# Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications

Johann Großschädl[1,2], Elisabeth Oswald[2], Dan Page[2], and Michael Tunstall[2]

[1] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg, Luxembourg
johann.groszschaedl@uni.lu
[2] University of Bristol,
Department of Computer Science,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, U.K.
{johann,eoswald,page,tunstall}@cs.bris.ac.uk

**Abstract.** The design of embedded processors demands a careful trade-off between many conflicting objectives such as performance, silicon area and power consumption. Finding such a trade-off can often ignore the issue of security, which can cause, otherwise secure, software to leak information through so-called micro-architectural side channels. In this paper we show that early-terminating integer multipliers found in many embedded processors (e.g., ARM7TDMI) represent an instance of this problem. The early-termination mechanism causes differences in the time taken to compute a multiplication depending on the magnitude of the operands (e.g., up to three clock cycles on an ARM7TDMI processor), which are observable via variations in execution time and power consumption. Exploiting the early-termination mechanism makes Simple Power Analysis (SPA) attacks relatively straightforward to conduct, and may even allow one to attack implementations with integrated countermeasures that would not leak any information when executed on a processor with a constant-latency multiplier. We describe a number of case studies, including both public-key (RSA, ECIES) and secret-key algorithms (RC6, AES), to demonstrate the threat posed by early-terminating multipliers. Furthermore, we describe an implementation of one such attack on an implementation of AES, where we were able the extract the entire key using just eight power traces.

**Keywords:** Side channel analysis, power analysis, computer arithmetic, general-purpose processor, micro-architectural cryptanalysis.

## 1 Introduction

Within the context of embedded system design, factors such as area and power consumption need to be carefully balanced against performance. This is particularly important when a single component acts as a bottleneck to performance, or is particularly large or power hungry; examples include dedicated multiplier circuits that exist within embedded processors. Approaches to the realisation of such circuits form a large design space. Ignoring issues such as pipelining, at one extreme, are fully parallel designs built entirely from combinatorial logic, for example those based on Wallace [40] or Dadda [14] trees. Such designs represent a low-latency solution since they produce a result in a single clock cycle, but do so at the cost of area and power consumption. At the other extreme are designs which are iterative in the sense that they make iterative use of more modest combinatorial logic (e.g., based on a bit-serial approach). Bit-serial multipliers essentially allow the opposite trade-off by increasing latency but reducing area and power consumption. Of course, intermediate points in the design space exist; these represent approaches that reduce the area of low-latency designs or reduce the latency of low-area or low-power designs. Well-known ways to arrive at such a compromise include the implementation of a digit-serial multiplier [20], the recoding of one of the two operands into a radix-4 representation [11, 26], or a combination of both [15].

Numerous 32-bit processors intended for the embedded market are equipped with digit-serial integer multipliers. For example, ARM7 processors such as the ARM7TDMI [5] contain a $(32 \times 8)$-bit multiplier; other embedded processors feature $(32 \times 12)$-bit multipliers (e.g., Intel StrongARM SA-1100 [22]) or $(32 \times 16)$-bit multipliers (e.g., MIPS32 4Km [28], PowerPC 440 [21], as well as certain ARM9/11 models). Each such processor executes a given multiplication in an iterative fashion by making several passes through the multiplier datapath; each iteration takes an 8- or 16-bit digit of the multiplier-operand, starting with the least-significant digit. The result of the first iteration is fed back into the multiplier and combined with the intermediate products of the following iterations to eventually yield the full result. Generally speaking, a $w$-bit processor comprising of a digit-serial multiplier with a digit-size of $k < w$ bits requires $n = \lceil w/k \rceil$ clock cycles to calculate the $2w$-bit product of a $(w \times w)$-bit multiplication (an extra clock cycle may be necessary if the full product is to be written back to general-purpose registers). One of the reasons why digit-serial multipliers are attractive is the proliferation of Digital Signal Processing (DSP) and multimedia applications in embedded and mobile devices. These application domains are multiplication-intensive, which means that the latency of multiply instructions impacts heavily on overall performance. In order to better support DSP and multimedia kernels, many embedded processors with digit-serial multipliers employ a technique commonly referred to as *early termination*. That is, after each iteration the multiplier checks whether the remaining digits are all zero; if this is the case, the multiplication is terminated "early" and the result is immediately returned. The early-termination mechanism can reduce the latency of multiply instructions if the operands are small, which is often the case in DSP and multimedia applications. For example, a processor with an early-terminating $(32 \times 8)$-bit multiplier can multiply two 8-bit pixel colour values in a single clock cycle, or two 16-bit audio samples in two cycles (instead of four cycles as would be the case without early termination).

Both Kocher et al. [25] and Ravi et al. [35] point out the importance of considering security as an additional dimension in the embedded system design space that demands the same attention as more traditional metrics of interest such as cost, performance, and power. Within this setting, the threat of side channel analysis against embedded systems poses a particularly difficult problem. By passively profiling or actively influencing execution of cryptographic algorithms, it is possible that an attacker can recover otherwise secret information stored in an embedded device. Focusing on power analysis attacks [27], Simple Power Analysis (SPA) refers to a scenario where an attacker typically collects only one, or very few, power traces and attempts to recover secret information by focusing on differences between patterns within each trace. In contrast, Differential Power Analysis (DPA) typically uses several or many traces and analyses differences between the traces [24]. The problem of side channel leakage becomes especially pronounced if a processor itself causes otherwise secure cryptographic software to leak information through so-called *micro-architectural side channels* [4]. Put simply, micro-architectural attacks exploit certain features or effects of standard processor components (e.g., cache sub-systems and branch prediction units) to induce or amplify side channel leakage. In recent years, micro-architectural cryptanalysis based on cache hits/misses [1, 10, 34] as well as branch (mis-)predictions [2, 3] has been studied extensively, and several successful attacks are reported in the literature [7, 33]. These approaches allow an attacker to extract secret keys from cryptographic software, even if it features effective side channel countermeasures that would completely prevent leakage on processors without a cache or branch prediction.

In this paper we show that early-terminating integer multipliers are an example of a micro-architectural side channel and that they can leak significant information about the secret keys used in cryptographic software. As described previously, the early-termination mechanism causes differences in the latency of multiply instructions, which are observable via variations in execution time and power consumption. For example, the latency of a $(32 \times 32)$-bit multiply instruction producing a 64-bit result can vary by up to three clock cycles on a processor with an early-terminating $(32 \times 8)$-bit multiplier (e.g., ARM7TDMI, ARM920T), or up to two cycles if the multiplier has a digit-size of 12 bits (e.g., StrongARM SA-1100), or one cycle in the case of an early-terminating $(32 \times 16)$-bit multiplier (e.g., MIPS32 4Km, some PowerPC cores). Side channel attacks exploiting the early-termination mechanism belong to the category of micro-architectural attacks since the level of susceptibility depends on the micro-architectural design of a processor. However, micro-architectural cryptanalysis based on early-terminating multiplication differs in some aspects from cache attacks and branch-prediction attacks. Firstly, some variants of cache attacks (e.g., those described in [1, 33]), as well as the Simple Branch Prediction Analysis (SBPA) attack [2], rely on other processes (e.g., a "spy" process running in parallel on the same processor) to evict cache lines or to reveal the branch predictor state. An early-termination attack, on the other hand, is completely passive in the sense that it does not require a spy process running on the target processor: instead, our attacks work by feeding cryptographic software carefully chosen input content (i.e., plaintexts) that provoke the early-termination mechanism. Often, very few such inputs, and hence very few executions of the implementation under attack, suffice to extract the entire secret key. A second difference is the type of processors for which said attacks are relevant. While cache memory and branch prediction units can be found in virtually any high-performance processor, they are less common in the embedded market, especially in the low-power segment. Early-terminating integer multipliers, on the other hand, are widely deployed in embedded processors and, as such, early-termination attacks expand the scope of micro-architectural cryptanalysis into the embedded domain.

Our contribution in this paper is threefold. First, we describe in detail how early-terminating multipliers work and what information they leak through power and timing side channels. Even though side channel leakage due to data-dependent instruction timing has been widely investigated within other contexts, the threat posed by the early-termination effect is still unknown or at least undocumented[3]. In particular, we are not aware of any *open* literature describing a concrete attack that exploits early-terminating multiplications[4], while dozens of papers on cache and branch-prediction attacks exist. We consider it of paramount importance to bring the security implications of the early-termination effect to public attention so that engineers become aware of this new micro-architectural side channel and integrate appropriate countermeasures into their products. Our second contribution is to survey vulnerable cryptographic primitives, and to demonstrate how such vulnerabilities can be practically used to mount attacks. We conducted a number of experiments with software implementations of AES, RC6, RSA, and ECC on an ARM7TDMI processor. In all our experiments we succeeded in extracting the entire key with just a few power traces; in some cases a single power trace was sufficient. However, due to space restrictions, we are only able to describe one attack in detail. The third and final contribution is an analysis of potential hardware and software-based countermeasures.

In Section 2 we provide background detail on the early terminating multiplier within ARM7 processors and how it leaks information via power analysis. In Section 3 we investigate theoretical attacks on symmetric and public key primitives including AES, RC6, RSA and ECIES. Since space is at a premium, we describe results from a single set of concrete attacks on AES-128 in Section 4; where our most successful attack was able to retrieve the entire key from eight power consumption traces. Finally in Section 5 we describe and evaluate countermeasures against this form of attack, and conclude in Section 6.

---

[3] There are some lines on the early-termination mechanism in [18, Section 4.2]. However, the authors of [18] consider the early-termination mechanism only for performance analysis of two variants of Montgomery multiplication. Security aspects of the early-termination mechanism (i.e. side channel leakage) are not mentioned.

[4] To the best of our knowledge, none of the current or former processor vendors affected by this attack (e.g., ARM Limited, MIPS Technologies, Intel Corporation, Freescale Semiconductor, etc.) has published a white paper with cautionary notes on the early-termination effect. Of course, this does not necessarily mean that such white papers do not exist; they are just not available to the public.

**Algorithm 1:** A functional description of ARM7TDMI multiplication.

**Input:** The 32-bit integers $x$ and $y$.
**Output:** The 64-bit result $r = x \cdot y$.

$t_0 \leftarrow 0$
**for** $i = 0$ **up to** 3 **step** 1 **do**
  $t_1 \leftarrow x \cdot y_{7\ldots 0}$
  $t_0 \leftarrow t_0 + (t_1 \ll 8i)$
  $y \leftarrow y \gg 8$
  **if** $y = 0$ **then return** $t_0$
**end**

## 2 Background

In this section, we describe the early-termination mechanism in detail, using the ARM7TDMI [5] as a concrete example. We focus on this specific platform because it has a dominant role in the 32-bit embedded processor market. However, we point out that the attacks described in this paper can be mounted on any embedded processor with an early-terminating integer multiplier, including (but not limited to) the StrongARM SA-1100, the MIPS32 4Km, and certain PowerPC models.

We use the following notation: $w$ refers to the processor's word size; in our case $w = 32$ since we are dealing with an ARM7 processor. Let $x_i$ with $0 \le i < w$ denote the $i$-th bit of some $w$-bit word $x$. Furthermore, let $x_{(y)}$ denote $x$ written in base-$y$. For example, $\texttt{F0}_{(16)}$ is the decimal value 240 written in base-16, $X = (X_0, X_1, \ldots X_{n-1})_{(256)}$ is a vector of $n$ elements where each element is written in base-256 (i.e., each element is a byte).

### 2.1 Multiplication on an ARM7TDMI

The ARM instruction set provides several $(32 \times 32)$-bit multiplication instructions which pass two 32-bit inputs $x$ and $y$ to the multiplier circuit. For example, the `umull` and `mul` instructions produce 64-bit and 32-bit unsigned outputs respectively [6]. Signed alternatives are provided that function in a similar manner; however, we exclusively consider unsigned instructions because they are more commonly used in cryptographic algorithms. We term $x$ the multiplicand and $y$ the multiplier. In principle, one can imagine the multiplication hardware operating as described in Figure 1.

The $(32 \times 8)$-bit multiplier processes one 8-bit digit of $y$ in each step, working from the least-significant to the most-significant. After the $i$-th step, if $y = 0$ then the algorithm terminates early and returns the accumulated result in $t_0$. Since $y$ is right-shifted at each step, this essentially means each $j$-th digit of the original $y$, with $j > i$, is checked: if all digits are set to zero then early termination occurs. On the other hand, if $y \neq 0$, then at least one such $j$-th digit is non-zero, and hence a $(32 \times 8)$-bit multiplication is used to form a partial

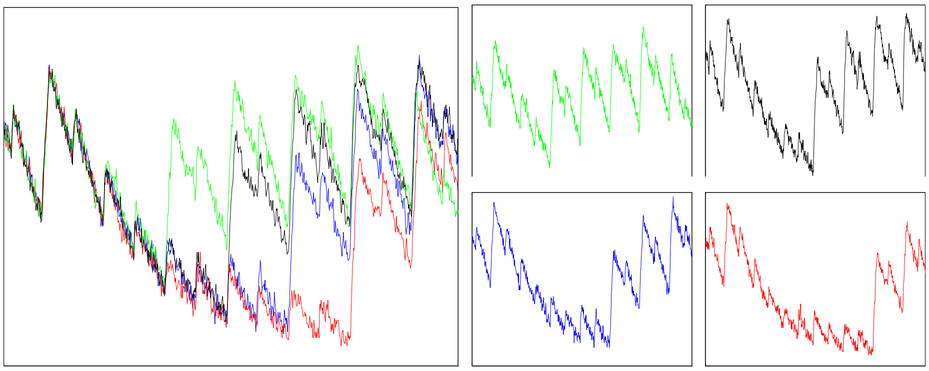

Fig. 1: Overlaid (left) and individual (right) power consumption traces showing ARM7 multiplications that take 2, 3, 4 and 5 clock cycles (top left to bottom right).

product $t_1$, which is scaled and added to $t_0$. The early-termination mechanism means one can consider the algorithm taking between 1 and 4 steps.

In short, this means that a $(32 \times 32)$-bit multiplication is executed in one clock cycle if bits $y_8$ to $y_{31}$ of the multiplier $y$ are all set to zero, in two clock cycles if $y_{16}$ to $y_{31}$) are all set to zero (but $y_8$ to $y_{15}$ are not), in three clock cycles if bits $y_{24}$ to $y_{31}$ are all zero (but $y_8$ to $y_{23}$ are not), and in four cycles otherwise. The so-called "long" multiply instructions that return a 64-bit result (e.g., `umull`) need an additional clock cycle since they have to write-back two 32-bit words into general-purpose registers via a single write port. Putting everything together, the `umull` instruction occupies the execute stage[5] of the pipeline for between two and five clock cycles, depending on the magnitude of the multiplier-operand $y$.

### 2.2 Recovering Multiplication Latency using SPA

"Long" multiply instructions, such as `umull` [6], allow one to specify two source registers (`Rm`, `Rs`) from which the operands to be multiplied are read, and two destination registers (`RdLo`, `RdHi`) into which the lower (resp. upper) part of the 64-bit product is placed. The ARM7TDMI supports early termination on `Rs`, which means that the latency of the `umull` instruction can vary by up to three clock cycles depending on the magnitude of the operand in `Rs` [5]. In what follows, we assume that the multiplicand $x$ is stored in `Rm` and the multiplier $y$ in `Rs`. Consequently, the value of $y$ provokes early termination; where an operand within some pseudo-code algorithm or source code is fed to the multiplier circuit as $y$, we term it an *early-terminating operand*.

[5] The ARM7TDMI processor has a simple three-stage pipeline comprising of Fetch, Decode, and Execute stages [5].

As mentioned above, the `umull` instruction occupies the execute stage of the processor pipeline for between two and five clock cycles, depending on the magnitude of $y$. As a result, the number of clock cycles required to execute the `umull` instruction leaks information about the multiplier $y$. The exact information returned is the number of most-significant 8-bit digits of $y$ that are set to zero, excluding the least-significant digit, which is always processed (see Figure 1). This observation can be made in a "course-grained" way by noting timing differences over an entire execution, or in a "fine-grained" way by observing the power consumption of the processor while it executes a particular `umull` instruction. Figure 1 shows exemplar power consumption traces from an ARM7TDMI clocked at 7.37 MHz to demonstrate this side channel. These traces have been captured using a Tektronix DPO 7104 digital oscilloscope with a differential probe connected to a 1 $\Omega$ shunt in the power supply line.

The `umull` instruction terminates early on the operand read from register `Rs`, i.e. the multiplier $y$ according to our definition from above. On the other hand, the second operand read from register `Rm` has no impact as to whether or not early termination occurs. If one of the two operands to be multiplied is small and known *a priori*, then the programmer can reduce the latency of the `umull` instruction by assigning registers in such a way that this operand provokes the early-termination mechanism. Optimising compilers also try to increase the probability of early termination through appropriate assignment of small operands (e.g. loop counters, array indices). However, there is no guarantee that a compiler performs optimisations so that early termination occurs (or does not occur) on certain input data *or* that an optimisation decision will be safe if the same program is executed on a different, but compatible, processor (e.g., with a different multiplier type). Furthermore, there exist scenarios where a programmer has little or no control over the early-termination mechanism; one may think of Java applets executed in a virtual machine running on a processor. In this case it depends primarily on the virtual machine whether or not the multiplication of a given pair of operands terminates early.

## 3 Theoretical Attacks

In the following we demonstrate that the early-termination mechanism facilitates SPA attacks on both secret-key and public-key cryptosystems.

### 3.1 AES

The structure of the Advanced Encryption Standard (AES) [32], as used to perform encryption, is illustrated in Figure 2. Note that we restrict ourselves to considering AES-128 and that the description omits a permutation typically used to convert the plaintext $P = (P_0, P_1, \ldots, P_{15})_{(256)}$ and key $K = (K_0, K_1, \ldots, K_{15})_{(256)}$ into the matrix form used in the specification [32]. The encryption itself is realised via iterated use of a number of round functions on a state matrix $X$:

**Algorithm 2:** The AES-128 encryption function.

**Input:** The 128-bit plaintext block $P$, and 128-bit key $K$.
**Output:** The 128-bit ciphertext block $C$.

$X \leftarrow$ `AddRoundKey`$(P, K)$
**for** $i \leftarrow 1$ **to** 10 **do**
  $X \leftarrow$ `ShiftRows`$(X)$
  $X \leftarrow$ `SubBytes`$(X)$
  **if** $i \neq 10$ **then**
    $X \leftarrow$ `MixColumns`$(X)$
  **end**
  $K \leftarrow$ `KeySchedule`$(K)$
  $X \leftarrow$ `AddRoundKey`$(X, K)$
**end**
$C \leftarrow X$
**return** $C$

**Algorithm 3:** The AES `MixColumns` function.

**Input:** $X = (X_0, X_1, \ldots, X_{15})_{(256)}$
**Output:** $Y = (Y_0, Y_1, \ldots, Y_{15})_{(256)}$

**for** $i \leftarrow 0$ **to** 15 **do**
  $Y_i = 2 \bullet X_i \oplus 3 \bullet X_{(i+4) \bmod 16} \oplus X_{(i+8) \bmod 16} \oplus X_{(i+12) \bmod 16}$
**end**
**return** $Y$

- The `ShiftRows` function is a byte-wise permutation of the state.
- The `SubBytes` function applies a substitution table (i.e., an S-box) to each byte of the state; formally, this table is an inversion over $\mathbb{F}_{2^8}$ followed by an affine transformation.
- The `KeySchedule` function generates the next round key from the previous one. The first round key is the input key with no changes, subsequent round keys are generated using the `SubBytes` function and XOR operations.
- The `AddRoundKey` function mixes a round key with the state using a XOR operation.
- The `MixColumns` function is shown in Figure 3, where $\bullet$ represents polynomial multiplication over $\mathbb{F}_{2^8}$ modulo the irreducible polynomial $x^8+x^4+x^3+x+1$. That is, polynomial multiplication by 2 and 3 represent multiplication with $x$ and $x+1$ respectively.

An 8-bit implementation typically represents the state as an array of 16 bytes and implements each step of the round function in a direct manner. Within such an implementation, the `xtime` function (a polynomial multiplication by 2) used by `MixColumns` (and within the S-box used in `SubBytes`) is can be implemented as a look-up table, or careful use of data-independent control-flow, to prevent side channel attacks. More specifically, using a look-up table avoids the data-dependent XOR needed to perform reduction by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

**Algorithm 4:** An implementation of the AES `xtime` function on a 32-bit platform.

**Input:** $A = (a_0, a_1, a_2, a_3)_{(256)}$
**Output:** $B = (\texttt{xtime}(a_0), \texttt{xtime}(a_1), \texttt{xtime}(a_2), \texttt{xtime}(a_3))_{(256)}$

1 $R_1 \leftarrow A \wedge \texttt{80808080}_{(16)}$
2 $R_1 \leftarrow R_1 \gg 7$
3 $R_2 \leftarrow R_1 \cdot \texttt{1B}_{(16)}$
4 $R_1 \leftarrow A \ll 1$
5 $R_1 \leftarrow R_1 \wedge \texttt{FEFEFEFE}_{(16)}$
6 $R_1 \leftarrow R_1 \oplus R_2$
7 **return** $R_1$

However, on a 32-bit platform it can be attractive to compute (rather than look-up) results of `xtime` as described in Figure 4. Intuitively, this approach, due to Bertoni *et al.* [9], appears more time consuming than a single look-up; however, it allows four applications of `xtime` to be computed in parallel. Compared to the traditional T-tables approach to implementing AES on 32-bit platforms, this realisation of `xtime` allows a trade-off toward performance over memory footprint (which is crucial for embedded applications) and guards against cache-based side channel attacks.

Assuming that $R_1$ is the early-terminating operand[6], the danger of this approach is clear: if an attacker can recover how many clock cycles it takes to compute the multiplication in step 3, they can determine how many of the most-significant bytes of $R_1$ are set to zero.

### 3.2 RC6

Consider encryption using the block cipher RC6 [36] described in Figure 5 for the specific case of $w = 32$; this description assumes that the round keys represented by $S$ are an auxiliary input. Focusing on the plaintext input $B$, the pertinent feature with regard to early termination occurs in step 4 which, crucially, is after the initial whitening step. In the first round (i.e., when $i = 0$), this step represents a $(32 \times 32)$-bit unsigned multiplication where both operands are derived from the input $B$ and $S[0]$.

[6] An optimising C compiler would rather use the constant $\texttt{1B}_{(16)}$ as early-terminating operand (even though there is no guarantee for this). However, the attack on `xtime` is nonetheless practically relevant if we consider a Java implementation of Figure 4. In this case the programmer has no control over the early-termination mechanism as it depends primarily on the virtual machine whether $R_1$ or $\texttt{1B}_{(16)}$ is the early-terminating operand.

**Algorithm 5:** The RC6 encryption function.

**Input:** A 4-tuple of 32-bit plaintext values $(A, B, C, D)$.
**Output:** A 4-tuple of 32-bit ciphertext values $(A, B, C, D)$.

1 $B \leftarrow B + S[0]$
2 $D \leftarrow D + S[1]$
3 **for** $i = 0$ **up to** $r$ **do**
4 $\quad t \leftarrow (B \cdot (2B + 1)) \lll 5$
5 $\quad u \leftarrow (D \cdot (2D + 1)) \lll 5$
6 $\quad A \leftarrow ((A \oplus t) \lll u) + S[2i]$
7 $\quad C \leftarrow ((C \oplus u) \lll t) + S[2i + 1]$
8 $\quad (A, B, C, D) \leftarrow (B, C, D, A)$
9 **end**
10 $A \leftarrow A + S[2r + 2]$
11 $C \leftarrow C + S[2r + 3]$
12 **return** $(A, B, C, D)$

Assume the early-terminating operand for this multiplication is $B + S[0]$. An attacker can recover the number of cycles taken by this multiplication. Using adaptive choices of $B$, the attacker can perform trial encryptions until the whitening step computes a result $B + S[0]$ in which the most-significant byte is zero. Since this intermediate value is used as the early-terminating operand in step 4, the fact that the most significant byte is zero causes early termination.

In an ideal setting, this would mean one could search for a $B$ such that $B + S[0] = 0$, i.e., $S[0] = -B$, and hence recover $S[0]$. However, as detailed in Sections 2.1 and 2.2, one cannot recover information about the least-significant byte of the early-terminating operand. This means that instead of recovering $S[0]$ directly, one narrows the possible range of values: one finds that $B + S[0] \in \{0 \ldots 255\}$. Even so, one can view this as leaking 24 bits of the 32-bit round key $S[0]$; the same approach yields $S[1]$ via observation of the second multiplication involving $D$ (within the same set of acquisitions).

### 3.3 Exponentiation in $\mathbb{Z}_N$ (e.g., RSA)

A central operation in RSA [37] is exponentiation in $\mathbb{Z}_N$ where $N = p \cdot q$ for secret, large primes $p$ and $q$. In textbook RSA, this operation takes plaintext (resp. ciphertext) $x$ and exponentiates it by a key $y$ to compute ciphertext (resp. plaintext) $r = x^y \bmod N$. Let $y_i$ denote the $i$-th bit in the binary expansion of $y$, and $S$ and $M$ represent modular squaring operations and multiplications in $\mathbb{Z}_N$.

Typically, $x$ is controllable by an attacker while $y$ is fixed: either it represents the public or private RSA key. The classic square-and-multiply method (i.e., left-to-right binary exponentiation) described in Figure 6 provides a simple method to compute $r$. Within the $i$-th iteration of the algorithm a multiplication is executed if, and only if, $y_i = 1$. This leaks the value of $y$ if an attacker can distinguish squaring operations (i.e., step 3) from multiplications (i.e., step 5):

**Algorithm 6:** The left-to-right binary exponentiation algorithm.

**Input:** The integers $x$, $y$ and $N$.
**Output:** The integer $r = x^y \pmod N$.

```
1 t ← x
2 for i = |y| − 2 downto 0 do
3     t ← t^2 (mod N)
4     if y_i = 1 then
5         t ← t · x (mod N)
6     end
7 end
8 return t
```

if the attacker observes the sequence $SM$ during iteration $i$ (i.e., a squaring operation then a multiplication is executed) then $y_i = 1$ whereas if they observe $S$ alone then $y_i = 0$.

Since the values in the algorithm described in Figure 6 are multi-precision, e.g. 1024-bits held in 32-bit words, a method such as Montgomery multiplication [30] is typically used to perform modular multiplication. Step 3 uses $t$ as both operands to the modular multiplication; since $t$ is essentially random as the algorithm progresses, both operands are random. However, the multiplication in step 5 uses $x$ as one operand. If the digits of $x$ form early-terminating operands to $(32 \times 32)$-bit multiplications within the larger modular multiplication, the early-termination mechanism can be invoked. Specifically, if an attacker controls $x$, they can select a value that is "special" in the sense it is low-weight (i.e., a number of 32-bit digits are zero). Such an $x$ permits the attacker to distinguish between modular squaring operations and multiplications based on how often the early-termination mechanism is invoked; following the reasoning above this leaks $y$ which, depending on the context, is potentially the private key.

Interestingly, this approach not only works for "textbook RSA" as outlined above, but also when the plaintext (resp. ciphertext) is padded according to PKCS #1 [38]. The PKCS #1 standard provides recommendations for the implementation of public-key cryptography based on the RSA algorithm, covering both encryption and signature generation/verification. Version 2.1 of PKCS #1 specifies two padding schemes for encryption, namely the Optimal Asymmetric Encryption Padding (OAEP) and an older padding scheme from PKCS #1 version 1.5 which is not recommended for new applications. In both cases the message is first encoded before the encryption (i.e., modular exponentiation) is performed. Conversely, the decryption of the ciphertext starts with a modular exponentiation to recover the plaintext, which is then decoded into the original message. Assume an attacker wishes to extract the secret key used in an RSA decryption operation by exploiting the early-termination mechanism: to achieve this, they manipulate the ciphertext such that it has a low-weight (e.g., by injecting a number of 32-bit words that are set to zero). As mentioned previously, the decryption process starts with a modular exponentiation of the ciphertext using the secret key as exponent, followed by the decoding of the obtained plaintext to retrieve the original message. Consequently, a manipulation of the ciphertext can only be detected *after* the exponentiation has finished. Any multiply instruction using one of the low-weight words as operand will terminate early, thereby enabling an attacker to distinguish modular multiplications from modular squaring operation when the exponentiation is performed as shown in Figure 6. Extracting the secret key from a PKCS #1-compliant implementation of RSA decryption is essentially no harder than attacking textbook RSA, provided that the attacker has the possibility to manipulate a ciphertext or to inject chosen ciphertexts.

Contrary to RSA decryption, attacking a PKCS #1-compliant implementation of RSA signature generation is not so straightforward. The PKCS #1 standard specifies signature schemes with appendix, which means that the message to be signed is first hashed to produce an intermediary representation, which is then used as operand for the modular exponentiation. Therefore, mounting the attack requires producing a message whose intermediary representation has low-weight, which can be accomplished by brute force message search.

**Attacking SPA-Resistant $m$-ary Exponentiation** The early-termination effect amplifies side-channel leakage, but an SPA attack is, of course, also possible without exploiting this effect. In order to thwart SPA attacks on RSA, a number of "regular" exponentiation techniques have been proposed; these range from always computing a, possibly dummy, a multiplication to $m$-ary exponentiation with a recoded exponent [23]. The aim of these methods is to perform the exponentiation in such a way that always the same sequence of operations (i.e. modular multiplications and squaring operations) is executed, irrespective of the exponent. The $m$-ary method is based on $m$-ary expansion of the exponent $y$, whereby $m$ is typically a power of 2, i.e. $m = 2^k$. It uses a table of $m - 2$ pre-computed powers of the base $x$, i.e. $x^i$ for $i \in \{1, \ldots, m-1\}$, and processes an $k$-bit digit of the exponent $y$ at a time, which reduces the number of modular multiplications compared to the binary method. Möller [29] proposed a recoding scheme for $m$-ary exponentiation where each $k$-bit digit that is equal to zero is replaced with $-m$, and the next most significant digit is incremented by one. This leads to an exponent recoded with digits in the set $\{1, 2, \ldots, m-1\} \cup \{-m\}$. A good overview of other exponent recoding schemes yielding regular $m$-ary exponentiation can be found in [23].

Unfortunately, these regular $m$-ary exponentiation techniques succumb to a SPA attack when exploiting the early-termination effect. The attacker just needs to select the base $x$ in such a way that exactly one of the pre-computed powers of $x$ contains a byte equal to zero at the "right" positions. Whenever a multiplication with this power of $x$ is executed, the early-termination mechanism is invoked, which leaks the value of the corresponding $k$-bit digit from the secret exponent. Repeating this attack with other values of $x$ such that a different power of $x$ contains byte equal to zero will eventually allow the attacker to fully recover the private exponent. The early-termination effect makes attacking these regular $m$-ary exponentiation methods—which are designed to be SPA-resistant—almost as easy as attacking a completely unprotected implementation, such as the one shown in Algorithm 6.

### 3.4 Point multiplication on $E(\mathbb{F}_p)$ (e.g., ECIES)

Consider an elliptic curve $E(\mathbb{F}_p)$. A central operation to cryptographic schemes based on such a curve is scalar multiplication of some point $\boldsymbol{P} \in E$ by a secret integer $d$, i.e., $\boldsymbol{Q} = d \cdot \boldsymbol{P}$. Let $d_i$ denote the $i$-th bit in the binary expansion of $d$, and $A$ and $D$ represent point addition and doubling operations on $E$. Depending on the exact setting $\boldsymbol{P}$ might be fixed or unknown; consider instead a setting where $\boldsymbol{P}$ is supplied as input and hence is controllable by an attacker.

The double-and-add algorithm [19, page 97] provides a simple method to compute $\boldsymbol{Q}$; since this is the additive analogue to the multiplicative algorithm, it is vulnerable to similar side channel attacks [13]. One way to harden the algorithm is to split the point addition operation into parts each of which is identical, in terms of the field operations it performs, to a point doubling operation. Put simply, instead of a sequence such as $DA$ the attacker now observes the sequence $XXX$ where each $X$ represents an atomic, indistinguishable operation which could be a point doubling operation or a step in a point addition.

Point doubling and addition sequences specified by Gebotys and Gebotys [16] was the first example of this; multiplication by small constants is implemented using shifts. Several "dummy" operations, which source or target the $\perp$ value, are included to pad the sequences so the same operation occurs at each index. Gebotys and Gebotys [16, pp. 117–118] are careful to note that on their experimental platform (a StarCore SC140 VLIW-based DSP)

> "the only field operations which had variable clock cycle counts were the modular reductions which may or may not be required after additions, subtractions, or shifts".

Of course, where an early-terminating multiplier is used to perform $(32 \times 32)$-bit multiplications within the field squaring operations and multiplications, this ceases to be true. In order to overcome this countermeasure, an attacker can select a $\boldsymbol{P}$ whose $x$ or $y$ coordinates are "special" in the sense they are low-weight (i.e. a number of 32-bit digits are set to zero). In this setting, *even though* a high-level SPA countermeasure is implemented, an attacker can still distinguish between a point doubling operation and a point addition by observing when this low-weight coordinate is used and hence the early-termination mechanism is invoked more often than usual. This can be viewed as related to the attack of Goubin [17] where an attacker attempts to provoke computation using "special" points (e.g., one where the $x$ or $y$ coordinate is set to zero).

The elliptic curve (EC) point multiplication operation is a central operation of all EC based cryptosystems: given an EC point $\boldsymbol{P}$ and a scalar value $k$, the operation $\boldsymbol{Q} = [k]\boldsymbol{P}$ outputs another point $\boldsymbol{Q}$ on the curve. There are various implementation strategies for this including the simple binary algorithm (aka double-and-add algorithm, see [13]). In this algorithm, a sequence of EC point addition operations and EC point double operations. More precisely, in every iteration a point doubling operation is performed. However, if, and only if, the $i$-th bit of $k$ equals one, a point addition operation is performed as well.

It has been observed before [13] that naïve implementations of EC point multiplications are vulnerable to SPA attacks. Consequently, SPA resistant implementations were proposed using, for instance, indistinguishable operations [12], see [16] for a concrete implementation. The goal of indistinguishable operations is to make EC point addition and EC point doubling operation "look alike" in terms of their power profiles.

Using the early termination feature of a multiplier we can even break implementations using such indistinguishable formulas. In the $i$-th step of the binary algorithm, the point $\boldsymbol{P}$ is added if (and only if) $k_i = 1$. Now, if we set $\boldsymbol{P}$ to be a "special" point, i.e., a point which has either one or both coordinates with leading bytes of zeroes, then the early termination will always occur when $\boldsymbol{P}$ is added to the current intermediate point. Assuming that the intermediate points $\boldsymbol{Q}$ that occur during the computation of the binary algorithm have random coordinates, i.e., do not lead to the early termination effect in the same way $\boldsymbol{P}$ does, the point addition operation is identifiable because of the early termination effect. Identifying the point addition operation allows identifying the bits of $k$ which are equal to one.

The remaining problem is to identify ECC based cryptosystems in which an attacker can control (i.e., choose) the base point $\boldsymbol{P}$. Typically, such cryptosystems supply a base point $\boldsymbol{G}$ as part of their domain parameters; since this point is fixed it cannot be chosen by an attacker. This rules out schemes such as ECDSA [31] where point multiplication within the signature generation function uses $\boldsymbol{G}$ as the base point. However, KEM-DEM based encryption schemes such as ECIES [39] use a Diffie-Hellman (DH) style key exchange mechanism within their KEM component. In the decryption step of such a KEM, the private key is used to multiply a point derived from the ciphertext (i.e., can be chosen by an attacker). Consequently, this point multiplication leaks the private key if the implementation makes use of an early-terminating multiplier and the attacker chooses the base point appropriately. Using the same observation, other DH based protocols including ephemeral and static versions of ECDH, as well as ECMQV, are vulnerable.

## 4 Concrete Attacks on AES

In this section we present some concrete attacks applied to AES as implemented on an ARM7TDMI processor [5]. In each case, we assume that an attacker is able to control the plaintexts being encrypted by the AES implementation, and is able to observe a suitable side channel, such as the power consumption. For clarity, two attacks are presented, where the first attack only observes the first round of the AES implementation, and the second attack observes the first two rounds.

### 4.1 Theory

Before discussing how the attacks were implemented we will describe how our attacks work in theory. In both cases we make observations via Simple Power Analysis (SPA) and exclude key hypotheses based on our observations. In the first attack this leads to an exhaustive key search, and in the second attack allows the key to be determined with no exhaustive key search.

**Attacking the First Round** As described in Section 3.1, when the `xtime` function is computed insecurely on a 32-bit platform an attacker can observe how many of the most significant bytes of $R_1$ are set to zero and the most significant byte of $R_1$ that is set to one. If we observe the multiplication that occurs in step 3 of the algorithm described in Figure 4, an attacker can derive bits of the result of $\texttt{SubBytes}(X_i \oplus K_i)$, for some $i \in \{0, 1, \ldots, 15\}$, by counting how many clock cycles this multiplication takes.

Each of the observations will provide information on a certain a number of bits from different bytes output from the `SubBytes` function. Given that these bits are from different bytes, an attacker will require at least eight observations for each byte to derive the value of the associated secret key byte. However, in order to determine the second least significant byte, an attacker will require at least eight observations where the two most significant bytes are set to zero. If we assume that the plaintext being encrypted is random this will occur with a probability of 1/4. An attacker would, therefore, expect to require 32 observations to derive the three most significant bytes and reduce an the number of possible key hypotheses from $2^{128}$ to $2^{32}$. As described in Section 2.2, no information can be derived on the least significant byte. While we can note that eight calls to the `xtime` function are required to compute the `MixColumns` algorithm (see Figure 3) on a 32-bit platform, there are only four unique calls.

**Attacking the Second Round** If an attacker acquires the power consumption during the computation of the first two rounds of AES more information is available. The same information is available to an attacker as given by the `MixColumns` function in the first round. However, an attacker can also determine information on the key bytes that cannot be determined by observing the first round. In this section we describe how an attacker could exploit the information by observing early terminations in the first two calls to the `MixColumns` function.

We denote $Y_i$ as the $i$-th byte of the result of the second computation of the `SubBytes` function for $i \in \{0, 1, \ldots, 15\}$. If, for example, an attacker derives a bit in the second round, they can then compute which combinations of secret key bytes will produce the observed bit given the known plaintext. Imagine, for example, the most significant bit of the first byte of the result of the `SubBytes` function is observed. In this case, we can note that the most significant bit of $Y_0$

is known, where $Y_0$ is defined as

$$Y_0 = S(\gamma(S(X_0 \oplus K_0)) \oplus \gamma(S(X_5 \oplus K_5)) \oplus S(X_5 \oplus K_5) \\ \oplus S(X_{10} \oplus K_{10}) \oplus S(X_{15} \oplus K_{15}) \oplus S(K_{13}) \oplus K_0 \oplus 1),$$

$S$ denotes the `SubBytes` function, and $\gamma$ is the `xtime` function (the formulae for producing $Y_i$ for $i \in \{0, 1, \ldots, 15\}$ are given in Appendix A). The combinations of $K_0$, $K_5$, $K_{10}$, $K_{15}$, $K_{13}$ that will produce the known bit can be noted. This can be repeated for every acquisition, and the intersection of the combinations of the key bytes can be taken to reduce the number of possible hypotheses for these key bytes.

This process can be repeated for other bits observed in the second round to produce another set of hypotheses for a different set of key bytes. When another set of possible key bytes is created, the intersection between the new set and previously generated sets can be analysed to reduce the number of key hypotheses. That is, if a combination of bytes does not exist in one set of hypotheses it can be removed from all the other sets.

This information can be combined with the information already derived from analysing the first round, by noting when secret key bytes present in the list of hypotheses generated from the first analysis are no longer possible. This is convenient for a subsequent exhaustive search as an algorithm for working all the possible keys is trivial. It is slightly inefficient since some combinations of secret key bytes will not be possible. However, in our simulations of this attack, combining the information for all the outputs of the second `SubBytes` function was always sufficient to determine the secret key without an exhaustive search.

**Attacking Later Rounds** There is no interest in including information from later rounds. This is because the information from the second round is typically enough to determine the secret key. If the number of acquisitions is not sufficient for the secret key to be determined then there are problems with storing the total possible hypotheses in a form that can be accessed in a reasonable amount of time.

### 4.2 Practice

In this section we describe a practical implementation of the attacks detailed above. The acquisitions referred to in this section were acquired without any specific filtering or averaging.

**Attacking the First Round** A series of 48 acquisitions of the power consumption were taken during execution of the first round of an implementation of AES. The traces were synchronised after the first multiplication in the `MixColumns` function. The difference that each trace needed to be shifted was noted to give the number of clock cycles that it took to compute each multiplication. This was then repeated for each subsequent computation in the first round. This information was then used to reduce the number of possible keys, as described in

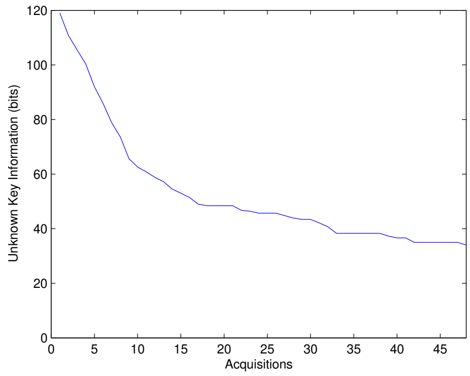


Fig. 2: The size of an exhaustive key search in bits, after analysing $x$ acquisitions.

Section 4.1. The number of keys that would need to be tested after the inclusion of the information corresponding to each acquisition is shown in Figure 2. The number of key hypotheses is rapidly reduced to approximately $2^{34}$, which is significantly more than the theoretical result given above. No information was available on $K_0$, $K_5$, $K_{10}$, $K_{15}$. This is because the ARM7 structure is little-endian, so the bytes are not loaded into $R_1$ in the intuitive order. It took two hours on a standard PC to find the correct key by testing $17.07 \cdot 10^9$ key hypotheses.

**Attacking the Second Round** Sixteen acquisitions of the power consumption were taken during execution of the first two rounds of an implementation of AES. As previously, the traces were synchronised after each multiplication in the `MixColumns` function.

The attack described in Section 4.1 was applied to the resulting multiplication lengths, after having first reduced the number of possible key hypotheses by analysing the information available from the first round. When using the information from ten to 16 acquired traces the secret key was found instantly. When the information from nine traces was analysed the secret key was found after 15 minutes, and after two hours when the information from eight traces were used.

No exhaustive search was necessary as the formulae for the output of the second `SubBytes` function reduce the number of hypotheses to one. However, it can take some considerable time to eliminate all the false hypotheses.

In order to minimise the amount of memory required to store the possible key combinations in memory the order in which the formulae for $y_i$ are treated can be ordered such that information on secret key bytes that had few hypotheses was derived first. In the described attack the bytes were evaluated in the following order: $Y_{15}$, $Y_{14}$, $Y_2$, $Y_3$, $Y_1$, $Y_{11}$, $Y_5$, $Y_6$, $Y_9$, $Y_{10}$, $Y_7$, and $Y_{13}$.

**Algorithm 7:** A constant time algorithm to replace ARM7 multiplication.

**Input:** The 32-bit integers $x$ and $y$.
**Output:** The 64-bit result $r = x \cdot y$.

1 $\gamma \leftarrow (y \wedge \texttt{00FFFFFF}_{(16)}) + \texttt{01000000}_{(16)}$
2 $\tau \leftarrow (y \wedge \texttt{FF000000}_{(16)}) \gg 24$
3 $r \leftarrow x \cdot \gamma$
4 $r \leftarrow r + ((x \cdot \tau) \ll 24)$
5 $r \leftarrow r - (x \ll 24)$
6 **return** $r$

## 5 Countermeasures

Given the successful attack described in Section 4, it is clearly attractive to examine potential countermeasures. We do so by focusing on step 3 of the vulnerable `xtime` implementation described in Figure 4, but note that the techniques are more generally applicable.

The most invasive countermeasure would be to alter the processor itself. For example, one can easily imagine including a dedicated instruction (or a processor mode) that disables the early-termination mechanism during execution of security-critical regions of a program. However, this is disadvantageous in the sense that such an approach is potentially costly and cannot be retrospectively applied to existing processors. As such, one can also consider software-only countermeasures. The simplest approach of this type is to ensure that secret information is never used as an early-terminating operand. In the context of AES-128, this means placing the constant value $\texttt{1B}_{(16)}$ in the register that governs how long a multiplication takes, i.e., `Rs` rather than $R_1$.

Where this is not possible (e.g., both operands are secret, or the programmer does not directly control instructions being executed as could be the case in interpreter-based platforms such as Java), one can imagine replacing each use of an insecure multiplication instruction with a more heavy-weight algorithm. The algorithm described in Figure 7 represents an example: it forces the number of cycles required to perform a multiplication to be a constant, i.e., to be data-independent.

Essentially this works by masking the multiplier $y$ so that the most-significant byte in one multiplication is always non-zero, and the other is always a multiplication with one byte. The two invocations of the real multiplication in steps 3 and 4 therefore leak no information. Use of the algorithm within our implementation of AES-128 increases the execution time from 1.24 milliseconds to 1.56 milliseconds. However, this is still a significant improvement over implementing the `MixColumns` in a byte-wise manner; our implementation requires 2.53 milliseconds in this case. The impact on an implementation of a modular exponentiation is larger; for example, the execution time of a 1024-bit exponentiation on the ARM7TDMI was increased from 1.6 to 3.135 seconds (at 7.37 MHz).

For symmetric cryptosystems, the best "countermeasure" against side channel attacks exploiting the early-termination mechanism is to avoid integer multiplications when designing algorithms, as recommended by Bernstein [8].

## 6 Conclusions

In this paper we describe and analyse some security issues that arise when cryptographic software is executed on an embedded processor with an early-terminating integer multiplier. Even though the early-termination mechanism provides clear advantages for certain applications (most notably digital signal and multimedia processing), it poses a serious challenge for security-critical applications that need to withstand side channel attacks. We have explained why, and demonstrated how, the early-termination mechanism causes differences in the latency of multiply instructions; in turn, this results in easily observable variations in execution time and power consumption. Such data-dependent variations make power analysis fairly straightforward, and may even allow an attacker to extract the secret key from implementations with integrated high-level countermeasures. While the side channel leakage caused by the early-termination mechanism is obvious for public-key cryptosystems performing multi-precision multiplications (e.g., RSA and ECIES), we have also demonstrated that block ciphers, such as AES, are vulnerable to SPA attacks when executed on an ARM7TDMI processor. Consequently, careful attention must be focused on the implementation of cryptographic software at a low-level so that the early-termination mechanism does not produce side channel leakage; this can be costly and difficult to achieve via software-only countermeasures. Another conclusion that can be drawn from the discovery of security issues caused by early-terminating multipliers (and other micro-architectural side channels) is that processor vendors need to reassess their goals in micro-architectural design of embedded processors: security aspects require and deserve the same attention as other metrics of interest such as performance, silicon area, and power consumption.

## Acknowledgements

The authors are grateful to Çetin Koç, Marcel Medwed, and Stefan Tillich for valuable comments and suggestions which helped to improve the quality of this paper.

The work described in this paper has been supported by the EPSRC under grants EP/E001556/1 and EP/F039638/1, and, in part, by the European Commission through the ICT Programme under contract ICT-2007-216676 ECRYPT II. The information in this paper reflects only the authors' views, is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

## References


1. Onur Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the 1st ACM Workshop on Computer Security Architecture (CSAW 2007)*, pages 11–18. ACM Press, 2007.
2. Onur Acıiçmez, Çetin K. Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS 2007)*, pages 312–320. ACM Press, 2007.
3. Onur Acıiçmez, Çetin K. Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Masayuki Abe, editor, *Topics in Cryptology — CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer Verlag, 2007.
4. Onur Acıiçmez, Jean-Pierre Seifert, and Çetin K. Koç. Micro-architectural cryptanalysis. *IEEE Security & Privacy*, 5(4):62–64, July/August 2007.
5. ARM Limited. ARM7TDMI Technical Reference Manual (Revision r4p1). ARM Doc No. DDI 0210, Issue C, available for download at `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf`, November 2004.
6. ARM Limited. ARM Architecture Reference Manual. ARM Doc No. DDI 0100, Issue I, available for download at `http://www.arm.com/miscPDFs/14128.pdf`, July 2005.
7. Daniel J. Bernstein. Cache-timing attacks on AES. Preprint, available for download at `http://cr.yp.to/papers.html#cachetiming`, 2005.
8. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer Verlag, 2008.
9. Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient software implementation of AES on 32-bit platforms. In Burton S. Kaliski Jr., Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 159–171. Springer Verlag, 2003.
10. Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *Proceedings of the 6th International Conference on Information Technology: Coding and Computing (ITCC 2005)*, volume 1, pages 586–591. IEEE Computer Society Press, 2005.
11. Andrew D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, June 1951.
12. Eric Brier and Marc Joye. Weierstraß elliptic curves and side-channel attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography — PKC 2002*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer Verlag, 2002.
13. Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES '99*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer Verlag, 1999.
14. Luigi Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34(5):349–356, May 1965.
15. Stephen B. Furber. *ARM System-on-Chip Architecture*. Addison-Wesley, second edition, 2000.

16. Catherine H. Gebotys and Robert J. Gebotys. Secure elliptic curve implementations: An analysis of resistance to power-attacks in a DSP processor. In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 114–128. Springer Verlag, 2002.
17. Louis Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In Yvo Desmedt, editor, *Public Key Cryptography — PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 199–210. Springer Verlag, 2003.
18. Gaël Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 293–301. Springer Verlag, 2000.
19. Darrel R. Hankerson, Alfred J. Menezes, and Scott A. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer Verlag, 2004.
20. Richard Hartley and Peter Corbett. Digit-serial processing techniques. *IEEE Transactions on Circuits and Systems*, 37(6):707–719, June 1990.
21. IBM Corporation. PowerPC 440x6 Embedded Processor Core User's Manual (Version 07). Available for download at `http://www.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core`, July 2008.
22. Intel Corporation. Intel® StrongARM® SA-1100 Microprocessor for Embedded Applications. Brief datasheet, order number 278092-005, June 1999.
23. Marc Joye and Michael Tunstall. Exponent recoding and regular exponentiation algorithms. In Bart Preneel, editor, *Progress in Cryptology — AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 334–349. Springer Verlag, 2009.
24. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer Verlag, 1999.
25. Paul C. Kocher, Ruby B. Lee, Gary E. McGraw, Anand Raghunathan, and Srivaths Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st Design Automation Conference (DAC 2004)*, pages 753–760. ACM Press, June 2004.
26. Olin L. MacSorley. High-speed arithmetic in binary computers. *Proceedings of the IRE*, 49(1):67–91, January 1961.
27. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Springer Verlag, 2007.
28. MIPS Technologies, Inc. MIPS32 4Km™ Processor Core Datasheet. Available for download at `http://www.mips.com/products/processors/32-64-bit-cores/mips32-m4k/`, November 2004.
29. Bodo Möller. Securing elliptic curve point multiplication against side-channel attacks. In George I. Davida and Yair Frankel, editors, *Information Security — ISC 2001*, volume 2200 of *Lecture Notes in Computer Science*, pages 324–334. Springer Verlag, 2001.
30. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
31. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). FIPS Publication 186-2, available for download at `http://www.itl.nist.gov/fipspubs/`, February 2000.
32. National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). FIPS Publication 197, available for download at `http://www.itl.nist.gov/fipspubs/`, November 2001.

33. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology — CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer Verlag, 2006.
34. Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, Bristol, U.K., June 2002.
35. Srivaths Ravi, Anand Raghunathan, Paul C. Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems*, 3(3):461–491, August 2004.
36. Ronald L. Rivest, Matthew J. Robshaw, Ray Sidney, and Yiqun L. Yin. The RC6™ block cipher. Technical report, RSA Laboratories, Bedford, MA, USA, August 1998. Available for download at `ftp://ftp.rsasecurity.com/pub/rsalabs/rc6/rc6v11.pdf`.
37. Ronald L. Rivest, Adi Shamir, and Loenard M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
38. RSA Security, Inc. PKCS #1 v2.1: RSA Cryptography Standard. Available for download at `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf`, June 2002.
39. Standards for Efficient Cryptography Group (SECG). SEC 1: Elliptic Curve Cryptography. Available for download at `http://www.secg.org/download/aid-385/sec1_final.pdf`, September 2000.
40. Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, 13(1):14–17, February 1964.

# A AES Early Termination Formulae

The formulae referred to in Section 4 are given below. The result bytes $Y_i$ is created from plaintext bytes $X_i$ and secret key bytes $K_i$ for $i \in \{0, 1, \ldots, 15\}$. For brevity the `SubBytes` function is written as $S$, and the `xtime` function as $\gamma$.

$$
\begin{aligned}
Y_0 = S(&\gamma(S(X_0 \oplus K_0)) \oplus \gamma(S(X_5 \oplus K_5)) \oplus S(X_5 \oplus K_5) \\
&\oplus S(X_{10} \oplus K_{10}) \oplus S(X_{15} \oplus K_{15}) \oplus S(K_{13}) \oplus K_0 \oplus 1) \\
Y_1 = S(&\gamma(S(X_4 \oplus K_4)) \oplus \gamma(S(X_9 \oplus K_9)) \oplus S(X_9 \oplus K_9) \\
&\oplus S(X_{14} \oplus K_{14}) \oplus S(X_3 \oplus K_3) \oplus S(K_{13}) \oplus K_0 \oplus K_4 \oplus 1) \\
Y_2 = S(&\gamma(S(X_8 \oplus K_8)) \oplus \gamma(S(X_{13} \oplus K_{13})) \oplus S(X_{13} \oplus K_{13}) \\
&\oplus S(X_2 \oplus K_2) \oplus S(X_7 \oplus K_7) \oplus S(K_{13}) \oplus K_0 \oplus K_4 \oplus K_8 \oplus 1) \\
Y_3 = S(&\gamma(S(X_{12} \oplus K_{12})) \oplus \gamma(S(X_1 \oplus K_1)) \oplus S(X_1 \oplus K_1) \oplus S(X_6 \oplus K_6) \\
&\oplus S(X_{11} \oplus K_{11}) \oplus S(K_{13}) \oplus K_0 \oplus K_4 \oplus K_8 \oplus K_{12} \oplus 1)
\end{aligned}
$$

$$
\begin{aligned}
Y_4 &= S(\gamma(S(X_9 \oplus K_9)) \oplus \gamma(S(X_{14} \oplus K_{14})) \oplus S(X_{14} \oplus K_{14}) \\
&\quad \oplus S(X_3 \oplus K_3) \oplus S(X_4 \oplus K_4) \oplus S(K_{14}) \oplus K_1 \oplus K_5) \\
Y_5 &= S(\gamma(S(X_{13} \oplus K_{13})) \oplus \gamma(S(X_2 \oplus K_2)) \oplus S(X_2 \oplus K_2) \\
&\quad \oplus S(X_7 \oplus K_7) \oplus S(X_8 \oplus K_8) \oplus S(K_{14}) \oplus K_1 \oplus K_5 \oplus K_9) \\
Y_6 &= S(\gamma(S(X_1 \oplus K_1)) \oplus \gamma(S(X_6 \oplus K_6)) \oplus S(X_6 \oplus K_6) \oplus S(X_{11} \oplus K_{11}) \\
&\quad \oplus S(X_{12} \oplus K_{12}) \oplus S(K_{14}) \oplus K_1 \oplus K_5 \oplus K_9 \oplus K_{13}) \\
Y_7 &= S(\gamma(S(X_5 \oplus K_5)) \oplus \gamma(S(X_{10} \oplus K_{10})) \oplus S(X_{10} \oplus K_{10}) \\
&\quad \oplus S(X_{15} \oplus K_{15}) \oplus S(X_0 \oplus K_0) \oplus S(K_{14}) \oplus K_1)
\end{aligned}
$$

$$
\begin{aligned}
Y_8 &= S(\gamma(S(X_2 \oplus K_2)) \oplus \gamma(S(X_7 \oplus K_7)) \oplus S(X_7 \oplus K_7) \\
&\quad \oplus S(X_8 \oplus K_8) \oplus S(X_{13} \oplus K_{13}) \oplus S(K_{15}) \oplus K_2 \oplus K_6 \oplus K_{10}) \\
Y_9 &= S(\gamma(S(X_6 \oplus K_6)) \oplus \gamma(S(X_{11} \oplus K_{11})) \oplus S(X_{11} \oplus K_{11}) \\
&\quad \oplus S(X_{12} \oplus K_{12}) \oplus S(X_1 \oplus K_1) \oplus S(K_{15}) \oplus K_2 \oplus K_6 \oplus K_{10} \oplus K_{14}) \\
Y_{10} &= S(\gamma(S(X_{10} \oplus K_{10})) \oplus \gamma(S(X_{15} \oplus K_{15})) \oplus S(X_{15} \oplus K_{15}) \\
&\quad \oplus S(X_0 \oplus K_0) \oplus S(X_5 \oplus K_5) \oplus S(K_{15}) \oplus K_2) \\
Y_{11} &= S(\gamma(S(X_{14} \oplus K_{14})) \oplus \gamma(S(X_3 \oplus K_3)) \oplus S(X_3 \oplus K_3) \\
&\quad \oplus S(X_4 \oplus K_4) \oplus S(X_9 \oplus K_9) \oplus S(K_{15}) \oplus K_2 \oplus K_6)
\end{aligned}
$$

$$
\begin{aligned}
Y_{12} &= S(\gamma(S(X_{11} \oplus K_{11})) \oplus \gamma(S(X_{12} \oplus K_{12})) \oplus S(X_{12} \oplus K_{12}) \\
&\quad \oplus S(X_1 \oplus K_1) \oplus S(X_6 \oplus K_6) \oplus S(K_{12}) \oplus K_3 \oplus K_7 \oplus K_{11} \oplus K_{15}) \\
Y_{13} &= S(\gamma(S(X_{15} \oplus K_{15})) \oplus \gamma(S(X_0 \oplus K_0)) \oplus S(X_0 \oplus K_0) \\
&\quad \oplus S(X_5 \oplus K_5) \oplus S(X_{10} \oplus K_{10}) \oplus S(K_{12}) \oplus K_3) \\
Y_{14} &= S(\gamma(S(X_3 \oplus K_3)) \oplus \gamma(S(X_4 \oplus K_4)) \oplus S(X_4 \oplus K_4) \\
&\quad \oplus S(X_9 \oplus K_9) \oplus S(X_{14} \oplus K_{14}) \oplus S(K_{12}) \oplus K_3 \oplus K_7) \\
Y_{15} &= S(\gamma(S(X_7 \oplus K_7)) \oplus \gamma(S(X_8 \oplus K_8)) \oplus S(X_8 \oplus K_8) \\
&\quad \oplus S(X_{13} \oplus K_{13}) \oplus S(X_2 \oplus K_2) \oplus S(K_{12}) \oplus K_3 \oplus K_7 \oplus K_{11})
\end{aligned}
$$