IDP: An Analysis of a Cache-Based Timing Side Channel Attack and a Countermeasure on PikeOS

Moritz August
Institute for Security in Information Technology
Technical University of Munich
Germany
moritz.august@tum.de

ABSTRACT
Virtualization has become of increasing importance for the security of embedded systems during the last years. One of the major threats to this security is posed by side channel attacks. In this work, Bernstein’s time-driven cache-based timing attack against AES is revisited in a virtualization security scenario and the PikeOS micro kernel system is presented. A novel countermeasure against timing attacks based on the scheduler of PikeOS is devised. The attack is mounted in the context of the implemented virtualization scenario and the results of several experiments are reported. The countermeasure is compared to related approaches.

1. INTRODUCTION
During the last decade, the security of embedded systems has become more and more crucial to a variety of products, ranging from industrial SCADA systems over mobile devices to products of the automotive sector. As the requirements for embedded systems differ from those for home computers or servers, do their capabilities to support security measures. Simultaneously, side-channel attacks have emerged as a class of particularly effective attacks against cryptographic security measures implemented in embedded systems on low- or middle-class-performance hardware. A still growing percentage of smart mobile devices is used not only for communication but also for sensitive tasks like payment. Motivated by this, one of the most relevant scenarios in which cryptography is combined with embedded systems today are virtualized environments. In these scenarios, typically a so called rich OS exists that offers the functionality for the user and runs in an untrusted domain. For security relevant tasks a trusted domain exists that allows for the secure execution of the corresponding applications. To develop secure products, it is therefore of major interest to analyze both the effect of side channel attacks on the security of cryptographic applications in a virtualization context and the robustness of existing systems against them. This work will hence analyze the effect of a cache-timing side channel attack on AES in a virtualization scenario implemented for the PikeOS microkernel operating system, as an example of a real-time system dedicated to security. It will furthermore evaluate methods to counteract that threat by using the system’s scheduler. The structure of this work is as follows: Section 2 will first give an overview over recent related work in the field. Subsequently, Section 3 will revisit the aforementioned attack, while Section 4 will then present the PikeOS system. The countermeasure against passive time-driven side channel attacks will then be introduced in Section 5. The implementation chosen for this work will be explained in Section 6 and in Section 7 the conducted experiments and their results will be presented. Finally, this work will conclude with a discussion of the achieved results.

2. RELATED WORK
In [1], Weiss et al. devise a virtualization-based security architecture as it could be employed in mobile devices to separate the untrusted and user-modified rich OS, e.g., Android, from a trusted compartment used for the execution of security sensitive applications. In their scenario, the trusted environment serves as an AES encryption server used in a challenge-response protocol to authenticate the device against some backend server. Under the assumption that the untrusted environment could be hijacked by an attacker, they show that a man-in-the-middle attack can successfully be launched via an adapted version of the cache-timing attack by Bernstein [2]. Weiss et al. show that the adapted attack is generally able to significantly reduce the key space in order to enable brute-force attacks. This is possible despite the additional security provided by the virtualization environment and by simply regarding the AES server as a black box. This work picks up the approach of [1] and will therefore refer to it in the following sections.
Kim et al. present a novel countermeasure against cache-based side channel attacks in a virtualization environment called STEALTHMEM in [3]. This countermeasure works at hypervisor level by assigning each CPU of a group of CPUs sharing an L3 cache so called stealth cache lines. These cache lines can only be used by the processor they are assigned to and are never evicted. Therefore, sensitive data as , e.g., S-boxes in AES, can be stored in these cache lines without introducing cache or timing side channels for an attack. Via a driver in the hypervisor, each application can specifically access the memory pages, called stealth pages, of the pre-image set corresponding to the CPU’s stealth cache lines. It is shown in [3] that using the API provided by the driver, popular cryptographic algorithms such as AES, Blowfish or RSA can be adapted to use STEALTHMEM with only a few lines of changed code. Furthermore, the overhead of the security measure is stated to be relatively low.
A novel scheduling mechanism in the context of information flow control of webservices is proposed by Stefan et al. [4]. The scheduling scheme aims to prevent cache-based timing attacks and is based on the well known round-robin method. It uses a constant number of retired instructions as schedul-
by determining the first round of AES that is able to recover 60 bits of the collisions. The first attack is a known-plaintext attack on the execution time of table-based AES implementations on cache. All three attacks leverage the dependency of the execution of table-based AES implementations on cache content in classic preemptive time-based schedulers. Stefan et al. present a simple timing attack and show how this attack is prevented by the proposed scheduler. It is shown that for a reference implementation using the LIO (Labeled IO) IFC (Information Flow Control) library for Haskell the overhead both in size of the binaries and the execution time is negligible. Stefan et al. also show theoretically that their scheduler does not violate any IFC constraints. In [5] Bonneau et al. present three different cache-based timing attacks against efficient table-based implementations of AES. All three attacks leverage the dependency of the execution time of table-based AES implementations on cache collisions. The first attack is a known-plaintext attack on the first round of AES that is able to recover 60 bits of the key by determining the xor-result of some pairs of the key bytes. To do so it needs $2^{14.58}$ plaintext-timing samples. As it is pointed out by the authors, the remaining key space is still very large and a brute-force attack on the remaining bits hence infeasible for most classes of attack. The second attack is a known-plaintext attack on the final round of AES that is able to recover the full key using $2^{17}$ samples. It works by determining the pairwise xor-result of all key bytes and validating key candidates by reversing the key schedule. The third attack is an enhancement of the second that uses a refined hypothesis and by that is able to recover the full key with $2^{13}$ samples. All three attacks do not need access to the cache and therefore fall into the category of passive timing attacks that can also be executed remotely.

3. THE TIMING ATTACK

In this section, the attack presented by Bernstein in [2] will be revisited. To that end, first a short overview over classes of timing side channel attacks and the corresponding attacker models will be given. Next, the scenario in which the attack is mounted in the context of this work will be presented and sequentially the attack itself will be explained.

3.1 A short Taxonomy of Cache-Based Timing Attacks

Cache-based timing attacks can be divided into three different categories, each having a different attacker model. It is however common to all of them that they 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 architectural properties of the CPU. Hence it is applicable in a wide range of scenarios, including virtualization settings as examined in this work.

Time-driven attacks make use of this model in a very general way as they only require timing data of runs of a cryptographic algorithm, e.g., an encryption by AES. This corresponds to an attacker who has only very limited or no access to the cache. Trace-driven attacks additionally require the information about the precise memory-addresses that cause cache-hits and need to be able to profile the cache during the execution of the algorithm. This translates to an attacker, who has gained a substantial level of access to the cache. Finally, access-driven attacks assume to have knowledge about the cache-sets accessed by the algorithm. The underlying assumption is therefore that the attacker has full access to the cache.

As can be seen from the above explanation, time-driven attacks are the most widely, since platform-independent, applicable class of attacks. While all three classes are usable in the virtualization scenario discussed here, a timing-driven attack requires significantly less work to mount than a trace- or access-driven attack.

3.2 The Attack Scenario

In this work, the virtualization based security architecture that was presented and implemented using the Fiasco.OC micro kernel system in [1] is picked up and adapted. It is inspired by modern security architectures of operating systems in mobile devices like Android but could also be put to use in other domains. Figure 1 illustrates this architecture. As it can be seen the architecture comprises a rich environment, which runs the untrusted user applications, and a trusted environment that hosts the security relevant trusted applications.

Both environments are allowed to communicate with each other using protocol messages transmitted via the virtualization layer. To exchange data between the trusted and untrusted applications, shared memory pages are used. The user applications may use the trusted applications via special device drivers integrated into the rich OS kernel. It is noteworthy that it is also possible for all trusted applications to have an own trusted environment, so that they do not need to share any memory. 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 the shared page and calls the AES server. The ciphertext is then written back to the shared memory. In this scenario, an attacker has gained access to the rich OS by the use of
malware and wants to determine the key used by the AES server. This key could for example be of interest to him because the server is used in some kind of authentication protocol to encrypt nonces. If the attacker would have knowledge about the correct key, he could simulate the AES server and launch a man-in-the-middle attack. As he has access to the system, 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. This scenario is very close to situations as they could arise in real settings and therefore of high practical relevance.

3.3 Bernstein’s Attack

The timing attack on AES introduced by Bernstein in [2] can be classified as a time-driven template attack on the first round of AES. It was originally implemented as a client-side attack against an AES server that is listening for UDP-packets but is due to its general nature just as well applicable in the setting examined in this work.

The attack is aimed against efficient table-dependent implementations of AES like the OpenSSL implementation. In these implementations, the input of each round is used to lookup precomputed values stored in tables to speed up the encryption. In the first round for example, this input is simply

\[ p_b \oplus k_b, b \in \{0,..,15 \} \]

with \( p \) being the plaintext, \( k \) the key and \( b \) the index of the bytes. Bernstein observed that the indexing of those tables is therefore dependent on the secret key. Since not all parts of every table are stored in the cache at once, cache-hits and cache-misses will hence occur depending on the key and input values. This observation holds in particular for the input of the first round as stated in Equation 1. From this, one can conclude that the secret key and the execution time are correlated and that a simple statistical attack provided with can conclude that the secret key and the execution time are.

In the profiling phase this insight is divided into three phases.

The first phase of the attack for again the estimation of the standard deviation of this difference is approximated by

\[ d_{avg,j} = \frac{avg_{b,j} - \sum_{i=1}^{n} \text{timing}(AES_{enc}(p_i, k^0))}{n} \]

with \( p_i \) beeing the plaintext used for sample \( i \). Finally the standard deviation of this difference is approximated by

\[ ds_{b,j} = \frac{s_{b,j}}{\sqrt{\text{total}_{b,j}}} \]

assuming a probability distribution for the timing that is approximately normal and also ignoring the deviation of the total average.

In the attack phase these steps are repeated for the target of the attack for again \( n \) timing samples. It is worth noting that this phase is the only one that has to be done online. The first one can be done in advance while the last phase can be executed at an arbitrary point in time after the second phase is finished.

Once the attack phase is completed, the attacker can launch the correlation phase. He now uses the information he has gathered to compute the sample based correlation coefficient for the data from the profiling and attack phase without the regularisation term by

\[ \text{corr}_{b,i} = \frac{255}{\sum_{j=0}^{255} (d_{avg,j}^0 \ast d_{avg,j}^S)} \]

for every byte \( b \) and every possible value \( i \) of the key. The subscripts 0 and \( S \) are hereby denoting the values computed for the 0-key and the secret key respectively. Having computed these correlations, Bernstein assumes all values with a correlation above a well defined threshold to be possible candidates for the correct value of \( b \). The output of the attack is then a table containing the number of possible candidates for each key byte and a list of the possible values ordered from highest to lowest correlation.

In order to understand why the correlation works as a se-
4. PIKEOS

In this section, the real-time micro kernel operation system PikeOS, which is developed by SysGo\textsuperscript{1}, is presented together with the corresponding Linux Distribution ELiNoS and the tool chain for development of applications for PikeOS. The section starts with a brief overview over relevant definitions for the real-time context.

4.1 Definitions

The following definitions are reported based on those given by SysGo\textsuperscript{2}. It is necessary to understand those definitions in order to understand the following descriptions of PikeOS and its scheduler correctly.

**Deterministic** The ability of a system to respond within a defined interval of time

**Jitter** The variance of the response time of a system

**Latency** The delay between an event and the system’s response to it

**Hard Real-Time** The property of a system to always respond in a deterministic way for a defined latency and known jitter. This corresponds to the notion of a computation’s value decreasing to zero after a defined time.

**Soft Real-Time** The property of a system to use best effort strategies in order to behave in a deterministic way. If the system does fail to do so, this must not result in a catastrophic failure.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{architecture.png}
\caption{The architecture of an embedded system using the PikeOS micro kernel operation system. The three different separated security levels are illustrated as well as the ability to combine different types of personalities in different partitions on top of the separating PikeOS hypervisor.}
\end{figure}

**RTOS** A Real-Time Operating System is a low-level system kernel that supports the development and execution of hard or soft real-time applications.

**GPOS** A General Purpose Operating System is a software platform that supports a great variety of applications and can support, ignore or prevent real-time constraints.

4.2 The Micro Kernel

The PikeOS micro kernel system is designed to allow multiple applications with highly differing requirements in terms of real-time constraints and security or safety to run independently from each other on the same hardware. To achieve this goal, PikeOS makes use of a variety of techniques. To allow for the simultaneous use of multiple applications, such as guest operating systems, APIs or run-time environments, PikeOS makes use of Embedded Virtualization. This means that the available resources of the hardware, like memory, I/O-devices or CPU-time, are split up into several partitions to which access is only allowed to the application assigned to this partition. Therefore, the simultaneous execution of mutually untrusted applications is possible, since no application can access or alter the resources of another. Furthermore, the assigned resources can be matched to the needs of every VM and by that the total amount of hardware needed for more complex systems is reduced. This also affects the integration of new software which is also eased by this design.

To implement this concept, PikeOS makes use of paravirtualization so that the micro kernel acts as an abstraction layer between hardware and application. The access to the system’s resources is hence mediated by the micro kernel, which can be seen as a hypervisor. Every attempt to execute a privileged instruction or access resources outside of an application’s own partition is evaluated by the micro kernel first and then either allowed or denied. As the micro kernel only comes with the basic functionality needed to manage the partitions and resources, such as scheduling, context switches, process communication and synchronization etc.,
the trusted code base for each application is significantly reduced in comparison to monolithic kernels. Since all drivers run in user-space, also the software complexity is minimized as each application can chose which functionality of the system it needs. This architecture is a strong advantage both security- and safety-wise as it already counteracts a lot of possible inter-application attacks exploiting shared resources or unmonitored execution of privileged instructions. It also eliminates the possibility of unintended interference between applications and consequent failures. Since the trusted code base is very small this also makes the analysis of applications in terms of security a lot more feasible and reliable. The concept of Embedded Virtualization in PikeOS is illustrated in Figure 3.

As mentioned above, partitions in PikeOS can host a wide range of different applications, so-called personalities. These include among others:

OS/RTOS
- ELinOS, µTRON, Windows, Legacy RTOS

RTE
- POSIX, Real-time Java, Ada

API
- ARINC 653, PikeOS native API

PikeOS also offers support for multi-core systems and is in fact, due to its partitioning concept, well fit for use of and development for multi-core architectures. As its design is hardware agnostic, i.e. independent of specific features and properties of the underlying system, it is available for all important architectures, including PowerPC, x86, ARM and SPARC. For each of these architectures, PikeOS supports various boards of all main producers on the market.

It was already mentioned above that PikeOS offers strong security benefits. Its security concept follows the Multiple Independent Levels of Security (MILS) [7] standard and divides the whole system into three layers. The upper layer consists of the partitions running on the separation kernel that in turn forms the intermediary layer. The hardware layer is the lower layer. The PikeOS kernel mediates between the different partitions and between the VM-layer and the hardware layer as was explained above. In that way, the security of the upper and the lower level is independent of the other layer’s security, respectively. As the hypervisor does not rely on specific hardware properties, it is also not dependent on the other two layers. This concept is shown in Figure 3. To not only prevent the unauthorized access to other resources but to also make secure inter-partition communication possible, PikeOS offers the possibility to declare parts of the memory to be shared between specified partitions. It also provides different kinds of communication channels via which messages can be exchanged between partitions. For all these means of communication, fine grained privileges can be set to implement different modes of communication and security policies. As PikeOS follows the MILS standard, certification according to the Common Criteria is possible up to EAL 7. The required formal verification of the kernel is currently produced by SysGo.

The elaborate security concept together with its real-time support make PikeOS an interesting candidate for realizing a virtualization-based security architecture which is why it was chosen for this work.

4.3 ELinOS

SysGo offers its own embedded Linux distribution ELinOS\(^3\) that works out-of-the-box with PikeOS and is adapted to the requirements of embedded systems.

In its current version, ELinOS is based on the stable linux kernel version 2.6.4 and supports all common hardware architectures, such as ARM or x86, both for single- and multicore architectures. ELinOS can be fully customized according to the needs of the developer. It features open source real-time extensions, several boot strategies and file systems like Ext3 or NTFS. Furthermore, it supports various communication techniques like Ethernet, WLAN, Bluetooth and UMTS. To be applicable in industrial automation settings, it provides support for standards like USB, IPv6, CAN or Wi-Fi. ELinOS also supports security features like VPN or an integrated rules firewall.

To make development with ELinOS as easy as possible, a number of project templates is provided that can be adapted according to the project’s requirements with little effort. Also, over 2000 precompiled binaries and libraries are included with ELinOS.

Due to the convenient development process and it being the natural fit for PikeOS, ELinOS was used in the context of this work to implement one of the applications in the previously introduced scenario.

4.4 The Tool Chain

Along with PikeOS and ELinOS, SysGo offers its own IDE for development of embedded systems as well as a graphical configuration tool for ELinOS projects. The IDE is called CODEO\(^4\) and is based on the well known Eclipse IDE. It supports Java and C/C++ and includes all of the features for which Eclipse is known. In addition to this, CODEO adds functionality to support the development process for embedded systems.

CODEO offers a graphical front end to configure the architecture of the system, i.e. to add and delete partitions, assign applications to partitions and configure drivers and I/O channels. Another front end exists via which the inter-partition communication can be configured, priorities and CPUs can be assigned to applications and the scheduler’s settings can be adjusted. To analyze a system during runtime, CODEO provides a System Tracing tool via which the timing behaviour of every application can be examined precisely making use of event filters. Information about the state of the kernel and the partition can be gathered by the System Monitoring tool, which also allows for the remote changing of a partition’s state. The situation may arise where the hardware an application is supposed to be using is not available at development time. Therefore, CODEO also offers emulation of the hardware of all supported architectures. Furthermore, remote debugging is supported.

To simplify the development of ELinOS applications, the Embedded Linux Konfigurator ELK provides a graphical front end for the cloning of existing projects and the consequential configuration. It enables the developer to select the target’s hardware and include or exclude all features of ELinOS as it is necessary for the project. Also the kernel’s filesystem can be easily adapted. Like CODEO, ELK comes with all

\(^3\)http://www.sysgo.com/products/elinos-embedded-linux/
required compilers and linkers and selects the right ones automatically. Both tools were used for the implementation work done in the context of this work.

### 4.5 Scheduling in PikeOS

As PikeOS makes it possible for multiple applications of differing kinds to run simultaneously on the micro kernel, it has to take into account different timing constraints of these applications. It is for example very well imaginable that an application with hard real-time requirements runs in parallel to an application with only soft real-time constraints or even no need for real-time behaviour at all. To overcome the problem that this variety induces, 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 other time slot’s lengths. Now, every application is periodically scheduled for the length of the slot it’s assigned to. As every VM 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 would be 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 could 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. One can observe that it may very often occur that a real-time application’s processes finish before the time slot ends or that it does not have any processes to run at all. As it would harm the temporal determinism, the scheduler can not simply switch to the next application in this situation. Rather than to waste this time, the PikeOS scheduler uses this excess CPU time to schedule applications with no real-time constraints. To do so, 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. The PikeOS scheduler and its advantages compared to other schedulers are illustrated in Figure 4.

### 5. DISCRETE TIME COUNTERMEASURE

It is common knowledge in the field of embedded security that no countermeasure is unbreakable. Hence the reasonable approach is to make attacks infeasible in that they cost more than the attacked asset is worth rather than to try to prevent an attack completely. Also, one main pitfall of novel countermeasures is that they sometimes require too much changes to already established systems to be practically relevant. 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 little changes and inducing as little overhead as possible. The countermeasure is presented in the context of the scenario described in Section 3. Assume the rich OS and the trusted domain 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 the 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 mounts 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 both are 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 attacker for one encryption $t_{\text{enc}}$ is now given by

$$t_{\text{enc}} = n \cdot t_{\text{OS}} + m \cdot t_{\text{serv}} + \delta$$

with $t_{\text{OS}}$ being the length of the time slot of the rich OS and $t_{\text{serv}}$ being the length of the time slot of the AES server. The value $\delta$ represents the remaining time in the slot of the rich OS in which the encryption was requested plus the time passing in the first slot of the rich OS after the encryption is done before the attacker’s process is scheduled. The two variables $n$ and $m$ are positive integers. As $\delta$ is independent of the AES server and especially any cache-lookups done by it, it can be viewed as noise and the exploitable time $t_{\text{exp}}$ is given by

$$t_{\text{exp}} = n \cdot t_{\text{OS}} + m \cdot t_{\text{serv}}$$

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. A number of encryptions as needed for an attack will however take a significantly larger amount of time. This already will make an attack time-wise infeasible. 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 attack to distinguish the correct key hypothesis from false ones and will increase the number of needed samples. Hypothetically, the number of samples needed will increase at least quadratically compared to the number of samples needed without the countermeasure. Therefore the discrete time countermeasure is a strong shield against the kind of attacks considered here. Beside its strong effect, the countermeasure requires no change of any kind in the code and also causes arguably only little overhead timing wise. 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. Furthermore, in this setting it has only two parameters to tune.

6. THE IMPLEMENTATION

In this section, the implementation done for this work will be explained. After a brief description of the hardware that was used, an overview over the implementation and the protocol will be given. After that, the implementations of AES server and attacker will be presented in closer detail.

6.1 Overview

The implementation is based on the scenario presented in Section 3. The rich OS is implemented using the embedded Linux distribution ELinOS including the necessary code for the attacker to conduct the timing attack. The AES server 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 message ports and a shared 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 keys, plain- and ciphertexts.

The board that was used in this work is a Freescale i.MX6 Quad board which has the following CPU architecture:

- Quad-Core ARM Cortex-A9 with up to 1.2 GHz
- 32 KB I and D Cache per Core
- 1 MB shared L2 cache

The code was written on a Ubuntu 13.10 workstation using the CODEO IDE and ELK. The board was connected to the workstation via ethernet to load the compiled images. A picture of the board is shown in Figure 5.

6.2 Communication

For the technical implementation of the communication between AES server and the rich OS, two different means provided by PikeOS were used - so called queuing ports and a shared memory page. For the logical implementation, a simple protocol was set up. Queuing ports are communication channels that can be set up between two partitions in advance using the CODEO IDE and then initialized at run-time by the applications. These channels are unidirectional and owe their name to the fact that they treat incoming messages by FIFO and hence queue them until all of them are received. They are non-blocking, so the receiver has to check an incoming port periodically for the data while the sender just sends his data and does not get any response about the success of the data transfer. As the queuing ports are unidirectional, two ports were needed between the two partitions to enable bidirectional communication. It would have been possible to send both the protocol messages and the plain- and ciphertexts via the queuing ports. This would have however most likely induced a higher level of noise in the timing data than that what could be achieved by plain memory operations. Therefore, a shared memory page was setup between the two partitions. Both partitions were granted read-write access. In PikeOS, a shared page is

![Figure 5: A Freescale i.MX6 Quad board similar to the one used for this work.](http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5655318)

Rich OS

<table>
<thead>
<tr>
<th>Write key to the shared page</th>
</tr>
</thead>
</table>

AES Server

<table>
<thead>
<tr>
<th>New key</th>
</tr>
</thead>
</table>

Read key from the shared page

ACK

![Figure 6: The initialization protocol used to set up the AES server with a key.](http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5655318)
mapped to an arbitrary address in the memory of each partition and can then be used just like normal memory. This approach was therefore not only beneficial with respect to the attack but also made the exchange of plain- and ciphertexts and key data very convenient.

To implement the client-server functionality, a simple protocol was used that is divided into two sub-protocols. The initialization protocol is executed once at the beginning and simply tells the AES server to use the key that has been provided by the rich OS from now on. They key is stored in the shared memory page by the rich OS and then read from there by the AES server. After the server has read the key, he replies with an ACK. The initialization protocol is shown in Figure 6.

The encryption protocol is another simple handshake protocol that is used by the rich OS to request an encryption from the AES server. The rich OS sends a request message to the AES server and writes the plaintext to the shared page. The server receives the message and reads the plaintext. It then encrypts the plaintext, stores the ciphertext back to the shared page and sends an ACK. The protocol is displayed in Figure 7.

### 6.3 AES Server

The AES server was implemented in C using the native PikeOS C API to have as little overhead as possible and reduce the noise level to a minimum. The server works in the following way: Once it is started it first opens the queueing ports and maps the shared page, so that it can access both. After this is done, the server goes into an infinite loop and waits for messages from the client, the rich OS in this case. If it is asked to use a new key, it follows the protocol presented above and then initializes AES with the new key. If it receives an encryption request, it first loads the plaintext from the beginning of the shared page where it is stored by convention. It then encrypts this plaintext and stores the ciphertext in the AES server's shared page and executes the encryption protocol from the AES server.

### 6.4 Rich OS

The rich OS was implemented using the ELinOS embedded Linux distribution, based on a template project called BusyBox. The BusyBox project includes a binary that is started automatically at boot-time and simulates a standard Linux shell. It supports all of the most common commands like ls, cd, grep and the likes. By using the ELK configuration tool, new files can easily be integrated into the file system. This was used to include the cross-compiled attack code as user-space application in the system. To integrate the AES functionality into the kernel, a new kernel module was written. This module acts as an abstraction layer for user-space applications and takes care of the actual communication with the AES server.

The module is loaded by the rich OS kernel at boot-time and starts by initializing some components: it first initializes and maps a character device that is used as shared page between processes in user-space and the module. It also sets up a config file under `/proc/cacheattack/cfg` that can be used by the user-space programs to set a key. After this is done, the module then tries to open the two queueing ports for the communication with the AES server and to map the shared page between the two partitions.

If a user-space program wants to make use of the AES encryption functionality, this now works as follows. The program first opens the character device created by the module to use it as shared page. Then, it writes the key it wants to use and its size to the configuration file. This is noticed by the module, which now copies the key from the config file to the shared page between rich OS and AES server and executes the already discussed intialization protocol. From now on, the user-space application can use the AES encryption service by writing the plaintext to the character device and using the new syscall `sys_enc_doit`. The module then copies the content of the user-space shared page to the AES server’s shared page and executes the encryption protocol that was explained above. Once the encryption is finished, the ciphertext is copied from the AES server’s shared page.

---

**Figure 7:** The encryption protocol used to request an encryption from the AES server.

**Algorithm 1:** The AES server’s main loop.
create /proc/cacheattack/cfg;
map read/write functions for cfg;
create character device;
map character device;
while not in-port and out-port found do
  | find open port;
  | if open port is out-port then
  |   set open port as out-port;
  end
  | if open port is in-port then
  |   set open port as in-port;
end
open server’s shared page;
map server’s shared page;
Algorithm 2: The rich OS kernel module’s initialization function.
read key and key size from cfg;
store key in server’s shared page;
send do_setup to server;
send key size to server;
receive message;
while message is not ACK do
  | receive message;
end
Algorithm 3: The rich OS kernel module’s key setup function.

In this section, the results of the experiments conducted in the context of this work will be presented. First, an evaluation of the attack itself with respect to a variety of parameters will be presented. Those results will then be compared to results achieved by Weiss et al. in [1]. After that the results of experiments with the proposed countermeasure are stated and those results are subsequentially compared with other recent approaches.

7.1 Evaluation of the Attack on PikeOS
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:

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

The first criterion 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 criterion. In the best case only one candidate, namely the correct one, remains for each byte and the key is hence revealed completely. But also a significant reduction is already valuable to the attacker as he knows the remaining possible values and can launch a brute-force attack in the reduced key space. 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 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 yield a much lower average run-time than the standard \(O(n^2)\), \(n\) being the number of key candidates.

For all the experiments presented in this section, 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
Figure 8: A histogram describing the numbers of possible candidates for all bytes of the key for varying packet sizes. The x-axis lists all key bytes from left to right. On the y-axis, the results for 200, 400 and 600 bytes packet size are displayed respectively. The z-axis shows the number of remaining possibilities.

Table 1: The average position of the correct key byte candidates for the different packet sizes.

<table>
<thead>
<tr>
<th>Packet Size</th>
<th>Average Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>3.625</td>
</tr>
<tr>
<td>400</td>
<td>4.5</td>
</tr>
<tr>
<td>600</td>
<td>5.25</td>
</tr>
</tbody>
</table>

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 done with 512 million samples to have approximately 2 million samples for each possible key candidate.

7.1.1 Evaluation of different Packet Sizes

When Bernstein introduced his attack in [2], one trick he used to obtain the complete key solely was to vary the size of the packets that were send from the client to the AES server. Although only the first 16 bytes of each packet were used as plaintext, the differing sizes caused different timing behaviours. This led in turn to a different behaviour of the correlations for each key byte. So, by varying the packet sizes, Bernstein was able to pinpoint all the key bytes over time.

To analyze if this phenomena also plays a role for the setting examined here, three different packet sizes were evaluated: 200, 400 and 600 bytes. The other parameters, such as assignment of the applications to CPU cores, number of samples etc. were kept constant. The results are displayed in Figure 8 and Table 1.

As it can be clearly seen, a higher packet size causes both the numbers of possible key byte values to rise and a reduction in the average position of the correct candidates. The increase in the number of candidates is minor between 200 and 400 bytes but drastic between 400 and 600. The average position of the correct hypotheses is lowered roughly by one position per step. This means that in the virtualization setting, no informatin is gained by variation of the packet size. It is even harmful to increase it. This is most likely the case because a higher packet size might lead to a higher amount of time spent on copying the data. This might cause timing variations that are independent of the key and hence increase the noise.

7.1.2 Evaluation of different Clipping Thresholds

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 behaviour of his implementation. This value was therefore changed in this implementation. To evaluate the effect of this clipping, two different thresholds were investigated. 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. This was done for packet sizes of 200 and 400 bytes while all other parameters were left untouched again. The results are display in the Figures 9 and 10 and the Tables 2 and 3.

The results clearly show that the lower threshold leads to a significantly lower success rate for both packet sizes. This implies that the timings lying in the interval between the
thresholds indeed contained information about the key. Although timings vary between different implementations and hardwares, this finding underlines the fact that methods to manipulate the gathered data should only be used with high caution. Application of such methods based on a rule of thumb or intuition might in fact degrade the success rate severely as it was seen.

### 7.1.3 Evaluation of 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 while for the second one both partitions were run on all four cores. The third configuration involved the AES server running on all four cores while ELinOS was assigned only one core. The results are depicted in Figure 11 and Table 4.

It can be seen that configuration 1 gave the best result for both criteria, as scenario 2 yielded the worst. This is understandable since in scenario 1, only one L1 cache and the L2 cache are used to store the T-tables while in scenario 2 the T-tables are most likely scattered over the four L1 caches and the L2 cache. This decreases the the signal to noise ratio with high certainty and thus lowers the success rate. Additionally, as both the rich OS and the AES server use the same core their cache use will interfere which also reduces the quality of the timing samples. This effect is visible in the difference between scenario 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.

### 7.1.4 Evaluation of dedicated Cores

As it was mentioned above, the fact that both partitions use the same CPUs might be a disadvantage for the attack. 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. These results were contrasted with the result achieved with both partitions running on the same single core. Again, all other parameters were left untouched. The results are shown in Table 5 and Figure 12.

Interestingly, the use of dedicated cores leads to a slightly better success rate in terms of the total number of remaining key candidates while yielding an also slight decrease in the average position of the correct key byte values. As it can be seen, assigning one core to each partitions thereby results in a slightly better success rate than using two dedicated cores. This might be explained by the already discussed effect of using multiple L1 caches. Why the number of key candidates decreases while the average position increases slightly...
is however more difficult to answer. The first effect is understandable as the interference between the applications is removed. The second one might either be coincidence or caused by a phenomenon that would need to be investigated further.

7.1.5 Evaluation of different Numbers of Samples

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 as the noise gets averaged out more and more, leaving only the relevant information behind. To verify this assumption, the attack was conducted with 256, 512 and 1024 million samples. The results are displayed in Figure 13 and Table 6.

As it is shown, increasing the number of samples does in fact also increase the success rate of the attack. This does not come as a surprise as the relation between these two factors is clear. However, it is noteworthy that the increase between 512 and 1024 million is far lower than the one between 256 and 512 million. It seems as if the increase of the success rate shows a logarithmic behaviour. This seems plausible intuitively but would need further investigation into the theoretic limits of Bernstein’s attack. Bernstein himself did not attempt any such analysis to the best knowledge of the author.

One possible explanation for this phenomenon can however be given by taking the cache architecture into account. As only the upper \( k \) bits of a data word are used to index the cache lines, the timing behaviour 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. This hypothesis is supported by the findings of Neve et al. in [8].

7.1.6 Evaluation of different Keys

Since all previous experiments were done using the same key, it was interesting to examine how well the attack would work for a different key. To evaluate this, an alternative key was attacked and the result was compared to the one achieved for the old key. To make the comparison possible, all parameters were the same for both attacks. The results are depicted in Figure 14 and Table 7.

The results are interesting in that they are identical for the reduction of the key space but significantly different in the average position of the correct hypotheses. It seems reasonable that there is no big difference in the number of possible key byte values for two random keys with otherwise identical
attack setup. The same holds for the other criterion. Why
there is then such a difference may have multiple reasons. It
may very well be the case that some keys are “easier” than
others or that different combinations of key byte values lead
to a cache timing behaviour that is better exploitable. It
might also just be coincidence. To answer this question, fur-
ther research in this direction would be necessary. However,
the results verify that the attack works for independently
and randomly chosen keys and by that underline its gener-
ality.

7.2 Comparison to the Fiasco Micro Kernel

In [1], Weiss et al. present results for Bernstein’s attack
carried out in a very similar virtualization setting. In con-
trast to the hardware presented above, Weiss et al. used
a beagleboard that is based on a Cortex-A8 with 720 MHz.
To implement the virtualization scenario, the Fiasco.OC mi-
cro kernel together with L4Re was used. By making use of
L4Tasks, an address space separation similar to the one done
in PikeOS was implemented. Figure 15 depicts a diagram
of the implementation done by Weiss et al. As it can be
seen, the implementations are almost identical. Hence, the
results provided in [1] are well suited for a comparison with
the result presented above. Note that the same key was used
in [1] as it is here.

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 im-
plementation of Weiss et al. Nevertheless, only half of the
bytes could be reduced that far while for the implementa-
tion of [1], 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 com-
Table 8: The average number of clock cycles needed for one encryption for the standard configuration and the countermeasure.

<table>
<thead>
<tr>
<th>Scheduling Scheme</th>
<th>Average Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority-Based</td>
<td>149,000</td>
</tr>
<tr>
<td>Countermeasure</td>
<td>250,000</td>
</tr>
</tbody>
</table>

mon 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 [1]. The two results are compared in Figure 16.

As both implementations are very similar, the difference in the results is most likely caused by the different hardware and the different micro kernel in use. Both factors have a huge influence on the timing behaviour. Since the two results are fundamentally different, it is not possible to say if one setup is more vulnerable or secure than the other. The total number of possible keys was reduced to $2^{72}$ for the PikeOS setup and to roughly $2^{70}$ for the Fiasco setup showing a slight advantage for PikeOS. However, the results are close and in order to gain more insight, a comparison between the positions of the correct key values in the correlation rankings would have to be done.

7.3 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 behaviour of the system indistinguishable from the behaviour with simple priority-based scheduling was found to be to set the slot length to 5 ticks for both partitions. Using this configuration, single AES encryptions could be done without noticeable delay. 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. The attack was then launched. After 10 days the profiling phase was about 66% finished and due to the time constraints of this work the attack was aborted. The profiling phase would have needed at least 2 weeks to finish and the same therefore holds for the attack phase. This means that the total run-time of the attack is about one month for the above configuration of the scheduler. Remember that due to the different timing behaviour 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 assume, that for the attack to produce a useful output at least twice the number of samples and hence twice the time is needed. Even if the attacker would do the profiling phase offline, 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 timeframe. 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 8. For a more thorough evaluation of the discrete time countermeasure, additional experiments need to be conducted. It is clear however that this approach is one well worth investing further effort.

7.4 Comparison to other Countermeasures

In [3] and [4] two novel countermeasures against cache based attacks are introduced. Since these countermeasure target the same class of attacks, it is interesting to compare them to the discrete-time countermeasure. As the focus of this work was put on time-driven attacks the comparison will focus on this aspect, too.

The STEALTHMEM countermeasure that was explained in Section 2 tries to prevent both active and passive time- and access-driven attacks. To that end, it uses dedicated cache lines in the shared cache for each CPU. Like in this work, a virtualization environment is assumed. The hypervisor is extended with a special driver offering an API to the VMs that manages the access to the dedicated cache lines. This makes it clear that there is a strong difference between the discrete-time countermeasure and STEALTHMEM. While the countermeasure proposed here does not need any modification of the system or the cryptographic algorithms, an implementation of STEALTHMEM for Windows Server 2008 R2 with Hyper-V needed 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. This shows that in order to set up STEALTHMEM, a lot of work needs to be done. For the discrete-time countermeasure on the contrary only a reconfiguration of the scheduler is needed.

Furthermore, the encryption software has to be modified to make use of the stealth cache lines via the provided API causing both timing and code size overhead. The modification of the algorithms also is a potential pitfall. If not done correctly, some leakage remains and therefore breaks the countermeasure. While the countermeasure presented here also causes some timing overhead, the overhead in terms of code is zero as no modifications are needed. Even with this amount of work, STEALTHMEM can not ensure that a time-driven attack becomes impossible. In the optimal case for AES, all T-Tables or the S-box could be stored in the stealth pages and remain there for the rest of the execution. This assumes however that the stealth cache lines indeed provide enough space to store the whole table(s). If this is not the case, cache evictions will still occur and a time-driven attack can be mounted, possibly needing more samples.

The discrete time countermeasure on the other hand side does not rely on any characteristic of the cache. It masks the actual timings and hence reduces the information contained in the samples in any case.

The instruction-based scheduling theme aims to prevent cache based attacks by using some specified number of executed instructions as scheduling criterion. This results in the minimization or avoidance of scheduling-induced race conditions between processes that arise due to the dependency of the
execution time on the cache content. Both methods are very similar in that they use a fixed value as their criterion for the scheduling. However, using the number of executed instructions has some disadvantages. While this approach suffices to prevent a simple attack that explicitly tries to exploit this phenomenon it is not enough to prevent time-driven attacks such as Bernstein's. This is due to the fact that the instruction-based scheduling does not prevent that the execution time of instructions is dependent on the cache content. Neither does it mask this information in a way like the discrete time countermeasure does. Therefore, the overall execution time of for instance an AES encryption will still leak the full information. This is a severe disadvantage compared to the countermeasure proposed here. Also, since instruction-based scheduling is a novel approach it is not supported by current micro kernels and hence would have to be either integrated into an existing system or implemented completely new. This would then again cost a lot of effort. As already mentioned, this is not the case for the discrete time method. 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.

8. CONCLUSION

In the previous sections, Bernstein’s time-driven cache based side channel attack was reintroduced and the PikeOS micro kernel system was presented. A countermeasure against time-driven attacks that leverages the scheduler’s properties was devised and a prototypical implementation of a security virtualization architecture based on PikeOS was presented. The timing attack and the countermeasure were analyzed in the context of this setting and compared to related approaches and results.

As it has been shown, Bernstein’s attack poses a threat to virtualization scenarios implemented with PikeOS as it was able to reduce the key space to a size that allows for brute-force attacks. It was also shown that on average the correct key can be found by just searching through all combinations of the top 5 to top 10 values in the lists of key byte candidates. This yields an algorithm to find the key that is significantly faster than a standard brute-force search. As different configurations for the scheduler were examined it became clear that in order to protect a system from time-driven attacks it is useful to use all cores for the encryption server/application and to share these cores with other applications. This increases the noise in the data and hence makes it harder for the attack to extract the right key. Unfortunately this also increases the danger of an active cache based attack so that the configuration of the scheduler must be done carefully. In comparison with the results on a similar setup analyzed in [1] it was found that although the results were very different in detail, the key space was reduced to about the same size. This underlines the fact that time-driven attacks are due to their general nature dangerous across different hardware and systems. They therefore have to be taken into account when designing a secure architecture. Note that the implementation of the cryptographic algorithm also has to be chosen carefully. It is obvious from the results achieved in this work that trading memory for run-time comes at the high price of vulnerability against timing attacks.

It was found that the proposed countermeasure indeed is able to raise the cost of an attack to a level that makes the attack infeasible or unprofitable for the attacker. Although it does not completely rule out the possibility of an attack, it lowers the risk significantly while demanding no modification of the existing system. This makes it especially suitable for scenarios where the value of an asset is not exceedingly high and simple integration is beneficial, as for instance in the mobile device market. To investigate this approach further, experiments for different system architectures, hardwares and kernels should be done.

As the importance of virtualized security architectures will increase in the future, it is important to examine their advantages and disadvantages. This work showed that countermeasures against side channel attacks are very relevant in this context. Further research on this topic should therefore be conducted.

9. REFERENCES


