Power Analysis Resistant AES Implementation with Instruction Set Extensions

Stefan Tillich and Johann Großschädl

Graz University of Technology,
Institute for Applied Information Processing and Communications,
Inffeldgasse 16a, A–8010 Graz, Austria
{Stefan.Tillich,Johann.Groszschaedl}@iaik.tugraz.at

Abstract. In recent years, different instruction set extensions for cryptography have been proposed for integration into general-purpose RISC processors. Both public-key and secret-key algorithms can profit tremendously from a small set of custom instructions specifically designed to accelerate performance-critical code sections. While the impact of instruction set extensions on performance and silicon area has been widely investigated in the recent past, the resulting security aspects (i.e. resistivity to side-channel attacks) of this particular design approach remain an open research topic. In this paper we discuss and analyze different techniques for increasing the side-channel resistance of AES software implementations using instruction set extensions. Furthermore, we propose a combination of hardware and software-related countermeasures and investigate the resulting effects on performance, cost, and security. Our experimental results show that a moderate degree of protection can be achieved with a simple software countermeasure. Hardware countermeasures, such as the implementation of security-critical functional units using a DPA-resistant logic style, lead to much higher resistance against side-channel attacks at the cost of a moderate increase in silicon area and power consumption.

Keywords: Advanced Encryption Standard, instruction set extensions, embedded RISC processor, SPARC V8 architecture, power analysis, SCA resistance.

1 Introduction

The advent of side-channel attacks some ten years ago has forced the crypto community to completely reconsider the way cryptosystems are designed and implemented [8, 9]. While "traditional" cryptanalysis is primarily the domain of mathematicians who try to solve the hard mathematical problem upon which the security of a cryptosystem is based, the art of side-channel cryptanalysis aims to break concrete implementations of cryptosystems by using information (e.g. power consumption, timing, or electromagnetic emanation) leaking from a device while performing cryptographic calculations. It has been demonstrated in a number of publications that side-channel cryptanalysis is a highly effective yet relatively cheap tool to reveal sensitive data (e.g. secret keys) from unprotected or insufficiently protected implementations of cryptosystems [9]. Therefore, it is of little surprise that side-channel attacks have initiated a large body of research on countermeasures. Today, a variety of effective countermeasures for public-key cryptosystems, in particular elliptic curve cryptosystems, are known to provide good protection against side-channel attacks (SCAs) at the expense of a moderate increase in computation time. Secret-key cryptosystems like the Advanced Encryption Standard (AES) [11] have also been the subject of intensive research and numerous countermeasures for both hardware and software implementation have been proposed. In real-world cryptographic applications, public-key algorithms tend to be used rather infrequently (e.g. for key exchange) compared to secret-key algorithms performing such tasks like bulk encryption. Therefore, a severe degradation of performance in order to increase implementation security is normally less acceptable for secret-key algorithms.

One of the most important side-channel attacks against secret-key cryptosystems like the AES is differential power analysis (DPA) [8]. Countermeasures for defending against DPA can be divided into two categories: hiding and masking [9]. Both countermeasures try to reduce the correlation between the sensitive data and the observable side-channel information. Hiding tries to minimize the influence of the sensitive data, while masking randomizes the connection between sensitive data and the observable physical value. Both hiding and masking can be realized in hardware or in software; the former generally offers more possibilities than the latter. Well-known examples for SCA countermeasures are masking at the algorithmic level (e.g. [2]), randomization of operations (e.g. [9]), and the use of secure logic styles (e.g. [17]). Software countermeasures trade performance for security (i.e. entail a performance degradation), while hardware countermeasures tend to increase silicon area and power consumption.

A new implementation option for secret-key cryptosystems, which has become very popular in the last few years, is the use of instruction set extensions integrated into general-purpose processors [13, 14]. In this paper we investigate possible countermeasures against side-channel attacks to secure such implementations. As instruction set extensions emphasize the hardware/software co-design paradigm, we also investigate the problem of secure implementation by having an integrated view of both hardware and software components.

This paper is organized as follows: In Section 2 we briefly review the instruction set extensions for AES which we will use for our evaluation. Section 3 analyzes side-channel leakage in the context of extended processors. Sections 4, 5 and 6 describe different solutions for increasing the security of cryptographically enhanced processors. In Section 7 these approaches are compared with regard to a number of implementation aspects. We draw conclusions in Section 8.

2 Instruction Set Extensions for Cryptography

The concept of instruction set extensions has been very successful in different application domains like multimedia and signal processing. Therefore, it does

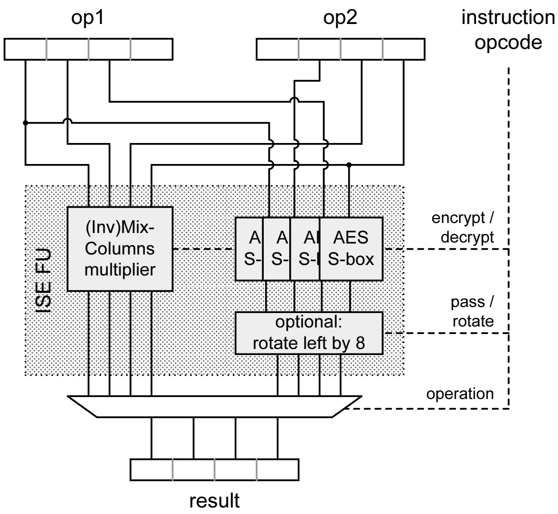

Fig. 1. Instruction set extensions for AES as proposed in [14]

not come as a surprise that researchers have also explored the benefits of this approach in the context of cryptography. The principle idea is to integrate a small set of custom instructions into a general-purpose processor with the goal to accelerate the processing of cryptographic workloads [13]. General-purpose processors with cryptography extensions combine high flexibility (due to the base instruction set) with high performance for cryptographic applications (due to the custom instructions). Extensions for public-key cryptography, like the ones outlined in [5], have focussed on a broad support of different cryptographic algorithms by accelerating common basic operations, e.g. long integer modular multiplication. On the other hand, extensions for secret-key algorithms have often been limited to a single algorithm like AES [1, 4, 10, 14].

The aspect of implementation security in the context of instruction set extensions has been largely left open; the only exception is the work described in [15], which deals with pure software countermeasures for AES implementations on 32-bit processors, both with and without the availability of instruction set extensions. The authors concluded that custom instructions help to integrate effective countermeasures, but also that the resulting impact on performance is enormous. In the present work we have tried to improve the resistance against side-channel attacks by modifying the implementation of the instruction set extensions and the base processor. The authors of [15] used a subset of the AES extensions published in [14] for their considerations. In order to facilitate the comparison of our results with those of [15], we based our work on the same set of extensions. The HDL source code of the processor core and the extensions for AES is available for download from the homepage of the ISEC project [7].

Figure 1 gives a graphical representation of the functional unit for the AES instruction set extensions (ISE FU) proposed in [14]. The two 32-bit operands entering the functional unit are denoted with *op1* and *op2*. The opcode of the instruction configures the operation of the functional unit, yielding the 32-bit value *result*. The functional unit is capable to perform the AES transformations SubBytes, ShiftRows, and MixColumns, as well as their respective inverses. Using the custom instructions allows one to execute a 128-bit AES encryption in 196 clock cycles. For comparison, a standard software implementation without instruction set extensions requires 1,637 cycles.

3 Side-Channel Attacks on Instruction Set Extensions

In general, side-channel attacks on a cryptographic device are possible when intermediate values with a close relation to the key have an influence on an externally observable physical value. If an attacker can measure this physical value while the device performs operations with some unknown key, a portion or even all of this key might be disclosed. Well-known examples of such physical values are execution time, power consumption, and electromagnetic emanation [9]. An especially powerful attack is differential power analysis (DPA), first published by Kocher et al. [8], which employs statistical methods to extract sensitive data even from very noisy measurements. Our work centers around methods to increase resistance of an AES implementation against DPA.

Differential power analysis targets intermediate values which depend on a small portion of the key and the plaintext or ciphertext. We denote such vulnerable intermediate results as *critical data* and manipulations of such values as *critical operations*. For secret-key algorithms, critical data is commonly processed at the beginning and at the end of the algorithm, where the intermediate results are strongly related to the plaintext or ciphertext. In order to defend against DPA, we first need to analyze the flow of critical data through the processor and its potential impact on the overall power consumption.

Figure 2 depicts a typical datapath of a RISC processor augmented with a custom functional unit. This example shows four pipeline stages: the decode stage between the *register file* and the *op1*, *op2* registers, the execute stage between *op1*, *op2* and the *result* register, the memory stage between *result* and *wr.result* register, and the write stage between *wr.result* and the *register file*. There are several feedback paths from different stages back into the decode and execute stage. The memory stage is connected to the data cache. Also depicted are inputs and outputs for the control flow (*ex.PC*, *jmpl address*, *wr.PC*). The functional unit for the instruction set extensions (ISE FU) is located in parallel to the original ALU/Shifter unit.

In order to analyze the side-channel leakage of this datapath, we assume that the *reg1* bus carries critical data, which can be subjected to a side-channel attack. The parts of the datapath marked by bold lines and gray shades process or hold values that are related to the original value of *reg1*. It is important to note that the critical data passes through both the ALU/Shifter as well as the ISE FU, irrespective of the result selected by the following multiplexor. Also

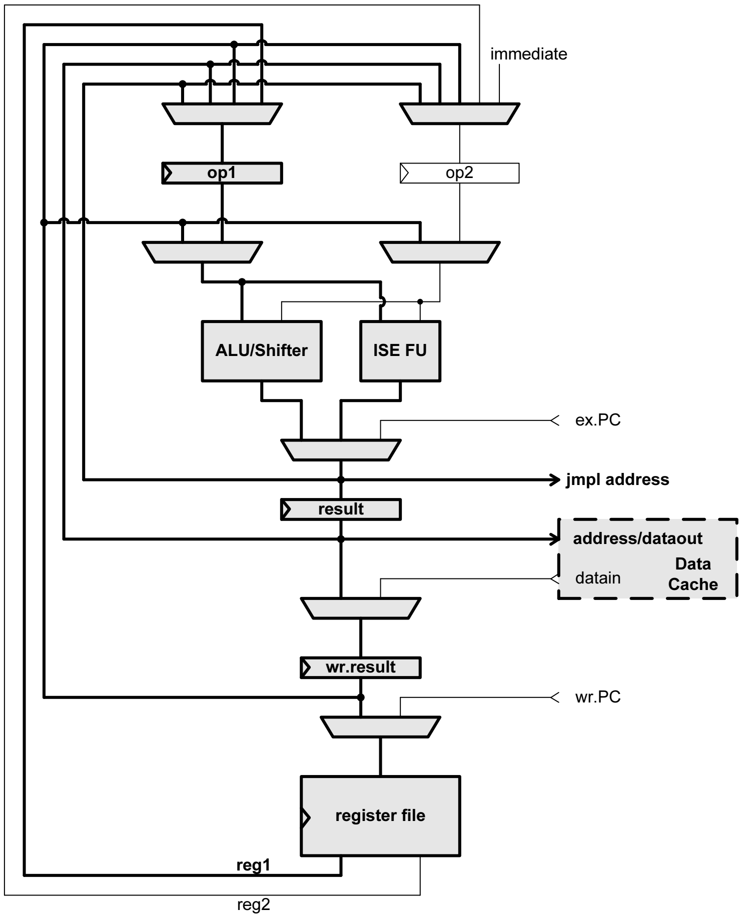

Fig. 2. Typical datapath of a processor with instruction set extensions

note that all feedback paths, the *jmpl address*, as well as the data cache input carry critical values.

An analysis of AES software using the instruction set extensions has revealed that almost all instructions processing critical data can be easily rearranged into such an order so that no feedback path needs to be used (i.e. no forwarding occurs). Moreover, critical data is not used for control transfer and is never stored to memory. Therefore, our first step to improve implementation security is to detach these paths from the ISE datapath. The resulting, slightly modified datapath is shown in Figure 3.

Two important changes can be seen in this figure. Firstly, operands for the functional units (*op1* and *op2*) can be blocked from entering the ALU/Shifter whenever the ISE FU is active. Moreover, additional multiplexors in the feedback and output paths allow to prevent propagation of critical data. All multiplexors

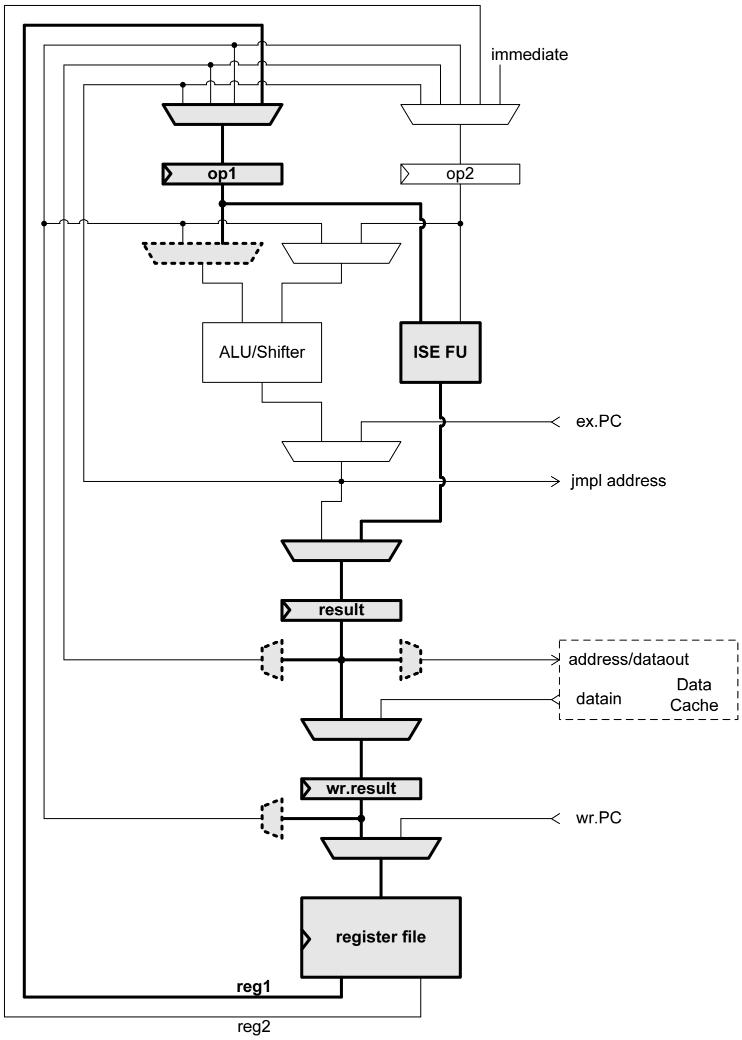

Fig. 3. Modified processor datapath to suppress propagation of critical values

which suppress the propagation of the critical value related to *reg1* are marked with dashed lines. Through this simple measure the portion of the datapath carrying critical data and hence requiring side-channel analysis countermeasures is reduced significantly in comparison to the datapath in Figure 2.

3.1 Proposed Countermeasures

Based on the analysis of the critical sections of a typical processor datapath, we examine three different strategies to increase the resistance against side-channel attacks in the next sections. These three solutions differ with respect to the ease of implementation, level of security, and impact on the critical path, area, and power consumption, which we compare in Section 7.

4 Option 1: Complete Datapath in Secure Logic

The straightforward way to secure the processor containing the extended datapath shown in Figure 3 is to implement all critical parts in a DPA-resistant logic style. For example, the circuit could be built using Wave Dynamic Differential Logic (WDDL), as proposed by Tiri and Verbauwhede in [17]. This solution is applicable to secure all types of cryptographic instruction set extensions, provided that the critical values do not leave the secured datapath, e.g. are stored to unprotected cache or memory.

Note that it is also necessary to include the register file in the secure implementation. As modern embedded RISC processors can have a large number of registers, securing the register file can become very costly in terms of silicon area. For example, SPARC V8 processors feature a number of so-called register windows with 24 individual and 8 shared 32-bit registers. To limit the overhead in terms of area, it is possible to implement only a part of the registers in a secure logic style, and restrict critical values to these registers.

5 Option 2: Random Precharging

Another measure to increase resistance against power analysis is to randomly charge the datapath before (precharge) and after (postcharge) a critical value is processed. This measure can be implemented solely by careful modification of the cryptographic software, but delivers at best a moderate increase in security. A similar countermeasure was proposed in [3] for hardware implementations.

Random precharging can be realized in software by prefixing and suffixing each instruction that processes critical data with the same instruction using random operand values. As the same instruction is used, the same paths will be active and loaded with random values. We will show in this section that for the critical instructions for AES, the result of the execute stage is a uniformly distributed random value if both operands are also uniformly distributed. This property is necessary to ensure that all vulnerable paths after the functional unit are also charged with uniformly distributed random values.

The random charging of the datapath before and after critical instructions can, of course, not completely prevent side-channel leakage. Nevertheless, the switching activity for the critical values is randomized to a certain degree. When the preceding value of a bit line is randomized, then the transition characteristics with respect to a fixed second value will also be randomized. When the preceding value of a critical value was fixed to 0 and becomes uniformly distributed, the switching characteristics change accordingly:

- $0 \rightarrow 0$ transitions become $0 \rightarrow 0$ or $1 \rightarrow 0$ transitions
- $0 \rightarrow 1$ transitions become $0 \rightarrow 1$ or $1 \rightarrow 1$ transitions

A similar situation occurs when a preceding value fixed to 1 gets randomized:

– $1 \rightarrow 0$ transitions become $1 \rightarrow 0$ or $0 \rightarrow 0$ transitions
– $1 \rightarrow 1$ transitions become $1 \rightarrow 1$ or $0 \rightarrow 0$ transitions

Without loss of generality, we can limit our analysis to the first case. We can observe that a 0 for the second (i.e. the critical) value will result in a $0 \rightarrow 0$ or $1 \rightarrow 0$ transition with equal probability. If the critical value is 1, then a $0 \rightarrow 1$ or $1 \rightarrow 1$ transition can occur. An attacker who correctly predicts a 0 or 1 of the critical value will have to distinguish the effects of $0 \rightarrow 0$ and $1 \rightarrow 0$ transitions from $0 \rightarrow 1$ and $1 \rightarrow 1$ transitions. When no random precharging is employed, it is sufficient to distinguish $0 \rightarrow 0$ from $0 \rightarrow 1$ transitions, which is generally much easier. By the same argument, an attacker has to distinguish $0 \rightarrow 0$ and $0 \rightarrow 1$ transitions from $1 \rightarrow 0$ and $1 \rightarrow 1$ transitions for an instruction with random operands occurring after the attacked instruction.

We will now examine the case for AES instruction set extensions in more depth. As already noted in Section 2, we use the high-performance extensions proposed in [14], namely the instruction families `sbox4s/isbox4s/sbox4r` and `mixcol4s/imixcol4s`. The `sbox4s` instruction takes two bytes each from the first and second operand and substitutes them according to the AES S-box. The `isbox4s` instruction does the exactly same with the inverse S-box. The `sbox4r` instruction substitutes the four bytes of the first operand, followed by a rotation to the left of the result by one byte. For `mixcol4s`, an AES State column is assembled from two bytes of each of the operands. This column is transformed according to the AES MixColumns operation, yielding the corresponding output column of the AES State. The `imixcol4s` instruction does exactly the same for the AES InvMixColumns transformation.

These two families of instructions are almost sufficient to implement manipulations of critical data for the whole AES algorithm. The only additional instruction required is the `xor` instruction.

One important point to show is that it suffices to execute a custom instruction with two random operands to charge every node in the extended datapath to a uniformly distributed random value. To this end we have to examine the result for each extended instruction with random operands:

– `sbox4s/isbox4s/sbox4r`: The input to the ISE functional unit is assembled and calculated from independent bytes of the two operands. The AES S-box itself is a bijective mapping. Therefore, two uniformly distributed random operands produce a uniformly distributed result.
– `mixcol4s/imixcol4s`: The input to the ISE FU is again assembled from independent bytes of the operands. For each byte of the resulting output column, the four bytes of the input column are linearly transformed (multiplied by a constant in $GF(2^8)$), and then added (xored) together. Both does not change the uniform distribution of the input operands.
– `xor`: A bitwise exclusive or of two uniformly distributed operands, yields a uniformly distributed result.

5.1 Further Implementation Details

It is important to know which instructions process critical data with respect to power analysis and therefore need to be precharged. Generally, these are all instructions which process data that depends on the plaintext or ciphertext and a small portion of one or more round keys. We consider the initial AddRoundKey, all four transformations of the first round, and SubBytes of the second round as potentially vulnerable. At the end of the AES algorithm, all transformations of round nine and ten need to be protected. The intermediate values in the middle rounds depend on five or more bytes of the cipher key, thus requiring at least 2^{40} hypotheses for a DPA attack. We consider this level of protection to be sufficient for most applications. However, it is not difficult to extend the protection to further rounds.

As we have already noted in Section 3, we want to avoid the use of feedback paths for critical data whenever possible. In our example of the 4-stage data pipeline, we can prevent the propagation of critical values via feedback paths by inserting at least two unrelated instructions between the instruction generating the critical value and the first instruction using it as an operand. In the case of the examined AES instructions set extensions, this criterion can be fulfilled with no or little performance loss, by rearranging the instructions in an assembly implementation.

Another potential way to circumvent the protection of software random precharging is to artificially generate traps at specific moments in time. If the corresponding trap handler routine is executed between the random precharge instruction and the actual instruction which manipulates critical data, then the effect of random precharging can be undone. To prevent this, interrupts should be disabled during the critical sections of the AES algorithm. These sections typically require only a few dozen cycles, so this measure should not have a detrimental impact on system responsiveness. Note that in principle it would be sufficient to disable all interrupts which are under the user's control (e.g. UART and timer interrupts).

For the examined AES extensions and the previously outlined degree of protection, it is sufficient to protect 20 instructions at the beginning and 20 instructions at the end of the algorithm. With the postcharge of the last instruction of each sequence and two random 32-bit operands per random charge we need $2*(2*(20+1)) = 84$ random 32-bit words (i.e. 336 random bytes). Depending on the implementation of the random number generator, they could be generated on-the-fly or loaded from a pre-generated array of random values in memory.

6 Option 3: Protected Mask Unit

In this section we present a generic method for securing cryptographic instruction set extensions. The basic idea is to split the processor into an insecure and a (relatively small) secure zone. Critical data in the insecure zone is always protected with a Boolean mask. The secure zone contains storage for the respective mask, a mask generator and the functional units for the extensions. The secure zone is implemented in a secure logic style.

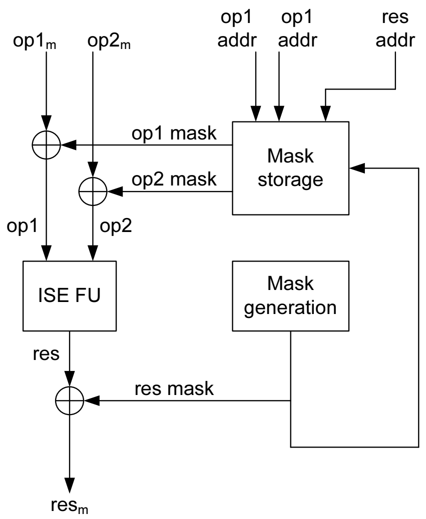

Fig. 4. Secure zone of the processor

Figure 4 shows the principal structure of the secure zone. The register addresses of the masked operands $op1_m$ and $op2_m$ select the corresponding masks from the storage section. Then the operands are unmasked and processed by the functional unit, yielding the result res. The generator produces a new random mask and applies it to the result res before it leaves the secure zone. This new mask is also saved in the mask storage for the register which receives res_m. Outside of the secure zone, the critical values remain masked and can therefore be handled without restrictions, e.g. stored to cache and memory. The insecure zone does not require any special architectural modifications or a more careful implementation in comparison with the corresponding part of the unprotected processor.

This countermeasure confines the processing of the unmasked critical values almost exclusively to the secure zone, as these values retain their Boolean mask in the insecure zone. An important exception from this rule is the exclusive-or operation of a masked value (u_m) with an unmasked value (v), which does not change the mask: $u_m \oplus v = u \oplus m \oplus v = (u \oplus v) \oplus m = (u \oplus v)_m$. The AddRoundKey operation of AES is an example where the State is the masked value and the corresponding round key is the unmasked value. Note that the round key is not dependent on the plaintext or ciphertext and is therefore not directly vulnerable to a DPA attack. Template attacks which target the round key have been shown to be infeasible on 32-bit processors (see [15] for further details).

In the case of the AES extensions, it is sufficient to include the functional units for the `sbox4s` and `mixcol4s` instruction families in the secure zone. The critical values can be confined to seven 32-bit registers during execution of the AES algorithm, so the mask storage unit only needs to be able to store seven 32-bit masks. SubBytes, ShiftRows, and MixColumns would then be performed unmasked in the secure zone, while AddRoundKey could be done using the masked State in the insecure zone.

The registers whose masks are held in the mask storage could either be fixed or dynamically changed with the help of a tiny content-addressable memory (CAM), similar to a cache. A mechanism for masking plaintext and unmasking ciphertext is also required, which could be done, for example, with a dedicated instruction.

The cycle count for AES operations would remain almost the same for a protection solution with a mask unit (requiring just a few additional cycles for masking at the beginning and unmasking at the end). The architecture of the ISE FU can also remain unchanged since the critical values are processed unmasked. A drawback of the solution with the mask unit is that it introduces a resource (namely the mask storage unit) which can only be used by a single task. To share this unit between tasks, its content needs to be saved on a task switch, preferably in memory. In this way, the masks would leave the secure zone and open up a potential vulnerability against second-order DPA attacks, as now both the masked value and the corresponding mask would appear in the insecure zone. However, it might still be possible to save the contents of the mask storage into the unprotected memory on a task switch, without impairing the overall security. If an attacker cannot control or predict task switches, then it would be very hard to observe leakage related to specific critical values. Such functionality could be implemented both in the processor's hardware or in the operating system.

7 Security and Performance Analysis

In this section we compare the three proposed solutions with respect to a number of implementation aspects, whereby we try to keep the scope as broad as possible. However, for some metrics it is necessary to take certain design choices into account. We have chosen to use WDDL [16, 17] as an example of a secure logic style. All figures for silicon area, critical path, and power consumption are given for an implementation based on a $0.13\,\mu$m standard-cell library. The cycle count refers to an AES-128 encryption using instruction set extensions.

7.1 Applicability and Implementation Complexity

Implementing the complete datapath in secure logic (option 1) can be applied to any kind of instruction set extension. It requires a careful partitioning of the processor into an insecure and a secure part and the implementation of the latter in a secure logic style. The software only needs to be modified to restrict critical operations to the secure datapath. Random precharging (option 2) is also generally applicable to cryptographic instruction set extensions. Being a software countermeasure, it can be flexibly adapted to protect specific operations which are deemed vulnerable. The hardware can be left unchanged. The solution with a protected mask unit (option 3) is only applicable if almost all critical operations can be limited to the secure zone. Its implementation is easier than option 1, as the boundary between insecure and secure zone is simple and well-defined. Changes to the software are not necessary, apart from explicit masking and unmasking operations at the beginning and end of the crypto operation.

7.2 Security

We express security as the factor by which the number of power traces necessary for mounting a successful DPA attack increases. For the solutions employing a secure logic style (options 1 and 3), this factor solely depends on the security of the chosen logic style. For a careful implementation of WDDL in an ASIC, an empirical evaluation has shown this factor to be over 750 [16].

In order to assess the security of random precharging (option 2), we have compared an unprotected and a protected AES software implementation. We have prototyped the Leon2-CIS processor with AES extensions on an FPGA-board designed for power measurements [12]. Then, we have performed a DPA attack on both AES software implementations using 100,000 power traces. Although the absolute results may differ from those of an ASIC implementation, the relative comparison gives an indicator for the security improvement. Figures 5 and 6 show the result for the DPA attack on the unprotected and the protected AES implementation, respectively.

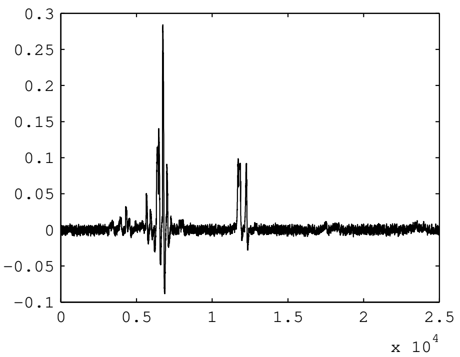

Fig. 5. Result of DPA attack on unprotected AES implementation

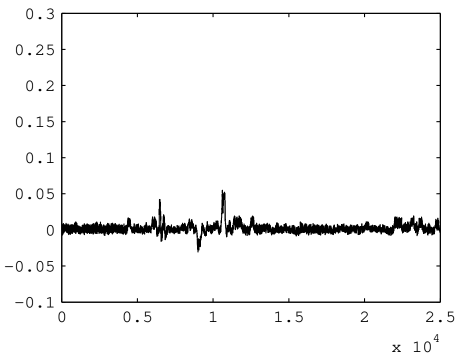

Fig. 6. Result of DPA attack on AES implementation with random precharging

The empirical results indicate that the correlation coefficient ρ can be reduced significantly from 0.284 to 0.055. Therefore, the maximum correlation is reduced by a factor of 5.16, which corresponds to an increase of the number of required power traces by a factor of over 26 [9].

Table 1. Cycle count for AES-128 encryption

Implementation	**Cycle count**	**Overhead**
Baseline implementation [14]	196	0%
Complete datapath in secure logic	392	100%
Random precharging	~ 400	$\sim 105\%$
Protected mask unit	~ 230	$\sim 17\%$

7.3 Performance

Table 1 shows how the different solutions impact the performance of a single AES-128 encryption. Using a secure logic style such as WDDL requires an extra precharge cycle for every operation performed in the secure part. This concerns every instruction when the complete datapath is in secure logic, and therefore leads to a doubling of the cycle count. For the protected mask unit, we assume that only the critical instructions—which constitute only a small fraction of the total instructions—require an extra precharge cycle for the secure logic. Random precharging in software leads to a higher overhead as it needs one precharge and one postcharge instruction per critical instruction, and also extra instructions to load or generate the random precharge values.

In summary, the cycle count for our protected implementations ranges approximately between 230 and 400 cycles for a single 128-bit AES encryption. An unprotected implementation on a processor without instruction set extensions requires about 1,637 cycles [14]. When compared to the overhead of the software countermeasures introduced in [15], which ranges between a factor of ten and 100, our proposed solutions remain very fast.

7.4 Implementation Overhead

For the random precharging solution, the hardware—and therefore also silicon area, critical path and power consumption—remain unchanged. For the other two options, the employed secure logic style plays a crucial role in determining the overhead for these factors. For standard-cell implementation using WDDL, the area typically increases by a factor of about 3.2–3.6. The increase of the critical path has been shown to range between 1.07–1.42 while the power consumption increases by a factor of about 3.5 [17]. For this evaluation we will use a factor of 3.5 for area, 1.2 for critical path, and 3.5 for power consumption. In our case, only a fraction of the total processor needs to be implemented in the secure logic style, which alleviates the total overhead factors.

We now try to give a rough estimation of this overhead for the Leon2-CIS processor used in [14]. The extended integer unit was reported to have a size of 16,370 gate equivalents (GEs) at a delay of 4 ns (UMC 0.13 μm technology). About half of the integer unit will have to be implemented in the secure logic style for option 1, requiring about $16,370 * (1/2) * 2.5 \approx 20,500$ GEs extra (note that if the size increases by 3.5, the extra area is $(3.5 - 1) = 2.5$ times the original area). Additionally, the register file also needs to be implemented securely. This

Table 2. Estimation of overhead for silicon area and critical path

Implementation	**Silicon Area**[1]	**Critical Path**
Complete datapath in secure logic	$+(20,500 + 940R)$ GEs	+0.8 ns
Random precharging	none	none
Protected mask unit	+28,000 GEs	+1.0 ns

will lead to a considerable increase in silicon area due to two reasons: firstly, it will be necessary to change the register file implementation from a custom memory block to flip-flops. Secondly, if WDDL flip-flops are used, the area for a single flip-flop is increased by at least a factor of 4 [17]. The size of a 32-bit CMOS register is about 235 GEs, and therefore a 32-bit WDDL register costs about 940 GEs. Depending on the number R of secured registers, we get an area of about $940R$ GEs for the secure register file. For example, a complete register window of a SPARC V8 processor consisting of 32 registers would cost about 30,000 GEs for secure implementation. This illustrates that option 1 is very costly in terms of silicon area. Taking our estimated overhead factor of 1.2, the critical path will increase to 4.8 ns.

For the protected mask unit, the secure part of the circuit mainly consists of three parts: the ISE functional unit (FU), the mask storage, and the mask generator. The area of the ISE FU is about 3,000 GEs [14]. The mask storage includes seven 32-bit registers, which requires about 1,700 GEs. For the mask generator, we estimate the size of a radix-32 version of the key generator of the Trivium stream cipher, based on the figures reported in [6]. Such a circuit would require about 5,500 GEs and will be able to deliver 32 fresh pseudo-random bits per cycle. Taking into account circa 1,000 GEs for additional circuitry (e.g. xor gates, etc.), the total overhead would be $(3,000 + 1,700 + 5,500 + 1,000) * 2.5 = 28,000$ GEs. The critical path through the ISE FU would also be slightly increased by the xor stages and the decode logic in the mask storage unit (see Figure 4), so that it can be estimated to about 5 ns when compared to option 1. Table 2 summarizes the overhead estimations. It should be noted that these figures only give a rough idea of the magnitude of overhead.

7.5 Combination with Other Countermeasures

In [15], software countermeasures are proposed to secure AES implementations using instruction set extensions. Since implementing the complete datapath in secure logic (option 1) is an orthogonal solution, both countermeasures could be directly combined to increase implementation security further. Options 2 and 3 can be combined with the measures of [15] to a certain degree (e.g. randomization of operations), which also leads to a further increase in resistance against power analysis attacks.

[1] R denotes the number of secured 32-bit registers.

8 Conclusions

In this paper we have investigated three different approaches for increasing the DPA-resistance of AES software implementations executed on a 32-bit RISC processor with cryptography extensions. The first approach is to implement all security-critical parts of the processor datapath—including (parts of) the register file and functional units—using a DPA-resistant logic style. While this approach does not degrade performance, it comes at a high cost in terms of area since a significant portion of the processor has to be implemented in DPA-resistant logic. The security of this approach depends primarily on the security of the used logic style.

The second approach is a software countermeasure, which uses random precharging at the instruction level. A major advantage of this approach is that it does not require any modifications of the processor hardware. However, the achievable security gain is rather moderate. Furthermore, random precharging also increases execution time, but thanks to the instruction set extensions the performance of the protected implementation is still significantly higher than that of an unprotected implementation on a conventional SPARC V8 processor without cryptography extensions (400 cycles vs. 1,637 cycles [14]).

The third approach involves both hardware as well as software and uses a secure mask unit. This approach splits the processor into a secure zone and an insecure zone, whereby the secure zone—containing storage for the mask, a mask generator, and the functional units for the extensions—is protected with a DPA-resistant logic style. The hardware cost of this approach has been estimated to be 28,000 GEs, which is significantly less than the hardware cost of the first approach (even if assuming only a single protected register window). Similar to the first approach, the exact security and additional area depends on the properties of the used DPA-resistant logic style. The benefit of this approach is that it has only a minimal impact on performance, and that it does not impose any restrictions on the operations outside of the secure zone.

In summary, the side-channel countermeasures discussed in this paper enable different tradeoffs between performance, hardware cost and security, which are not achievable with pure software or pure hardware implementations.

Acknowledgements. The authors would like to thank Stefan Mangard and Dan Page for their help in writing this paper. The research described in this paper has been supported by the Austrian Science Fund (FWF) under grant number P18321-N15 ("Investigation of Side-Channel Attacks") and P16952-N04 ("Instruction Set Extensions for Public-Key Cryptography"), by the European Commission under grant number FP6-IST-033563 (Project SMEPP) and, in part, by the European Commission through the IST Programme under contract IST-2002-507932 ECRYPT.

The information in this paper reflects only the authors' views, is provided as is and no guarantee 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. G. Bertoni, L. Breveglieri, R. Farina, and F. Regazzoni. Speeding Up AES By Extending a 32-Bit Processor Instruction Set. In *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 275–282. IEEE Computer Society, September 2006.
2. J. Blömer, J. Guajardo, and V. Krummel. Provably Secure Masking of AES. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2005.
3. M. Bucci, M. Guglielmo, R. Luzzi, and A. Trifiletti. A Power Consumption Randomization Countermeasure for DPA-Resistant Cryptographic Processors. In E. Macii, O. G. Koufopavlou, and V. Paliouras, editors, *14th International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation, PATMOS 2004, Santorini, Greece, September 15-17, 2004, Proceedings*, volume 3254 of *Lecture Notes in Computer Science*, pages 481–490. Springer, 2004.
4. A. J. Elbirt. Fast and Efficient Implementation of AES via Instruction Set Extensions. In *21st International Conference on Advanced Information Networking and Applications (AINA 2007), Workshops Proceedings, Volume 1, May 21-23, 2007, Niagara Falls, Canada*, pages 481–490. IEEE Computer Society, 2007.
5. J. Großschädl and E. Savaş. Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2004.
6. F. K. Gürkaynak, P. Luethi, N. Bernold, R. Blattmann, V. Goode, M. Marghitola, H. Kaeslin, N. Felber, and W. Fichtner. Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt. Record of The State of the Art of Stream Ciphers (SASC) Workshop 2006, February 2006.
7. ISEC project. Instruction Set Extensions for Cryptography (ISEC): Project Webpage. `http://www.iaik.tugraz.at/isec`.
8. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
9. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9.
10. K. Nadehara, M. Ikekawa, and I. Kuroda. Extended Instructions for the AES Cryptography and their Efficient Implementation. In *IEEE Workshop on Signal Processing Systems (SIPS'04)*, pages 152–157, Austin, Texas, USA, October 2004. IEEE Press.
11. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.
12. K. Schgaguler. Assay of the DPA Vulnerability of Micro Electric Circuits Based on FPGA Measurements. Master's thesis, Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria, October 2005.

13. Z. Shi and R. B. Lee. Bit Permutation Instructions for Accelerating Software Cryptography. In Earl E. Swartzlander, Graham A. Jullien, and Michael J. Schulte, editors, *12th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2000), 10-12 July 2000, Boston, MA, USA*, pages 138–148. IEEE Computer Society, 2000.
14. S. Tillich and J. Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2006.
15. S. Tillich, C. Herbst, and S. Mangard. Protecting AES Software Implementations on 32-bit Processors against Power Analysis. In *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS'07)*, Lecture Notes In Computer Science, 2007. To be published.
16. K. Tiri, D. Hwang, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. Prototype IC with WDDL and Differential Routing - DPA Resistance Assessment. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 354–365. Springer, 2005.
17. K. Tiri and I. Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, volume 1, pages 246–251. IEEE Computer Society, 2004.