparison to other Countermeasures In [11] and [19], two novel countermeasures against cache-based attacks are introduced. Since these countermeasures target the same class of attacks as the discrete-time countermeasure, it is interesting to compare their approaches with ours. As the focus of this work was put on time-driven attacks, the comparison will focus on this aspect as well.

The STEALTHMEM countermeasure [11] tries to prevent both active and passive time- and access-driven attacks in virtualization environments. To that end, it uses dedicated cache lines in the shared cache for each CPU. Depending on the variant of STEALTHMEM used, this either reduces the total available amount of memory and shared cache, or it takes extra time to ensure that the stealth cache lines are not evicted from the cache. Both variants imply a small penalty in performance of about 5.9% and 7.2% respectively, and AES encryptions of 50,000 bytes are about 5% slower with the first variant. Unlike the STEALTHMEM approach, the discrete-time countermeasure has no impact on the available cache and system memory. However, due to the larger time slots in our countermeasure, the overall performance degrades by about 30% for AES encryptions on 4 KB of data, as explained above. To use STEALTHMEM, the hypervisor is extended with a special driver offering an API to the VMs that manages access to the dedicated cache lines. For Windows Server 2008 R2 with Hyper-V, this amounts to 5,000 lines of C code to be added to the hypervisor and 500 lines of C code added to the Windows boot loader modules. Furthermore, the implementations of cryptographic algorithms have to be modified to make

use of the stealth cache lines via the provided API. For using our discrete-time countermeasure on the contrary, only a reconfiguration of the scheduler is needed. Neither the system nor the implementation of the cryptographic algorithm has to be changed. Also note that the required modification of the algorithm presents a potential pitfall. If not done correctly, some leakage remains and therefore breaks the countermeasure. Moreover, the amount of available cache lines that can be reserved for a core is limited so it has to be made sure that all relevant lookup tables fit inside to prevent information leakage.

The instruction-based scheduling scheme suggested in [19] aims at preventing cache-based attacks that exploit certain scheduling-induced race conditions between processes that arise due to the dependency of the execution time on the cache content. Both methods are similar in that they use a fixed value as their criterion for the scheduling. As the name implies, instruction-based scheduling uses a specified number of executed instructions as scheduling criterion. This prevents only those attacks that try to exploit the mentioned race conditions – but only when the processes are run on a single core. Furthermore, it is not sufficient to prevent time-driven attacks such as Bernstein's, since an attacker can still measure the total execution time which still depends on the cache. The discrete-time countermeasure on the other hand prevents this kind of race conditions even with multiple cores, and masks the overall execution time of an AES encryption. Also, instruction-based scheduling is a novel approach and hence not widely supported by current micro kernels. Therefore, extra effort has to be done to integrate it into existing systems or implement a new one which supports instruction-based scheduling. This is not the case for the discrete-time method, where our countermeasure can be readily configured. With respect to the overhead, both methods are fairly similar as they do not need any adaption of the applications and only induce a small time overhead.

7 Conclusion

In this work, we stated an attack scenario using a time-driven cache attack against embedded devices used in cyber-physical systems (CPSs) on the example of PikeOS, a microkernel-based operating system framework compliant to the ARNIC standard. We evaluated the attack with different scheduler configurations showing that dedicated cores for the crypto routine provide the most timing leakage.Further, we compared the results to a similar setup [23] for virtualization based Trusted Execution Environments (TEEs). We showed that using a shared core similar to the microkernel configuration in [23], the PikeOS setup of this work is about 7 order of magnitude less vulnerable in reduction of key space, but at least one order of magnitude in the worst configuration using dedicated cores. Furthermore, we provided the scheduler based *discrete-time* countermeasure against time-driven cache attacks. Compared to other novel countermeasures, it does not depend on any hardware, software architecture or algorithm changes. Thus, our approach can be used as a drop-in configuration update for running CPSs, or other embedded platforms using a configurable scheduler.

References

1. Onur Acıiçmez and Çetin Koç. Trace-driven cache attacks on aes (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer Berlin / Heidelberg, 2006.
2. Onur Acıiçmez, Werner Schindler, and Çetin Koç. Cache based remote timing attack on the aes. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer Berlin / Heidelberg, 2006.
3. Aeronautical Radio, Inc. *Avionics Application Software Standard Interface, ARINC Specification 653*, 1997.
4. Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.
5. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *CHES'06*, pages 201–215, 2006.
6. Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. Improved trace-driven cache-collision attacks against embedded aes implementations. In Yongwha Chung and Moti Yung, editors, *Information Security Applications*, volume 6513 of *Lecture Notes in Computer Science*, pages 243–257. Springer Berlin Heidelberg, 2011.
7. Shay Gueron. Intel® advanced encryption standard (aes) instructions set. Technical report, 2008.
8. D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy – S&P 2011*. IEEE Computer Society, 2011.
9. Robert Kaiser and Stephan Wagner. Evolution of the pikeos microkernel. In *First International Workshop on Microkernels for Embedded Systems*, page 50, 2007.
10. Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2009.
11. Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.
12. Robert Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202. Springer Berlin Heidelberg, 2008.
13. Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer Berlin Heidelberg, 2007.
14. Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at bernstein's aes side-channel analysis. In *ASIACCS*, page 369, 2006.
15. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005.
16. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

17. Raphael Spreitzer and Benoît Gérard. Towards more practical time-driven cache attacks. In David Naccache and Damien Sauveron, editors, *Information Security Theory and Practice. Securing the Internet of Things*, volume 8501 of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin Heidelberg, 2014.
18. Raphael Spreitzer and Thomas Plos. On the applicability of time-driven cache attacks on mobile devices. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 656–662. Springer Berlin Heidelberg, 2013.
19. Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 718–735. Springer, 2013.
20. Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 687–702, San Diego, CA, August 2014. USENIX Association.
21. Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In LarsR. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *Lecture Notes in Computer Science*, pages 390–406. Springer Berlin Heidelberg, 2013.
22. Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Security evaluations beyond computing power. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 126–141. Springer Berlin Heidelberg, 2013.
23. Michael Weiss, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In *14th International Conference on Financial Cryptography and Data Security (Financial Crypto 2012)*, Lecture Notes in Computer Science. Springer, 2012.