DPA, Bitslicing and Masking at 1 GHz

Josep Balasch(✉), Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede

Department of Electrical Engineering-ESAT/COSIC and iMinds, KU Leuven, Kasteelpark Arenberg 10, 3001 Leuven-Heverlee, Belgium
{Josep.Balasch,Benedikt.Gierlichs,Oscar.Reparaz, Ingrid.Verbauwhede}@esat.kuleuven.be

Abstract. We present DPA attacks on an ARM Cortex-A8 processor running at 1 GHz. This high-end processor is typically found in portable devices such as phones and tablets. In our case, the processor sits in a single board computer and runs a full-fledged Linux operating system. The targeted AES implementation is bitsliced and runs in constant time and constant flow. We show that, despite the complex hardware and software, high clock frequencies and practical measurement issues, the implementation can be broken with DPA starting from a few thousand measurements of the electromagnetic emanation of a decoupling capacitor near the processor. To harden the bitsliced implementation against DPA attacks, we mask it using principles of hardware gate-level masking. We evaluate the security of our masked implementation against first-order and second-order attacks. Our experiments show that successful attacks require roughly two orders of magnitude more measurements.

Keywords: Side-channel analysis · DPA · ARM Cortex-A8 · Bitslicing · Gate-level masking

1 Introduction

Side-channel attacks allow to extract secrets, such as cryptographic keys or passwords, from embedded devices with relatively low effort. Kocher reported in his seminal paper [23] extracting cryptographic keys from the observation of the execution time taken by an implementation of Diffie-Hellman, RSA or DSS. A common characteristic of side-channel attacks is that they target concrete implementations, and thus they are oblivious to the intrinsic mathematical security of the algorithm. They can be readily applied to implementations of algorithms that are resistant to traditional mathematical attacks.

Apart from timing, many other side-channels have been discussed in the literature. Most notably, the instantaneous power consumption is a powerful side-channel for embedded devices [24], and efficient exploitation mechanisms, such as Differential Power Analysis (DPA), are known. DPA requires access to the target device to collect a number of instantaneous power consumption traces

T. Güneysu and H. Handschuh (Eds.): CHES 2015, LNCS 9293, pp. 599–619, 2015.
DOI: 10.1007/978-3-662-48324-4_30

while the device is running the cryptographic implementation. The key can be derived from the statistical analysis of the power consumption traces.

A popular variant of power analysis attacks are Electromagnetic Analysis (EMA) attacks [17,38]. EMA attacks measure the electromagnetic emanations from the device and subsequently apply similar statistical techniques as DPA. An advantage is that electromagnetic measurements do not require to establish electrical contact, thus EMA can be less invasive than conventional power analysis.

Side-channel attacks on small embedded devices, such as microcontrollers and cryptographic co-processors, are nowadays a well-understood threat and a fruitful field of academic research. However, there are only a few studies of side-channel attacks on more powerful general-purpose systems. This is highly relevant to the gradual paradigm shift towards moving the cryptographic operation to the main processor, as proposed in mobile payments and host card emulation.

In this paper we investigate the DPA susceptibility of block-cipher implementations on high-end embedded devices. As an illustrative test case, we focus on the Advanced Encryption Standard [2] (AES) and an ARM Cortex-A8 processor. This processor core is found in portable consumer electronic devices, such as phones (Apple iPhone4, Samsung Galaxy S, Google Nexus S), tablets, set-top boxes, multimedia entertainment systems (Apple TV, Apple iPod Touch 4th gen), home networking or storage appliances and printers.

The Cortex-A8 is a powerful and complex processor that features significant differences with typical targets of side-channel attacks. It is a 32-bit processor with a 13-stage pipeline, dynamic branch prediction, L1 and L2 cache memories, a rich ARMv7 instruction set and a separate SIMD execution pipeline and register file (NEON). It can run at up to 1 GHz clock frequency. At the software level, there is normally a full multi-tasking operating system with shared resources, different competing processes and interrupts. It is not clear if DPA can be successfully applied to such target devices. One goal of our work is to fill this gap.

1.1 Related Work

AES on High-End Embedded Devices. An efficient option for AES software implementations on high-end processors is the *T-table* approach due to Daemen and Rijmen [14]. Its core idea is to merge three of the four AES transformations (`SubBytes`, `ShiftRows` and `MixColumns`) into four lookup tables. At the cost of storing 4 kbytes, this method allows to compute an AES-128 encryption using only 160 table lookups and 44 XOR operations. Since the four lookup tables are rotations of each other, it is possible to reduce the memory requirements to 1 kbyte by storing a single table. For architectures with inline barrel shifter such as ARM, this characteristic can be used without performance loss [33].

While efficient, implementations based on lookup tables are a target for side-channel attacks on processors with cache memories. Exploiting cache-related

timing variabilities was already mentioned by Kocher [23], and further elaborated on by Kelsey *et al.* [21] and Page [36]. In recent years, several practical attacks against the T-table AES implementation of OpenSSL have been published, see for instance the works of Bernstein [6], Bonneau and Mironov [9] and Osvik *et al.* [34]. The root of the problem stems from the difficulty to load array entries into the CPU registers without this depending on the index pointer. As suggested by Bernstein *et al.* [7], a secure library should systematically avoid loads from addresses that depend on secret data. While one could always resort to *computing* the AES S-Box to achieve constant execution time, the performance penalties of straightforward implementations would be considerable.

It is in this context that bitsliced implementations rise as an attractive alternative for AES in software. Originally proposed by Biham [8] to improve the performance of DES in software, the idea behind bitslicing consists in describing a cryptographic algorithm as a sequence of Boolean operations which can be implemented with only bitwise instructions. Since there are no table lookups, bitsliced implementations are inherently resilient to cache timing attacks. The first bitsliced software implementation of the AES for x64 processors is due to Matsui [27]. An alternative implementation for 64-bit platforms is presented by Könighofer [25]. The advent of Single Instruction Multiple Data (SIMD) extensions on Intel Core2 processors has enabled a more efficient usage of the 128-bit XMM registers. Matsui and Nakajima [28] were first to take advantage of this and proposed a high-speed bitsliced implementation of the AES at 9.2 cycles/byte, albeit conditioned to input data blocks of 2 kbytes. More recently, Käsper and Schwabe [20] proposed the fastest implementations of AES-CTR and AES-GCM up to date, running at 7.59 cycles/byte and 10.68 cycles/byte, respectively.

Side-Channel Attacks on High-End Embedded Devices. With the notable exception of cache timing attacks, the susceptibility of high-end embedded processors to side-channel attacks has received only little attention in the literature, particularly when compared to the attention that has been given to less complex platforms. Gebotys *et al.* [18] showed how to attack Java implementations of AES and ECC running on a PDA equipped with a "mid-range" 32-bit ARM7TDMI at 40 MHz. The authors performed a differential EMA attack in the frequency domain in order to deal with the issue of trace misalignment. A follow-up work by Aboulkassimi *et al.* [3] similarly used differential EMA to attack AES implementations. The target device was a mobile phone with a 32-bit processor running at 370 MHz. Kenworthy and Rohatgi [22] applied Simple Power Analysis (SPA) and leakage detection techniques to show the susceptibility of several implementations to EMA. Although no processor frequency is specified, the acquisition bandwidth of the setup used was limited to 60 MHz. Finally, Nakano *et al.* [31] performed SPA attacks on ECC and RSA implementations running on an Android Smartphone clocked at 832 MHz.

Masking Countermeasures. A popular and well-studied countermeasure to thwart power analysis attacks is masking [12,19]. Contrary to other approaches, masking is a provable sound countermeasure and widely employed in practice.

In its simplest form, masking consists of splitting every key-dependent intermediate s that appears throughout the computation into two shares (s_1, s_2) such that $s_1 \star s_2 = s$. The group operation $\star$ is typically XOR. The splitting is such that each share s_i is statistically independent of the intermediate s. This condition should be preserved throughout the entire masked computation, and implies that knowledge of any individual s_i does not reveal any information about the intermediate s, and thus about the key.

Masking can be applied at different abstraction levels: from the algorithmic level (public key cryptography algorithms [13,30] as well as symmetric key algorithms such as DES [19] or AES [4]) to the gate level [42]. Algorithm-level masking can result in more compact implementations. However, this masking method is not a general approach as it is tied to a specific algorithm. On the other hand, gate-level masking performs the splitting at the bit level and provides the implementer with a set of secure logic gates to compute on. It it thus a versatile method to securely implement any given circuit.

1.2 Contributions

Our first contribution is to investigate the feasibility of DPA attacks on modern gigahertz embedded processors. Our experimental platform is a Sitara ARM Cortex-A8 32-bit RISC processor mounted on a Beaglebone Black (BBB) platform and running a complete Ångström Linux distribution. Our test application is a bitsliced implementation of AES-128 encryption immune to cache timing attacks. Our experiments show that the most difficult part of an attack is of practical nature (measurement point, triggering, alignment) and that basic DPA attacks succeed with a few thousand measurements. For the sake of reproducibility, we describe all steps carried out in our analysis in detail.

Our second contribution is to apply gate-level hardware masking to protect our implementation. We show that it is not difficult to equip an unprotected bitsliced implementation with masking. In addition we fully implement a masked AES on the same platform and test its resistance to first-order and second-order attacks. Our experiments show that breaking our masked implementation requires roughly two orders of magnitude more measurements than breaking the unprotected implementation.

2 A Bitsliced AES Implementation

Our test application is a bitsliced implementation of the AES based on the construction of Könighofer [25]. We adapted it for our 32-bit processor. Note that this is a poor decision if one aims for performance, i.e. bitsliced implementations pay off in software contexts only if the target processor contains large (and possibly many) registers. Nevertheless, our aim is neither to propose nor to achieve high-throughput implementations, but rather analyze bitsliced implementations from the DPA-security standpoint. In fact, and as will become clear later, our insights also apply for larger wordsize architectures such as e.g. NEON.

Hardware Description of AES. In the following we focus on AES-128 encryption. The first step towards a bitsliced description consists in describing all cipher transformations (`AddRoundKey`, `SubBytes`, `ShiftRows` and `MixColumns`) as a fixed sequence of Boolean operations. The goal is to employ only bitwise operations i.e. an equivalent gate representation in hardware contexts. The main difficulty of this process consists in finding an efficient way to compute the non-linear part of the AES S-Box.

There exist many hardware flavours of AES depending on whether they aim for throughput, area, low-power, etc. For bitsliced contexts, we are interested in *compact* implementations. Most successful designs in this direction compute the inverse in $GF(2^8)$ using subfield arithmetic, as originally suggested by Rijmen [39]. This is the case of the works due to Rudra *et al.* [40], Wolkerstorfer *et al.* [44] and Satoh *et al.* [41], the latter building also on the tower-field representation of Paar [35]. As in [25], we employ the AES S-Box representation by Canright [11] illustrated in Fig. 1.

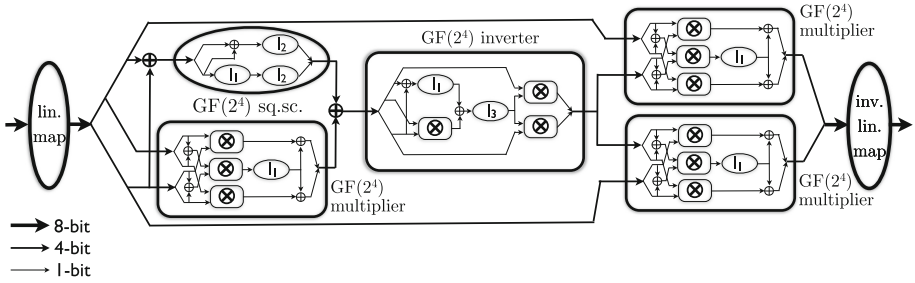


Fig. 1. AES S-Box representation due to Canright [11].

Bitsliced Format. In the standard AES representation a 128-bit input message block A is described as a 4×4 byte matrix. This is illustrated in the upper left hand side of Fig. 2. Each byte is addressed as A_i. The cipher transformations are commonly defined at byte level in order to operate on the matrix representation of the state, e.g. either at element level (`AddRoundKey` and `SubBytes`), at row level (`ShiftRows`) or at column level (`MixColumns`). This representation is however inadequate for bitsliced implementations, as all steps are defined at bit level. Therefore, one needs to find a different representation of the cipher state. The most straightforward option consists in arranging the state as a vector of 128 elements, each corresponding to a bit. This choice is however unsuitable in practice, as the state cannot be fully kept in registers and memory accesses easily become a major bottleneck.

An alternative bitsliced representation uses a more compact state of 8 elements [20,25,27], each containing a particular bit of the 16 state bytes A_i. Let us denote the bitsliced state elements by $\mathcal{R}_i$. Going to the bitsliced domain requires to split the bytes A_i and store the bits, from LSB to MSB, to the corresponding registers, $\mathcal{R}_1$ to $\mathcal{R}_8$. Note that one input message block A fills only 16 bits in each $\mathcal{R}_i$. Therefore several input messages can be processed in parallel, e.g. by

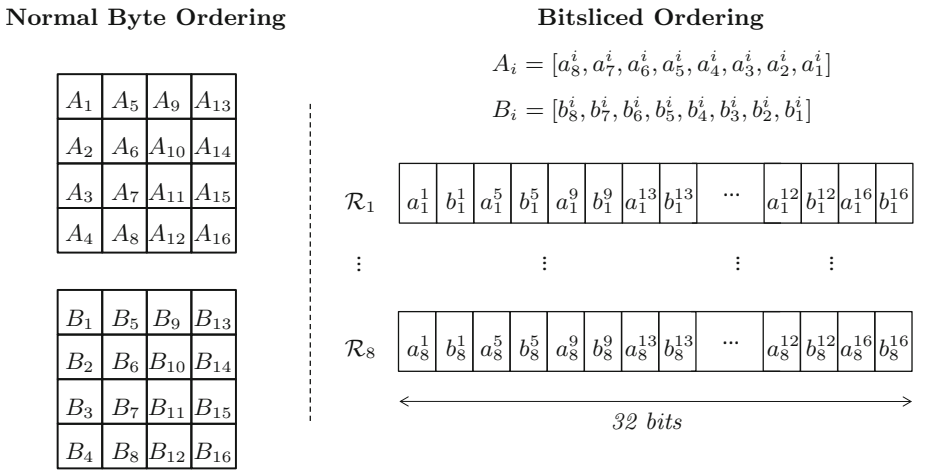


Fig. 2. Layout of bitsliced AES registers.

storing bits from several input message blocks into each register. This is illustrated in Fig. 2 for the case of 32-bit registers. In this case, a second plaintext B is processed concurrently with A.

Coding Style. We have coded our bitsliced AES implementation in C language and mimicked the concept of hardware gates by using software macros for all atomic operations. This approach allows us to write our program as a fixed sequence of calls to five main macros: bitwise operations (`XOR`, `AND` and `NOT`), data transfers (`MOV`) and left rotates (`ROTL`):

```
#define XOR(c,a,b)    c = a ^ b;
#define AND(c,a,b)    c = a &b;
#define NOT(c,a)      c = ~ a;
#define MOV(c,a)      c = a;
#define ROTL(c,a,l)   c = (a ≪ l) | (a ≫ (32 - l));
```

The main benefit of this approach is that protecting the implementation with gate-level masking requires only rewriting the macros. This point will be elaborated on in Sect. 4.

3 Developing an Attack

The BBB is a complex single board computer. The main component is a high-performance TI AM3358 Sitara System on Chip (SoC) based on the ARM Cortex-A8 core. To give an idea of the complexity, we point out that the main processor can be clocked up to 1 GHz and that the SoC features a DDR3 memory controller, a 3D graphics engine, a vast array of peripheral support (incl. USB, ethernet) and two 32-bit sub-processors (technically, programmable real-time units) for time-critical tasks, among others.

3.1 Strategies for Side-Channel Measurements

From all the components of the single board computer, we are mostly interested in the side-channel leakage of the ARM processor. An obvious way to access it would be to measure the SoC's power consumption. However, performing a power measurement on a BGA package is not straightforward, as the pins are covered by the package and not easily accessible.

A second strategy might be to measure the power consumption of the entire single board computer. At least, doing the actual measurement should not be difficult as the entire board is powered by a single 5 V supply (via USB or a dedicated connector, e.g. if more than 500 mA is needed). But we expect the global power consumption to be very noisy (many active components on the board). Furthermore, there is a dedicated power management IC and numerous decoupling capacitors between the power supply and the SoC. The high operating frequencies of the processor require capacitor banks that can deal with low, medium and high frequencies.

The third approach is the one we actually followed. We opted for a "contactless power measurement". To clarify, others have used the same technique and called it "electromagnetic measurement". We do use an electromagnetic pen probe but we do not aim at measuring emanations from the ARM processor. Rather, we measure the EM field around different components on the board that are somehow involved in the current loop to the ARM core. In general, voltage regulators and decoupling capacitors [15,32] are promising candidates. In our case, the dedicated power management IC is quite complex and physically located far away from the ARM core. For these reasons, we do not think that it would provide a useful signal. Therefore the decoupling capacitors are the best candidates. In general, the closer the capacitor is to the processor, the better signal it can provide. In summary, we use an electromagnetic probe to measure a signal that is correlated with the chip's power consumption. We therefore think of the technique as a contact-less power measurement.

3.2 Experimental Setup

Our experimental setup comprises:

– A stock BBB platform running a complete Linux Ångström distribution. We did not modify the software or the hardware and operate the board in its factory configuration. The Linux distribution is based on Debian 7 with kernel version `3.8.13-bone47 (root@imx6q-wandboard-2gb-0)`. This is a preemptive multitasking operating system with plenty of simultaneously running processes. We did not switch off any running service. The command `ps aux` reports 102 processes running on the system. Among others, we found running the `Xorg` graphical server (with the onboard HDMI driver output activated), the `apache2` webserver (including the `nodejs` server-side javascript runtime environment, to our surprise) and the `sshd` server (with an open session running throughout all experiments for monitoring purposes). We power

the board via the USB connection from the measurement PC. We did not make any effort to supply the board with a particularly clean voltage. The board is connected to the measurement PC via ethernet-over-USB. We did not disable the blinking blue leds that indicate activity.
- A Langer magnetic near field probe, model RF-B. The reported frequency bandwidth is from 30 MHz to 3 GHz [16].
- A wideband 30 dB low-noise amplifier from Langer, model PA 303. The reported frequency bandwidth goes from DC to 3 GHz.
- A Tektronix DPO70404C 8-bit oscilloscope with an analog bandwidth of 4 GHz. Most of the time we sampled at 6 GS/s to make full use of our setup's bandwidth (Nyquist rate).

We use the Linux command `cpufreq-set -f 1000 MHz` to bring the system into a high-performance state. In this state the board cannot enter low-power mode and the processor core is permanently clocked at 1 GHz, which is well within the bandwidth of our measurement setup. Figure 3 shows a photo of our experimental setup (left) and a representation of our EM probe and the field orientation it is able to register (right).

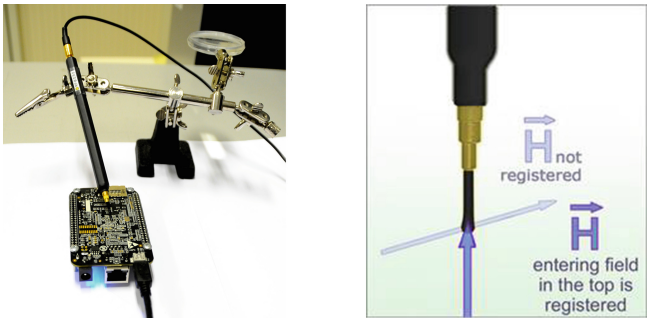


Fig. 3. Photo of our setup (left) and EM antenna schematic [16] (right).

3.3 Approach

Our first step is to find a suitable measurement position for the EM antenna. The EM antenna tip is small enough (we measured a diameter of 2 mm) to allow us to get in between components and to measure individual components' EM field without picking up too many signals from neighboring components. For the purpose of locating an appropiate position for the EM antenna, we wrote a short C program that exercises memory accesses and ran it on the BBB in a loop. The program executes 1000 NOPs, then repeatedly fills a buffer in memory with the value `0x00000000` and then with the value `0xffffffff` 1000 times, followed by again 1000 NOPs. We manually move the antenna over the PCB surface and slowly from component to component. We carefully monitor the sampled signal on the oscilloscope for a pattern that looks correlated to the execution of our

C code. We begin doing this by trial and error, and we focus on the capacitors in the SoC's power supply as explained above. Note that this is a tedious task because we need to get not only the probe's tip in the right *location* but we also need to get the probe in the right *orientation* (see Fig. 3 right). As the search was very time consuming we had a look at the BBB PCB schematics [1]. We identified a bank of capacitors in the SoC's VDD core supply. They should be good candidates for measurement points as their EM fields should contain a lot of useful signal about the processor core's activity. Next we locate these decoupling capacitors on the PCB and manually scan them with the EM probe one by one.

We did not find a useful signal around these capacitors (that does not mean there is no useful signal) and reverted to trial and error testing of other decoupling capacitors in the SoC's supply network. Eventually we found a good signal near C66 (see Appendix A in the full version of this paper [5]), a 0.1 μF multilayer ceramic capacitor in a 3.3 V supply rail. We used this probe position and orientation for all measurements and did not further explore the board for other useful signals.

Now that we found a suitable measurement point, the next step is to deal with the timing and triggering. We run our bitsliced implementation of AES-128 encryption from Sect. 2 on the BBB. We can send the inputs from the measurement PC and read back the outputs as well. Recall that we keep an SSH connection open between the measurement PC and the BBB for this purpose and for monitoring. After some trial and error work we find a good pattern (related to I/O) to trigger the oscilloscope on. Figure 4 shows the plot of an overview measurement. Note that we need to substantially lower the sampling rate for some of these long measurements. The execution of our C program causes the dense pattern in the middle of the plot. The isolated peaks left and right of the dense pattern are caused by other processes.

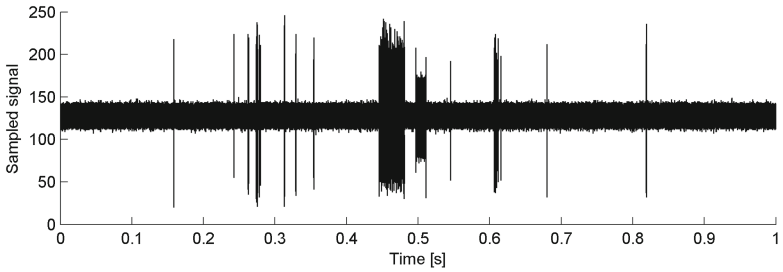


Fig. 4. Overview measurement of our unprotected AES.

Figure 5 shows the plot of a more focused measurement. We see patterns caused by the reception of 34 bytes (two plaintext blocks of 16 bytes each and two control words) followed by patterns caused by the AES encryption in the

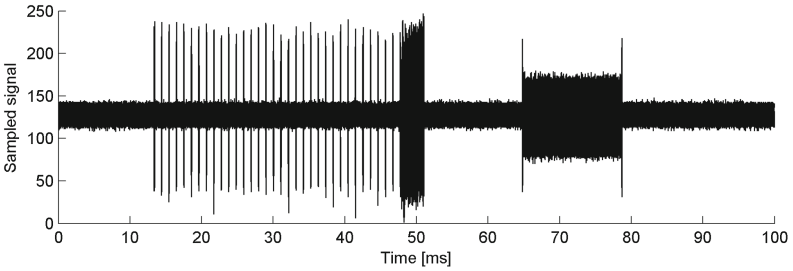


Fig. 5. Overview measurement of our unprotected AES.

middle of the figure. We do not know what causes the "block" pattern on the figure's right hand side.

Figure 6 shows a zoom on the patterns caused by the AES operation. It is tempting to let the human eye search for patterns of the ten AES rounds, but in fact the AES makes only a small part of this measurement (as marked by the dotted red rectangle in the figure). We are not sure what the other processing is. We know that some of it is the conversion of plaintexts to the bitsliced format, and from bitsliced format to ciphertexts.

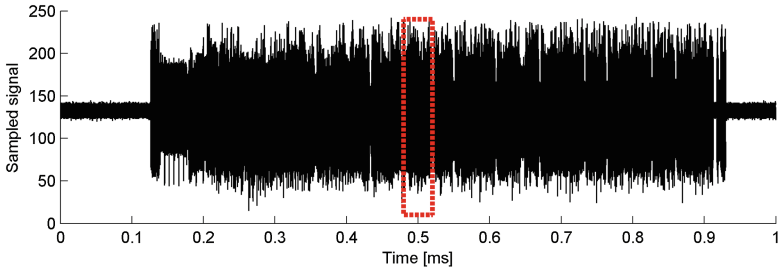


Fig. 6. Measurement of our unprotected AES.

Figure 7 shows a plot of a measurement with the actual AES-128 encryption in the middle of the plot. One would expect to see a sequence of nine very similar patterns (the first nine AES rounds) followed by a different pattern (the tenth round without `MixColumns`). However, in this figure we see a sequence of only eight very similar patterns followed by a different pattern. It seems that our measurement is missing one normal round. Our experiments confirm that what we recognize corresponds to rounds two to ten. The execution of the first AES round leads to a pattern that is more scattered over time, but it fills up the instruction cache so that the next rounds are executed much faster which leads to a more dense and clear pattern.

When we execute several encryptions in a batch, the second and all following encryptions typically run from cache and show clear patterns also for the first

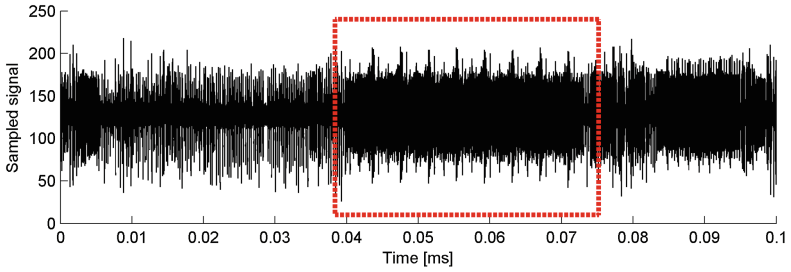


Fig. 7. One of the first measurements of our unprotected AES.

round that we can use for alignment. Figure 8 shows the single-sided amplitude spectrum of a measurement. The spectrum shows a clear and sharp peak at 1 GHz.

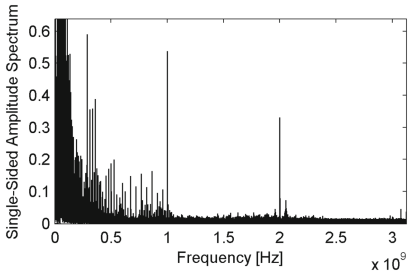


Fig. 8. Single-sided amplitude spectrum of a measurement.

Even though we found a seemingly stable trigger, the measurements of the AES encryption are actually heavily desynchronized. Recall that we are working with a high-end ARM processor on a complex SoC and that our C program is only one of more than 100 running processes (and we do not run it with elevated priority!). Therefore, filtering out mis-triggered measurements and carefully aligning the remaining measurements is crucial. In fact, we spend about seven times more time on the post-processing than on the measurement.

3.4 Attack

We aim to break our unprotected implementation with a first-order correlation DPA attack [10] against the first round. The next step is to try to find a pattern in the traces that is related to the (S-box computation in the) first round, and to align all useful traces on that pattern. Finally we try to attack the implementation with 10 000 aligned measurements.

We need to think about a power model because the implementation is bitsliced. A typical byte-oriented implementation uses an S-box table in memory

and processes the AES state byte by byte. The key point here is that all 8 bits of an S-box output are computed (or looked-up) at the same time. Hence one can expect all bits of the S-box output to leak at the same time and this gives rise to the commonly used "Hamming weight of the S-box output" power model. This is different for our bitsliced implementation. The eight S-box output bits are computed one after the other and stored in eight different registers. So if we assume for a moment that the implementation processes one plaintext block at a time, each of the eight registers holds 16 bits. For instance register $\mathcal{R}_1$ stores the 16 LSBs of the 16 state bytes. With the usual divide and conquer approach we aim to recover the key byte by byte. If we make a guess about one key byte we can predict one S-box output but the eight bits are spread over eightregisters that are not processed at the same time. For a normal univariate attack we can therefore exploit only one bit effectively. The other 15 bits in the same register are algorithmic noise. If we want to exploit more bits in the same register we need to guess more key bytes, which quickly becomes computationally expensive. Alternatively we can think to attack each of the eight bits of one S-box output separately and then perform some majority voting, but we did not investigate this approach.

Now in our implementation the situation is similar but it actually processes two plaintext blocks in parallel. This means for instance that register $\mathcal{R}_1$ stores 32 LSBs, 16 of one plaintext and 16 of the other. As the key is fixed both plaintext blocks get encrypted under the same key. Making a guess on one key byte we can attack both encryptions at the same time and predict two bits in a register (2 out of 32 instead of 1 out of 16 in the example above). Our power model is hence the Hamming weight (HW) of two bits in a register that are affected by the same sub-key. We stress that this observation has an *important consequence*: processing more plaintext blocks in parallel does not make an attack harder if the adversary is aware of the bitsliced implementation. In fact, the ratio of predicted bits and processed bits, and hence the ratio of signal to algorithmic noise, is constant.

Figure 9 shows an exemplary result of a 2-bit attack against one key byte. The plot on the left hand side shows the correlation traces for all key hypotheses obtained using 10 000 measurements. The trace for the correct key hypothesis is plotted in black. The plot shows that the correct key hypothesis leads to a distinguishable and clear correlation peak. The plot on the right hand side shows the highest and lowest correlation value for each key hypothesis (from the overall time frame) over the number of measurements. In addition we also plot the 99.99 % confidence interval for sample correlation equal to zero (dashed lines). The plot shows that only few thousand measurements are required for the correct key (black line) to stand out and hence for the attack to succeed.

Attacks targeting the same (other) key byte(s) using the leakage of other (the same) register(s) give very similar results. Surprisingly full key recovery is hence possible using only a few thousand measurements!

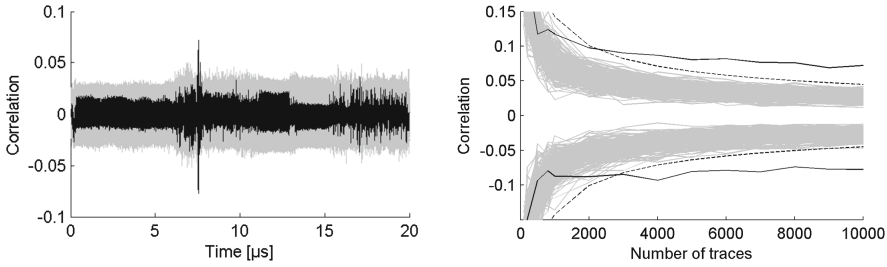


Fig. 9. Result of attack against unprotected implementation.

4 Masking a Bitsliced AES Implementation

Since a bitsliced software implementation mimics a hardware circuit, gate-level masking appears as a very attractive candidate to protect our AES implementation. Applying gate-level masking to an already existing implementation can actually be done in a pretty straightforward manner. It only requires to protect the design's elementary Boolean functionalities, while the original sequence of operations remains unmodified. A direct consequence of this is that any optimization performed in the unprotected implementation, e.g. to improve the design's throughput, is automatically inherited by the protected implementation.

Generally, linear functions such as the XOR gate, are trivial to mask by just computing on each share independently. The challenging part of gate-level masking is to provide a construction for non-linear gates. One of the first works tackling this problem is due to Trichina [43]. Trichina gives a secure AND gate that takes two shares of each input bit a, b and produces two output shares of $c = a \cdot b$. The secure AND gate consumes one fresh random bit r. If the input bit a (resp. b) is shared into a_1 and a_2 (resp. b_1 and b_2), the two output shares c_1, c_2 of the Trichina gate are defined as

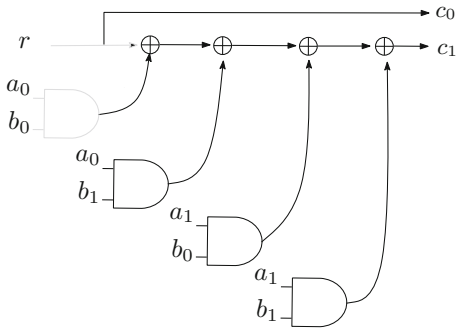
$$c_1 = r \tag{1}$$

$$c_2 = (((a_1 b_1 \oplus r) \oplus a_1 b_2) \oplus a_2 b_1) \oplus a_2 b_2. \tag{2}$$

It is easy to verify that this AND gate description is correct, namely, that the output shares XOR to $a \cdot b$. The description is also secure against first-order attacks: each variable occurring during the execution is independent of any unshared value a, b or c. Note however that the order of partial computations of c_2 is relevant for the security of the gate.

Masked Bitsliced Format. Applying first-order Boolean masking requires to split any sensitive intermediate variable s into two shares such that $s = s_1 \oplus s_2$. For our implementation, this implies that each of the eight original state registers $\mathcal{R}_i$ becomes a pair $(\mathcal{R}_i^1, \mathcal{R}_i^2)$ such that $\mathcal{R}_i = \mathcal{R}_i^1 \oplus \mathcal{R}_i^2$. We denote $\mathcal{R}_i^1$ as mask state and $\mathcal{R}_i^2$ as masked state. The plaintext in bitsliced format is shared in this way at the beginning of the execution.

Masked Operations. Our original bitsliced implementation employs only five operations which are described as macros. These are: `XOR`, `AND`, `NOT`, `MOV` and `ROTL`. Our masked implementation substitutes each occurrence of these by its secure equivalent: `SXOR`, `SAND`, `SNOT`, `SMOV` and `SROTL`, respectively. The secure equivalents operate sequentially on the mask state $\mathcal{R}_i^1$ and the masked state $\mathcal{R}_i^2$. The implementations of `SXOR`, `SMOV` and `SROTL` are trivial, as they consist of implementing the corresponding `XOR`, `MOV` or `ROTL` twice: one for $\mathcal{R}_i^1$ and another for $\mathcal{R}_i^2$. In a similar way, the secure `SNOT` is simply computed by applying `NOT` to one of the shares. The implementation of `SAND` is more elaborate and follows closely the lines of the Trichina gate. A circuit representation of the gate is shown in the left part of Fig. 10, while its macro representation is given in the right part of Fig. 10.




```
#define SAND(c, a, b){
  t0 = a[0] & b[0];
  t1 = a[0] & b[1];
  c[0] = RAND();
  t0 = t0 ^ c[0];
  t0 = t0 ^ t1;
  t1 = a[1] & b[0];
  t0 = t0 ^ t1;
  t1 = a[1] & b[1];
  t0 = t0 ^ t1;
  c[1] = t0;}
```

Fig. 10. Left: Trichina construction for the masked AND gate. Right: pseudocode for the `SAND` operation following the Trichina AND construction.

In contrast to the original bitsliced macros, each variable in the macro is now an array of 2 elements: mask state and masked state. Each of the two input arrays (`a`, `b`) is thus composed of two registers (`a[0]`,`a[1]`) and (`b[0]`,`b[1]`), respectively. Two temporal registers (`t0` and `t1`) are additionally used to preserve the correctness of the macro in case one of the source registers is also the destination, e.g. to prevent errors when the macro is called as `SAND(a,a,b)`. The result is placed in the output array `c` composed of registers `c[0]` and `c[1]`.

Randomness Generation. The two operations that require randomness in the masked bitsliced implementation are the initial plaintext sharing and each `SAND` operation. We use the kernel's `/dev/urandom` cryptographic RNG to obtain the required randomness. We do not read a single byte each time a random byte is needed, instead, we read a chunk of randomness and place it in an internal buffer at the beginning of each encryption. Then, during the actual encryption the randomness is simply taken from this internal buffer. We implemented this mechanism to minimally interrupt the execution of the encryption and get clean measurements.

We note that masking typically does not need cryptographically strong random numbers for the masks. Although we used a cryptographically strong source of randomness for the masks, a lighter RNG can be used if needed, e.g. for performance reasons. When we later report that the RNG is switched off, we fill the internal buffer from `/dev/zero` instead of from `/dev/urandom`.

Performance. We have compiled our implementations directly in the BBB using the compiler version available in the Ångström Linux distribution, i.e. gcc version 4.6.3. No special flags have been used. The throughput loss of the protected implementation is roughly a factor 5 compared to the unprotected implementation. Further, the RAM usage increases by 32 bytes because of doubling the register state size. Our internal buffer for storing random numbers holds 2048 bytes. The only macro in our implementation that consumes randomness is `SAND`, which is used 37 times during the calculation of `SubBytes`. Taking into account that 32 bytes are required to mask the input plaintexts, this gives $32 + 10 \times (37 \times 4) = 1512$ random bytes per AES execution, or equivalently, 756 bytes per plaintext block.

5 Evaluation of Masked Implementation

In this section we evaluate the DPA resistance of our bitsliced and first-order masked AES implementation.

5.1 Attack When RNG is Off

We first attack the implementation with the RNG switched off. In this case the implementation is effectively unprotected and we aim to break it with the same first-order attack as before: we guess one key byte and use the HW of the two affected bits in a register as power model. Since the code is different, the shape of the measurements is different as well, and we need to work through trial and error again to find a good pattern for trace alignment. And this is where the fact that the implementation is effectively unprotected helps. We should be able to break it easily, and we can use the attack result to judge and improve the discarding of mis-triggered measurements and the alignment step until we are satisfied. As a side note we also mention that we need to take longer measurements because in particular the S-box computation takes more time.

Figure 11 shows the result of an exemplary attack against one key byte. The plot on the left hand side shows that using 10 000 measurements the correct key hypothesis leads to a clear correlation peak. The plot on the right hand side confirms that, if the RNG is switched off, our masked implementation is as insecure as the unprotected implementation and can be broken with a few thousand measurements. Also in this case attacks targeting other key bytes and using the leakage of other registers give similar results. Full key extraction with a first-order attack is possible with a few thousand measurements.

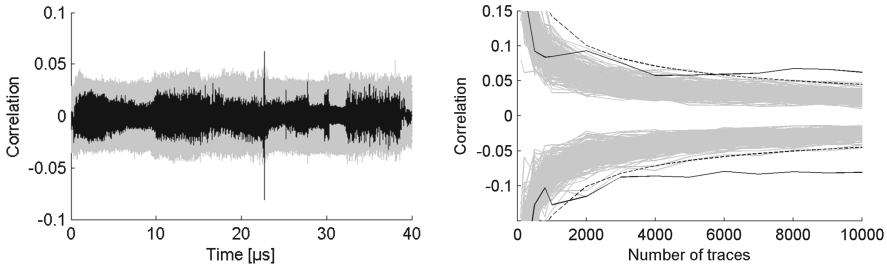


Fig. 11. Result of attack against masked implementation with RNG off.

5.2 Attack When RNG is On

Now we switch on the RNG and evaluate how much protection our masked implementation provides. Having performed the attacks with the RNG switched off has two important advantages. First, we can keep all settings for triggering, for discarding mis-triggered measurements, and for alignment because the executed code is exactly the same and the general shape of the measurements does not change. And second, we know exactly when the S-box computations are performed and we can therefore narrow down the time window to analyze (including some margin).

It is well known that implementing masking securely in software is very difficult, and we do not expect our first attempt to mask a bitsliced implementation to provide a high level of resistance to attacks. Nevertheless, to ensure that we have enough measurements at hand to break our implementation we acquired 2 000 000 measurements. We stress that trace acquisition is rather quick, but in contrast to most academic works we have to deal with the computationally intensive and hence slow post-processing (discarding mis-triggered measurements and alignment). After post-processing we are left with about 1.2 million aligned measurements.

We applied the same 2-bit first-order DPA attack as before in various settings, targeting different key bytes and registers. The results differ a lot depending on the specific setting. To give an idea of the range, we provide two results in Fig. 12. They target different key bytes and registers but both plots on the left hand side are computed using 1.2 million measurements. While in the upper plots the attack clearly succeeds and requires about 600 000 measurements, the attack in the lower plots fails even if using 1.2 million traces. Nevertheless, we confirmed that, using alternative combinations of target key byte and register, full key extraction with first-order DPA and using 1.2 million measurements is possible.

Considering the well known difficulties with masking in software and the surprisingly easy attacks against the unprotected implementation, we expected attacks against our masked implementation to succeed with much less traces. Our results are therefore promising and good news for the idea to combine bitsliced software and gate-level masking. Recall that our implementations are

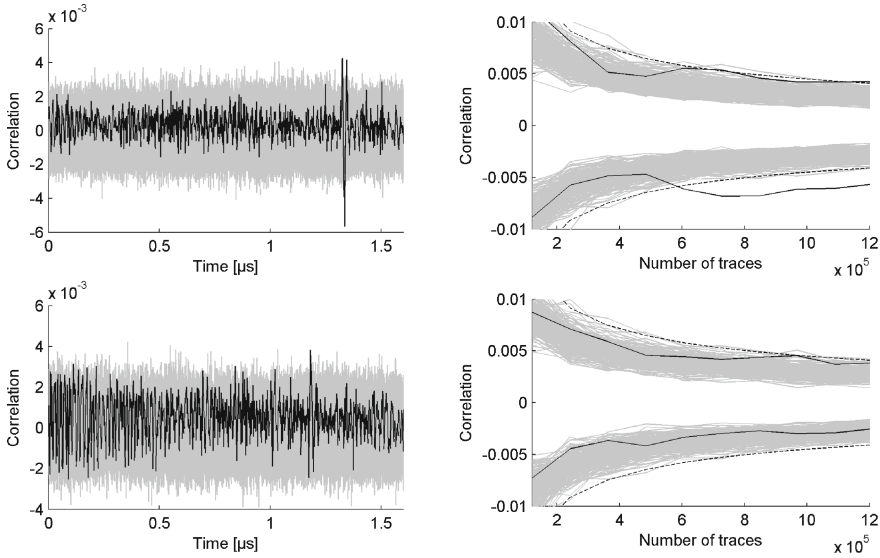


Fig. 12. Results of first-order attacks against masked implementation with RNG on.

coded in C, processed by a compiler and we have little control over the code that is eventually executed. Also, we stress that this is our first attempt to mask the implementation. The fact that the result of each individual Boolean operation is registered in a bitsliced implementation probably helps. Glitches that are an issue for masked *hardware* implementations [26] are no threat here.

We also performed a few exemplary univariate and bivariate second-order attacks. Concretely, we processed the measurements to combine each pair of time samples using the absolute difference combination function [29] or the centered product combination function [37]. This combination step yields a combinatorial blow-up in the number of time sample pairs to be analyzed jointly and makes these attacks very computationally expensive.

We then applied the same 2-bit attack to the combined measurements. Figure 13 depicts the results of an exemplary attack using the absolute difference combination function. The plot on the left hand side shows the maximum absolute value of the correlation coefficient for each key byte hypothesis across *all pairs* of time samples when using 1.2 million measurements. The correct value (indicated by a dashed vertical line) clearly stands out. For the plot on the right hand side we restrict the analysis to the single pair of time samples for which the correct key guess gives maximal correlation. The plot shows that the attack can be successful starting from around 400 000 measurements if the adversary already knows which pair of time samples to analyze. In other words, a more realistic attack will very likely require more measurements to succeed. An analysis over all pairs of time samples is, however, computationally expensive.

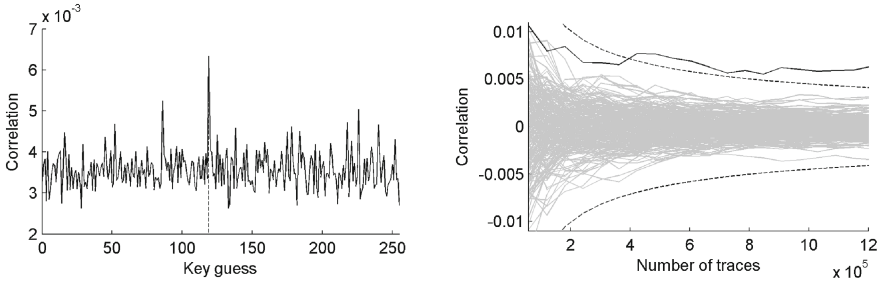


Fig. 13. Result of second-order attacks against masked implementation with RNG on.

Our results lead to two interesting observations. First, the second-order attack only works when we use the absolute difference combination function. A similar attack using the centered product combination function is unsuccessful. This is in contrast with theoretical results proving the optimality of the centered product combination function for second-order attacks [37] in the "Hamming weight leakage and Gaussian noise" model. We assume that our scenario does not meet this model and that the absolute difference combination function has a wider scope. And second, the number of measurements required for a successful first-order attack is not substantially lower than the number of measurements needed for our "idealized" second-order attack. This indicates that our masking is effective (albeit its implementation is not perfect).

6 Conclusion

The threat of side-channel attacks to the security of microcontrollers and cryptographic co-processors appears to be well understood by both industry and academia. Yet the same cannot be said for high-end embedded processors as used in phones and tablets. In this situation one may naturally wonder whether such complex, high-performance devices operating in the GHz range and executing multitasking operating systems are at all vulnerable to DPA. In this work we answer this question positively. By means of experiments we show that DPA attacks against constant-time bitsliced implementations of the AES running on a 1 GHz ARM Cortex-A8 processor are not only possible, but in fact rather easy to mount. The most challenging parts of an attack are triggering and trace alignment. Finally, we mask our implementation inspired by gate-level masking and evaluate its resistance against first-order and second-order DPA attacks. Our results indicate that the implementation is more secure than we anticipated and therefore highlight the potential of combining bitsliced software implementations and gate-level masking.

Acknowledgements. We would like to thank the CHES 2015 reviewers for their valuable feedback. This work has been supported in part by the Research Council of KU Leuven (GOA/11/007), by the Flemish Government FWO G.0550.12N and by the

Hercules foundation (AKUL/11/19). Oscar Reparaz is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO). Benedikt Gierlichs is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (FWO).

References

1. BBB PCB schematics. http://elinux.org/Beagleboard:BeagleBoneBlack#Board_Revisions_and_Changes
2. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards (FIPS) Publication 197 (2001)
3. Aboulkassimi, D., Agoyan, M., Freund, L., Fournier, J., Robisson, B., Tria, A.: ElectroMagnetic analysis (EMA) of software AES on Java mobile phones. In: Information Forensics and Security - WIFS 2011, pp. 1–6. IEEE (2011)
4. Akkar, M.-L., Giraud, C.: An Implementation of DES and AES, Secure against Some Attacks. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 309–318. Springer, Heidelberg (2001)
5. Balasch, J., Gierlichs, B., Reparaz, O., Verbauwhede, I.: DPA, Bitslicing and Masking at 1 GHz. Cryptology ePrint Archive, Report 2015/727 (2015). http://eprint.iacr.org/
6. Bernstein, D.J.: Cache-timing attacks on AES (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
7. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LatinCrypt 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012)
8. Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 260–272. Springer, Heidelberg (1997)
9. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Heidelberg (2006)
10. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
11. Canright, D.: A very compact S-box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005)
12. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
13. Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
14. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography. Springer, Heidelberg (2002)
15. Danis, A.U., Ors, B.: Differential power analysis attack considering decoupling capacitance effect. In: Circuit Theory and Design - ECCTD 2009, pp. 359–362 (2009). doi:10.1109/ECCTD.2009.5274996
16. Langer EMV. Probe specification. http://www.langer-emv.com
17. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: concrete results. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 251–261. Springer, Heidelberg (2001)

18. Gebotys, C.H., Ho, S., Tiu, C.C.: EM analysis of Rijndael and ECC on a wireless Java-based PDA. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 250–264. Springer, Heidelberg (2005)
19. Goubin, L., Patarin, J.: DES and differential power analysis. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 158–172. Springer, Heidelberg (1999)
20. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer, Heidelberg (2009)
21. Kelsey, J., Schneier, B., Wagner, D., Hall, C.: Side channel cryptanalysis of product ciphers. J. Comput. Securi. **8**(2/3), 141–158 (2000)
22. Kenworthy, G., Rohatgi, P.: Mobile Device Security: The case for side-channel resistance (2012). http://www.cryptography.com/technology/dpa/dpa-research.html
23. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
24. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
25. Könighofer, R.: A fast and cache-timing resistant implementation of the AES. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 187–202. Springer, Heidelberg (2008)
26. Mangard, S., Popp, T., Gammel, B.M.: Side-channel leakage of masked CMOS gates. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 351–365. Springer, Heidelberg (2005)
27. Matsui, M.: How far can we go on the x64 processors? In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, pp. 341–358. Springer, Heidelberg (2006)
28. Matsui, M., Nakajima, J.: On the power of bitslice implementation on Intel Core2 processor. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 121–134. Springer, Heidelberg (2007)
29. Messerges, T.S.: Using second-order power analysis to attack DPA resistant software. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, p. 238. Springer, Heidelberg (2000)
30. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Power analysis attacks of modular exponentiation in smartcards. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 144–157. Springer, Heidelberg (1999)
31. Nakano, Y., Souissi, Y., Nguyen, R., Sauvage, L., Danger, J.-L., Guilley, S., Kiyomoto, S., Miyake, Y.: A pre-processing composition for secret key recovery on Android smartphone. In: Naccache, D., Sauveron, D. (eds.) WISTP 2014. LNCS, vol. 8501, pp. 76–91. Springer, Heidelberg (2014)
32. O'Flynn, C., Chen, Z.: A case study of side-channel analysis using decoupling capacitor power measurement with the OpenADC. In: Garcia-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N., Miri, A., Tawbi, N. (eds.) FPS 2012. LNCS, vol. 7743, pp. 341–356. Springer, Heidelberg (2013)
33. Osvik, D.A., Bos, J.W., Stefan, D., Canright, D.: Fast software AES encryption. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 75–93. Springer, Heidelberg (2010)
34. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
35. Paar, C.: Efficient VLSI architectures for bit-parallel computation in Galois fields. PhD thesis, University of Essen (1994)

36. Page, D.: Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Cryptology ePrint Archive, Report 2002/169 (2002). http://eprint.iacr.org/
37. Prouff, E., Rivain, M., Bevan, R.: Statistical analysis of second order differential power analysis. IEEE Trans. Comput. **58**(6), 799–811 (2009)
38. Quisquater, J.-J., Samyde, D.: ElectroMagnetic Analysis (EMA): measures and counter-measures for smart cards. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 200–210. Springer, Heidelberg (2001)
39. Rijmen, V.: Efficient implementation of the Rijndael S-box (2001)
40. Rudra, A., Dubey, P.K., Jutla, C.S., Kumar, V., Rao, J.R., Rohatgi, P.: Efficient Rijndael encryption implementation with composite field arithmetic. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 171–184. Springer, Heidelberg (2001)
41. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A Compact Rijndael hardware architecture with S-box optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001)
42. Messerges, T.S., Dabbish, E.A., Puhl, L.: Method and apparatus for preventing information leakage attacks on a microelectronic assembly. US Patent 6,295,606, 25 September 2001
43. Trichina, E.: Combinational Logic Design for AES SubByte Transformation on Masked Data. Cryptology ePrint Archive, Report 2003/236 (2003). http://eprint.iacr.org/
44. Wolkerstorfer, J., Oswald, E., Lamberger, M.: An ASIC implementation of the AES SBoxes. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 67–78. Springer, Heidelberg (2002)