A Cache Timing Attack on AES in Virtualization Environments

Michael Weiß*, Benedikt Heinz*, and Frederic Stumpf*

Fraunhofer Research Institution AISEC, Garching (near Munich), Germany
{michael.weiss, benedikt.heinz, frederic.stumpf}@aisec.fraunhofer.de

Abstract. We show in this paper that the isolation characteristic of system virtualization can be bypassed by the use of a cache timing attack. Using Bernstein's correlation in this attack, an adversary is able to extract sensitive keying material from an isolated trusted execution domain. We demonstrate this cache timing attack on an embedded ARM-based platform running an L4 microkernel as virtualization layer. We also show that an attacker who gained access to the untrusted domain can extract the key of an AES-based authentication protocol used for a financial transaction. We provide measurements for different public domain AES implementations. Our results indicate that cache timing attacks are highly relevant in trusted execution environments.

Keywords: Virtualization, Trusted Execution Environment, L4, Microkernel, AES, Cache, Timing, Embedded

1 Introduction

Virtualization technologies provide a means to establish isolated execution environments. Using virtualization, a system can for example be split into two security domains, one trusted domain and one untrusted domain. Security critical applications which perform financial transactions can then be executed in the trusted domain while the general purpose operating system, also referred to as rich OS, is executed in the untrusted domain. In addition, other untrusted applications can be restricted to the untrusted domain.

It is generally believed that virtualization characteristics provide an isolated execution environment where sensitive code can be executed isolated from untrustworthy applications. However, we will show in this paper that this isolation characteristic can be bypassed by the use of cache timing attacks. A cache timing attack exploits the cache architecture of modern CPUs. The cache architecture has influence on the timing behavior of each memory access. The timing depends on whether the addressed data is already loaded into the cache (cache-hit) or

* The authors and their work presented in this contribution were supported by the German Federal Ministry of Education and Research in the project *RESIST* through grant number 01IS10027A.

it is accessed for the first time (cache-miss). In case of a cache-miss, the CPU has to fetch the data from the main memory which causes a higher delay compared to a cache-hit where the data can be used directly from the much faster cache. Based on the granularity of information an attacker uses for the attack, cache timing attacks can be divided into three classes: time-driven [7, 16, 2, 15], trace-driven [1, 9] and access-driven [16, 14]. Time-driven attacks depend only on coarse timing observations of whole encryptions including certain computations. In this paper, we use a time-driven attack which is the most general attack of the three. To perform a trace-driven attack, an attacker has to be able to profile the cache activity during a single encryption. In addition, he has to know which memory access of the encryption algorithm causes a cache-hit. More fine grained information about the cache behaviour is needed to perform an access-driven attack. This attack additionally requires knowledge about the particular cache sets accessed during the encryption. That means that those attacks are highly platform dependent while time-driven attacks are portable to different platforms as we will show.

Although trace- and access-driven cache attacks would be feasible in a virtualized system, it would require much more effort to setup a spy-process. For an access-driven attack, the adversary needs the physical address of the lookup tables to know where they are stored in memory and thus the information to which cache lines they are mapped. This cannot be accomplished by a spy-process during runtime in the untrusted domain, as there is no shared library. By a time-driven attack, it is sufficient to see the attacked system as a black box.

Bernstein [7] for instance used this characteristic for a known plaintext attack to recover the secret key of an AES encryption on a remote server. However, Bernstein had to measure the timing on the attacked system to get rid of the noisy network channel between the attacked server and the attacking client. While this is a rather unrealistic scenario since the server needs to be modified, it is very relevant in the context of virtualization. In the context of virtualization, the noise is negligible since local communication channels are used for controlled inter-domain data exchange. These communication channels are based on shared memory mechanisms which introduce only a small and almost constant timing overhead.

This paper is organized as follows. In the next section we state related works. We analyze the general characteristics of a virtualization-based system and present a generic system architecture that provides strong isolation of execution environments in Section 3. We believe that this system architecture is representative for related architectures based on virtualization that establish secure execution environments. Based on this architecture, we show the feasibility to adapt Bernstein's attack. Further, in Section 4, we show that standard mutual authentication schemes based on AES are vulnerable to cache timing attacks executed as man-in-the-middle in the untrusted domain. We provide practical measurements on an ARM Cortex-A8 based SoC running the Fiasco.OC microkernel [22] and its corresponding runtime environment L4Re as virtualization

layer to confirm our proposition in Section 5. Finally, we conclude with a discussion about the results and possible countermeasures in Section 6.

2 Related Work

Bernstein provides in [7] a practical cache-timing attack on the OpenSSL implementation of AES on a Pentium III processor. He describes a known plaintext attack to a remote server which provides some kind of authentication token. However, Bernstein does not provide an analysis of his methodology and an explanation why the attack is successful. This is revisited by Neve et al. [15]. They present a full analysis of Bernstein's attack technique and state the correlation model. Later Aciiçmez et al. [2] proposed a similar attack extended to use second round information of the AES encryption. However, they also provide only local interprocess measurements in a rather unrealistic attack setup similar to Bernstein's client-server scenario. Independently from Bernstein, Osvik et al. [16] also describe a similar time-driven attack with their *Evict+Time* method. Further, they depict an access-driven attack *Prime+Probe* with which they are able to extract the disk encryption key used by the operating system's kernel. However, they need access to the file system which is transparently encrypted with that key.

Ristenpart et al. [19] consider side-channel leakage in virtualization environments on the example of the Amazon EC2 cloud service. They show that there is cross VM side-channel leakage. They used the *Prime+Probe* technique from [16] for analyzing the timing side-channel. However, Ristenpart et al. are not able to extract a secret encryption key from one VM.

There are also more sophisticated cache attacks which can recover the AES key without any knowledge of the plaintext nor the ciphertext. Lately, Gullasch et al. [14] describe an access-driven cache attack. They introduce a spy-process which is able to recover the secret key after several observed encryptions. However, this spy-process needs access to a shared crypto library which is used in the attacked process. Further, a DoS attack on the Linux' scheduler is used to monitor a single encryption. Recently, Bogdanov et al. [8] introduced an advanced time-driven attack and analyzed it on an ARM-based embedded system. It is a chosen plaintext attack which is using pairs of plaintexts. Those plaintexts are chosen in a way that they exploit the maximum distance separable code. This is a feature of AES used during `MixColumns` operation to provide a linear transformation with a maximum of possible branch number. For 128-bit key length, they have to perform exactly two full 16-byte encryptions for each plaintext pair where the timing of the second encryption has to be measured.

Even though these attacks could be demonstrated in a virtualization-based system, it would require strong adaptations of the system which may result in an unrealistic attacker model. In contrast, the approach by Bernstein is more flexible and provides a more realistic attacker model for a trusted execution environment.

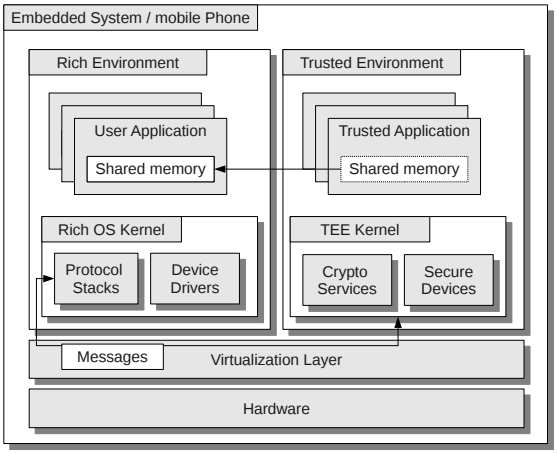


Fig. 1. High level security architecture of an embedded device based on virtualization

3 System Architecture

We present in this section the system architecture of a generic virtualization-based system. This system architecture is representative for other systems based on virtualization and is later used to demonstrate our cache timing attack.

The system architecture consists of a high level virtualization-based security architecture including the operating system and an authentication protocol used to authenticate a security sensitive application executed in the trusted domain.

3.1 Virtualization-based Security Architecture

Virtualization techniques can be used to provide strong isolation of execution environments and thus enables the construction of compartments. One compartment can then be used to execute sensitive transactions while the other compartment is used for transactions with a lower trust level. This design process is already partly employed by smartphone architectures. The Dalvik VM on Android provides some sort of process virtualization [20, p. 83], however, without providing the same level of isolation achieved by system virtualization [20, p. 369]. Due to the insecurity of current smartphones' and other embedded systems' architectures, it is expected that virtualization solutions will be used in the near future to increase security and reliability. This assumption is supported by current developments in the embedded hardware architectures (ARM TZ [3], Intel Atom VT-x [11]).

GlobalPlatform is currently in the process of specifying a high level system architecture of a trusted execution environment (TEE) [4]. The security architecture is mainly adapted from the TEE Client API Specification [13]. At the time of this writing, this is the publicly available part of the complete specification. It is shown in Figure 1. The system architecture consists of two execution

Table 1. Mutual authentication protocol using symmetric AES encryption

Verifier B		**Prover A**
shared key: k		shared key: k
$r_B := rnd()$		$r_A := rnd()$
	$\xleftarrow{\text{connect}()}$	
	$\xrightarrow{ID_B, r_B}$	
		$m_A := h(r_B \|\| r_A \|\| ID_A)$
	$\xleftarrow{ID_A, r_A, c_A}$	$c_A = E(m_A, k)$
$m'_A := h(r_B \|\| r_A \|\| ID_A)$		
$c_A \stackrel{?}{=} E(m'_A, k)$		
$m_B := h(r_A \|\| ID_B)$		
$c_B := E(m_B, k)$	$\xrightarrow{c_B}$	
		$m'_B := h(r_A \|\| ID_B)$
		$c_B \stackrel{?}{=} E(m'_B, k)$

domains, the trusted execution environment for the trusted applications and the rich environment for the user controlled rich operating system[1]. It is much more likely that the rich environment is infected by malware due to the greater software complexity. The trusted applications are either executed in their own virtual machine or are separated in different address spaces and do not share any memory to allow the deployment of trusted application by different vendors which may not trust each other. However, each trusted application depends on the security of the underlying isolation layer.

3.2 Authentication Scheme

To keep the trusted computing base (TCB) small and to reduce implementation complexity, the drivers and communication stacks are implemented in the rich operating system executed in the untrusted domain. Thus, to achieve for example authenticity of a transaction in an online banking application, a protocol resistant to man-in-the-middle attacks has to be used. The protocol's end point has to be in the trusted domain and not in the rich OS since the rich OS could be compromised. When the trusted application wants to communicate with its backend system, it has to prove its authenticity against the backend and vice versa. For this purpose, a mutual authentication protocol as shown in Table 1 between both parties needs to be employed. Note that this is only a simple example authentication scheme and also more sophisticated authentication schemes could be used. We assume that both parties have negotiated a secret symmetric key. The protocol uses random nonces as challenges and AES with the shared secret key k to generate the responses. Also an identifier of the particular sender is

[1] A rich operation system is a full operating system with drivers, userland and user interfaces, e. g., Android

Table 2. Timing attack on a trusted application

Untrusted VM		
To/From remote		**To/From trusted**
connect() ←		connect() ←
ID_B, r_B →	startClk()	ID_B, r_S →
ID_A, r_A, c_A ←	stopClk()	ID_A, r_A, c_A ←
	$m_A := h(r_B \| \| r_A \| \| ID_A)$	
	⋮	

included in the encrypted response. Before the execution of the encryption, this ID is concatenated with the challenges. Further, this concatenation is hashed to prevent concatenation attacks.

Both verifier and prover execute the mutual authentication protocol depicted in Table 1. The prover in this case is the trusted application whereas the verifier is a remote backend system. The untrusted domain is not taking part in the protocol and just acts as transparent relay. After execution of this scheme, the prover A has proven to the verifier B the knowledge of the secret k and vice versa. Further, the freshness of the communication is provided by this scheme. This simple mutual authentication is used to demonstrate the vulnerability of virtualization-based trusted execution domains against the timing attack depicted in the next section.

4 Attack Setup

For our attack setup, we focus on a virtualization-based system architecture of an embedded mobile device as stated above. In the following, we show that an attacker who has overtaken the rich OS in the untrusted domain, e. g., by the use of malware, can circumvent the isolation mechanism with a cache timing side-channel.

Our introduced authentication scheme is secure against man-in-the-middle attacks on protocol level. However, due to the fact that the untrusted domain is relaying the messages between the client application and the remote server, malware can use a time-driven cache attack to at least partially recover the AES-encryption key k. To this end, we use a template attack derived from the attack in [7] which is conducted in two phases, first the profiling phase (offline and online) and second the correlation phase. We assume that an attacker has gained access to the rich operating system. The attacker is then able to execute a small attack process which is used to generate the timing profile.

4.1 Profiling Phase

The profiling phase is run twice, one time offline with a known key k and a second time online on the real target with an unknown key k'. However, the malware

program which is running on the attacked system only has to generate the online profile. The profiling phase in this context looks as follows. The attacker process has to hook into the messaging system between rich OS and the trusted execution environment as depicted in Table 2. Since the protocol stack is implemented in the rich OS, this could be done in the rich OS kernel. Thus, the attacker is able to capture the server's challenge r_B and measure the time between relaying this challenge to the client and receiving the client's response message. This provides him the timing of the AES encryption of the known plaintext $m_A = h(r_B||r_A||ID_A)$, of course with the noise introduced by the hashing and other operations executed in addition to the actual encryption.

To recover the key in the later correlation phase, many challenge-response observations are needed to deal with the noise by averaging over all samples. Therefore, the attacker has to increase the number of challenge-response pairs to be collected. For that, he has several options depending on the used implementation of the virtualization layer and the client application. In upcoming TEE implementations, like the GlobalPlatform TEE, an untrusted user application may be used to initiate the trusted application. Thus, malware could initiate the trusted application as well and some kind of trigger application could be used to initiate the authentication process of the trusted application. The following connection request to the remote server can be blocked by the attacker as he has full control over the untrusted rich operating system and thus can intercept any communication. Instead of relaying the connection request to the remote server, the attacker establishes a local fake connection and sends an own generated nonce to the trusted application. After receiving the answer with the ciphertext, the attacker can send a connection reset and depending on how the trusted application is implemented, the protocol will just restart and a new challenge can be sent.

4.2 Correlation Phase

After receiving sufficient challenge-response pairs for the online timing profile, the attacker can correlate the profiles to recover at least partially the key k'. We provide detailed measurement results in Section 5. We use a correlation based on timing information during the first round of AES. It would be possible to also use information from the second round to reduce the amount of samples needed. However, to show that time-driven cache attacks are a threat to virtualization-based systems, it is sufficient to use the easier first round attack.

At first we define the function $timing()$ which computes the timing difference between the start and end of an operation. During the first run of the profiling phase, for each plaintext p, the overall encryption time is stored accumulated in a matrix $\mathbf{t}$ which is indexed by the byte number $0 \leq j < 16$ and the byte value $0 \leq b < 256$.

$$\mathbf{t}_{j,b} = \mathbf{t}_{j,b} + timing(enc\_AES(p, k)) \tag{1}$$

Further, the total amount of captured samples for each plaintext byte value is traced in a matrix **tnum** as shown in Equation 2.

$$\mathbf{tnum}_{j,b} = \mathbf{tnum}_{j,b} + 1 \tag{2}$$

After several samples the matrix **v** which is computed as depicted in Equation 3 is stored in the profile.

$$\mathbf{v}_{j,b} = \frac{t_{j,b}}{\mathbf{tnum}_{j,b}} - t_{\text{avg}} \tag{3}$$

t_{avg} shown in Equation 4 is the accumulated timing measurements of all plaintexts p_m divided by the total number of encryptions l.

$$t_{\text{avg}} = \frac{\sum_{m=0}^{l} timing(enc\_AES(p_m, k))}{l} \tag{4}$$

During the online part of the profiling phase, the matrices $\mathbf{t}'$ and $\mathbf{tnum}'$ are generated and the output $\mathbf{v}'$ is generated for the unknown key k'.

Finally, for every key byte j the correlation **c** for each possible value $0 \leq u < 256$ is computed as shown in Equation 5.

$$\mathbf{c}_{j,u} = \sum_{w=0}^{255} \mathbf{v}_{j,w} \cdot \mathbf{v}'_{j,(u \oplus w)} \tag{5}$$

According to the probability which is derived from the variance also stored in the profile, the values of **c** are sorted. Further, the key values with the lowest probability below a threshold as defined in [7] are sorted out.

5 Empirical Results

For practical analyses of the above described use-case, we built a testbed based on an embedded ARM SoC with an L4 microkernel as virtualization layer. As hardware platform, we decided to use the beagleboard in revision c4 because it is widely spread community driven open source board and also comparable to the hardware of currently available smartphones, for instance the Apple iPhone as well as Android smartphones. It is based on Texas Instruments' OMAP3530 SoC which includes a 32-bit Cortex-A8 core with 720MHz as central processing unit. The Cortex-A8 implements a cache hierarchy consisting of a 4-way set associative level 1 and an 8-way set associative level 2 cache. The L1-cache is split into instruction and data cache. The cache line size of both is 64 byte. For precise timing measurement, we used the ARM CCNT register, which provides the current clockcycles, the CPU spent since last reset. This is a standard feature of the Cortex-A8 and thus also available in current smartphones. However, it needs system privileges by default.

We implemented the scenario shown in Figure 1 and employed the mutual authentication scheme from Table 1 in a trusted environment. For the virtualization environment, we used the Fiasco.OC microkernel and the L4Re runtime

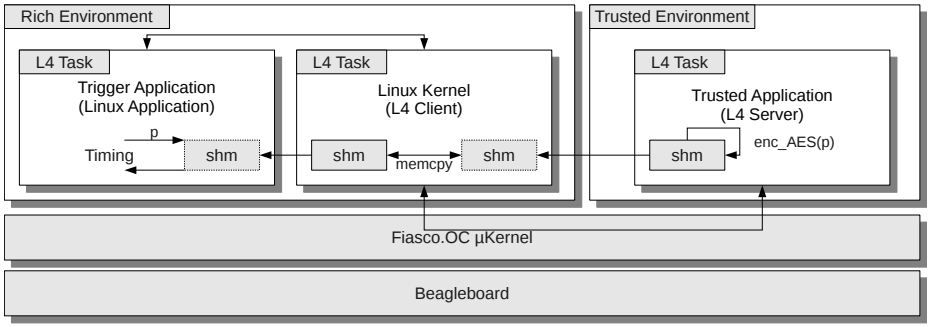


Fig. 2. Linux trigger application (simulating malware) connecting through L4Linux kernel services to trusted application executed as L4Server

environment from TUD's Operating Systems group. Fiasco.OC is a capability-based microkernel. In cooperation with the L4Re, it provides the functionality of a hypervisor for paravirtualized Linux machines. Further, it enables real time application and security applications to run directly on top of the microkernel in separated address spaces (L4Tasks) besides the Linux VMs. In fact, the L4Re virtualization runs Linux in user mode also in an L4Task. Further, each Linux application is executed in its own L4Task, however, with a special restriction that the L4Linux machine where the application belongs to is the registered pager of that task.

The rich OS is simulated by an L4Linux system. In L4Re an IPC mechanism in form of a C++ client server framework exists. This provides a synchronous control channel. The trusted application is implemented as an L4Server while the client part is implemented in the L4Linux kernel. A user level application is implemented on top of the L4Linux kernel to trigger the authentication of the trusted application. Instead of real challenges of a remote server, we also used this trigger application to generate random nonces as server challenges. This approach makes no difference to the timing measurement. The actual plaintext data (the remote server's nonce r_B) is written to a shared memory page by the client. The client, in our case the L4Linux kernel, requests this shared page in advance from the trusted application. The trusted application L4Server registers the page in the microkernel and transfers the capability for the page through the established IPC control channel to the Linux kernel. A detailed view of the software architecture of this attack is provided in Figure 2. As the rich OS is running in user mode, it is necessary to enable the access to the CCNT register beforehand in system mode. We used the boot loader *u*-boot to set this instruction before the hypervisor is executed. However, if the TEE would be realized for example with ARM TrustZone [3], the rich OS is executed in the so called NormalWorld. The SecureWorld of the processor is used for the trusted execution domain. An attacker could then access the CCNT register directly from the rich OS kernel since access rights of the NormalWorld's system mode are sufficient.

5.1 Measurement Setup

The side-channel leakage depend on the used AES implementation. Thus, we analyzed different AES implementations using our authentication protocol shown in Table 1. During the profiling phase, we used the null key for the offline part and for the online part we generated the randomly chosen key k':

$$k' = \texttt{0x 2153 fc73 d4f3 4a98 1733 bb3f 1892 008b}$$

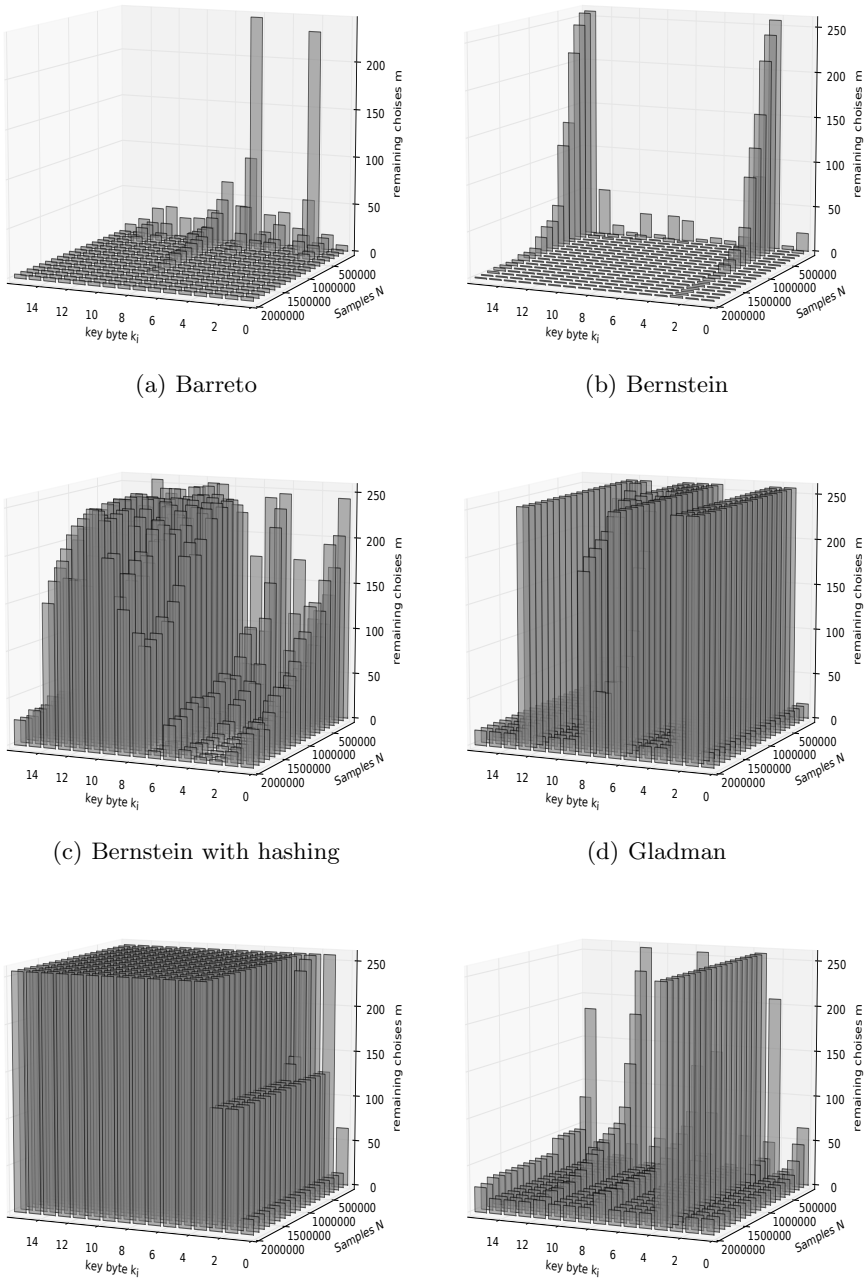
Further, we encrypt the plaintext generated by the trigger application directly and do not perform the hashing operation as described in the protocol. The reason for this is that the hashing generates more noise and makes the comparison between the different AES implementations less clear. Nevertheless, we provide the measurement result with the full protocol implementation exemplary for the AES implementation of Bernstein [6]. However, noise is not really considered in our work but clearly has an impact on the measurements.

We generate a profile every time when additionally 100K samples for each possible plaintext byte value are observed until 2M of each such samples were reached. To generate N samples for each possible value of all plaintext bytes, approximately $N \cdot 256/16$ messages with 16-byte random plaintexts have to be observed.

5.2 Results

We evaluated a broad range of different AES implementations as shown in Table 3. The implementations of Bernstein [6], Barreto [5] and OpenSSL [21] are optimized for 32-bit architectures like the Cortex-A8 whereas Gladman's [12] is optimized for 8-bit micro controllers. Niyaz' [18] implementation is totally unoptimized. Table 3 visualizes the online and offline profile of each implementation. The first column shows the minimum and maximum of the overall timing in CPU cycles which is used for the correlation. The second column shows information about the variation of this timing computed over all measurements. To make propositions over the signal to noise ratio, we also provide the average time spent in the AES encryption method. In Figure 3, the result of the correlation is shown. The plots depict the decreasing possibilities for each key byte by increasing samples. For each implementation, a subfigure is provided which plots the left choices m with $m \in]0; 256]$ in z-direction for each key byte k_i with $i \in [0; 15]$ from left to right, while the amount of samples N for the online profile with $N \in [100K; 2M]$ is plotted in y-direction from behind to front. For this result, a constant sample amount of 2M was used for the offline profile with the null key.

Barreto Barreto's implementation which is part of many crypto libraries is showing a high vulnerability against this time-driven attack. Barreto uses four lookup tables, each of 1 KByte in size. Thus, the lookup tables do not fit into one cache line. Additionally for the last round, a fifth lookup table is used. This type



(a) Barreto

(b) Bernstein

(c) Bernstein with hashing

(d) Gladman

(e) Niyaz

(f) OpenSSL

Fig. 3. Reducing key space by timing attack of different AES implementations

Table 3. Timing profile comparison between the different implementations

Implemenation		time (in cycles)		variation				time aes
		min	max	min	max	median	interval	(in cycles)
Barreto [5]	offline 0	33745.96	33772.29	-9.57	16.77	-0.47	26.34	$\approx$ 4231
	online k'	33745.71	33772.31	-9.87	16.73	-0.49	26.59	$\approx$ 4230
OpenSSL [21]	offline 0	33584.26	33605.61	-8.04	13.31	-0.16	21.35	$\approx$ 4222
	online k'	33585.64	33607.81	-8.99	13.18	-0.14	22.17	$\approx$ 4221
Bernstein [6]	offline 0	33731.61	33778.54	-11.44	35.49	-0.94	46.93	$\approx$ 4546
	online k'	33745.04	33781.29	-5.24	31.00	-0.78	36.24	$\approx$ 4573
Gladman [12]	offline 0	35139.63	35158.00	-6.26	12.10	-0.16	18.37	$\approx$ 5689
	online k'	35139.48	35157.03	-5.72	11.82	-0.16	17.55	$\approx$ 5689
Niyaz [18]	offline 0	59266.99	59280.43	-8.39	5.05	0.03	13.44	$\approx$ 24840
	online k'	59265.01	59278.61	-8.88	4.72	0.01	13.60	$\approx$ 24834

of implementation is also called T-Tables implementation. After 100K samples, only key byte 3 and 7 have more than 200 possibilities left and for key byte 9, the choices are above 50. The other 13 key bytes are all below 50. After 800K almost any key is pinpointed to 4 choices except key byte 9. However, this seems to be the limit for this implementation. That means, using additional samples do not improve the results any further. After 1.6M samples also for key byte 9 the limit is reached and only 4 choices are left. Nothing changes afterwards until 2M samples are reached. See Figure 3(a).

OpenSSL The OpenSSL implementation is almost the same as Baretto's implementation. However, the results of both implementations differ. For the OpenSSL implementation, the limit is reached at 16 choices per key byte. Furthermore, the attack was not able to reduce the key space for key byte 4 at all. One could believe that the results of Barreto's implementation and the results of OpenSSL have to be the same as the encryption function is exactly performing the same operations. However, as listed in Table 3, the overall time which is measured during the attack is about 200 cycles higher for Barreto's implementation because of the encryption function definition. Barreto passes parameters by value which are passed by reference in the OpenSSL encryption function header. Also the performed operations outside the measurement in the trigger application influences the cache evictions. In total, this causes more cache evictions and thus a higher variation of the AES signal, resulting in better correlation behaviour.

Gladman The same holds for the implementation of Gladman which we compiled with tables and 32-bit data types enabled. Here, also the choices for several key bytes are reduced to 16 possibilities. However, Gladman uses only one 256-byte lookup table which means the signal to noise ratio is even worse than in the other implementations. Further, as the cache is 4-way associative with a cache line size of 64 byte, the lookup table fits into one cache block at once. This makes evictions by AES itself nearly impossible. However, other variables

used during the computation can compete with the same lines in cache. This reduces the amount of cache evictions a lot in comparison to the 4 KByte tables implementations. So, there is no reduction of the key space for four key bytes at 2M samples.

Niyaz The implementation of Niyaz seems almost secure against this attack as shown in Figure 3(e). Niyaz also implements the AES with only one S-Box table of 256 byte in size. As in Gladman's implementation, this table also fits in one cache block. Thus, the timing leakage generated by the S-Box lookups is reduced. Further, the unoptimized code beside the table lookups in the encryption method will decrease the signal-to-noise ratio to make it even harder to extract information from the measurements using the correlation.

Bernstein Our results show that Bernstein's AES implementation is most vulnerable to our cache timing attack. However, we used the C compatibility version which is part of his Poly1305-AES [6] message authentication code since no ARM implementation is available. This implementation is the only one which totally leaks the secret key k'. Already after 400K samples, the key is almost completely recovered by the correlation and only 2 key bytes need to be computed using brute-force. Further, during the correlation phase, the possible key bytes are sorted by probability, thus, already after 100K, the correct key k' can be extracted as shown in Table 4. The first column of Table 4 shows the possible choices which are left after correlation. In the second column, the corresponding key byte index is listed while the third column shows the key values sorted by their probability. The values with highest probability are also the correct bytes of k' we introduced in this section. The correct values are printed bold in the table. For this implementation, we also executed the attack with the full mutual authentication protocol, with hashing enabled. We used the reference SHA1 implementation of the L4Re crypto package. In Figure 3(c), it can clearly be seen that the additional noise generated by the hashing function increases the amount of samples needed for the attack.

Table 4. Correlation results after 100K samples of online profile received with the C version of Bernstein's AES implementation; offline profile with 2M samples

choices	byte#	key values ⟵ probability
20	0	**21** 20 23 22 fc 25 26 ..
4	1	**53** 52 51 50
256	2	**fc** cb 9b a1 fd a6 a4 ..
80	3	**73** 70 76 71 75 74 72 ..
10	4	**d4** d6 d5 d7 d3 0a df ..
4	5	**f3** f1 f0 f2
6	6	**4a** 49 4b 48 4f 4d
3	7	**98** 9a 99
23	8	**17** 15 ce c9 13 12 ca ..
27	9	**33** 31 32 ec ea 30 ed ..
4	10	**bb** b8 ba b9
27	11	**3f** 3e 3c 3b 3a e2 e5 ..
4	12	**18** 1b 19 1a
11	13	**92** 90 91 93 97 96 9a ..
51	14	**00** c0 01 02 20 e9 21 ..
256	15	**8b** 06 93 8f 33 b3 0f ..

6 Conclusion

We have shown that the isolation characteristic of virtualization environments can be circumvented using a cache timing attack. This is due to the cache architecture of modern CPUs. Even if authentication schemes with hashing are used, the side-channel leakage of the cache can be used to significantly reduce the key space. Nevertheless, our attack requires many measurement samples and noise also makes our attack more difficult. As there are doubts about practicability of this kind of attacks, further research has to examine proper workloads and real noise. Indeed, cache timing attacks remain a threat and have to be considered during design of virtualization-based security architectures. Switching the algorithm for authentication would not be a solution to this problem. For instance, there exist cache-based timing attacks against asymmetric algorithms like RSA by Percival [17] and ECDSA by Brumley and Hakala [10] as well.

The first step to mitigate those attacks is to not use a T-Tables implementation. However, also the implementations of Gladman and Niyaz with the 256-byte S-Box tables leak timing information which reduces the key space. Since there are many samples needed for the time-driven attack, an attacker may not be able to reconstruct the key within reasonable time. However, there are access-driven attacks which only need several hundreds of samples [14] and even if these attacks are not adaptable to the scenario in this paper yet, it may be possible with further research. An additional option for implementations with a 256-byte S-Box would be to use the preload engine in cooperation with the cache locking mechanism of the Cortex-A8 processor, as the whole S-Box fits in a cache-set. On a higher abstraction layer, the communication stack and all relevant protocol stacks and drivers could be implemented in the trusted domain. However, this would increase the TCB significantly and thus also the probability to be vulnerable to buffer-overflow attacks. Another solution would be to use a crypto co-processor implemented in hardware. This could be either a simple micro controller which does not use caching, or a sophisticated hardware security module (HSM) with a hardened cache-architecture that provides constant encryption timing.

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. ARM Limited. *ARM Security Technology - Building a Secure System using TrustZone Technology*, prd29-genc-009492c edition, April 2009.

4. Samuel A. Bailey, Don Felton, Virginie Galindo, Franz Hauswirth, Janne Hirvimies, Milas Fokle, Fredric Morenius, Christophe Colas, and Jean-Philippe Galvan. The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market. Technical report, GlobalPlatform Inc., 2011.
5. Paulo Barreto, Antoon Bosselaers, and Vincent Rijmen. Optimised ANSI C code for the Rijndael cipher (now AES), 2000. `http://fastcrypto.org/front/misc/rijndael-alg-fst.c`.
6. D. J. Bernstein. Poly1305-AES for generic computers with IEEE floating point, February 2005. `http://cr.yp.to/mac/53.html`.
7. Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.
8. Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. Differential cache-collision timing attacks on aes with applications to embedded cpus. In *The Cryptographer's Track at RSA Conference*, pages 235–251, 2010.
9. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *CHES'06*, pages 201–215, 2006.
10. Billy Brumley and Risto Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 667–684. Springer Berlin / Heidelberg, 2009.
11. Intel Corporation. Intel® virtualization technology list. Website. `http://ark.intel.com/VTList.aspx` accessed 2011 September 15th.
12. Brian Gladman, 2008. `http://gladman.plushost.co.uk/oldsite/AES/aes-byte-29-08-08.zip`.
13. GlobalPlatform Inc. TEE Client API Specification Version 1.0, July 2010.
14. 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.
15. Michael Neve, Jean pierre Seifert, and Zhenghong Wang. Cache time-behavior analysis on aes, 2006.
16. 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.
17. Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.
18. Niyaz PK. Advanced Encryption Standard implementation in C.
19. 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.
20. Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
21. The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS, February 2011. `http://www.openssl.org`.
22. TU Dresden Operating Systems Group. The Fiasco microkernel. Website. `http://os.inf.tu-dresden.de/fiasco/` accessed April 6th 2011.