Enhancing an Embedded Processor Core with a Cryptographic Unit for Performance and Security*

Övünç Kocabaş, Erkay Savaş
Sabanci University
Orhanli, Tuzla, TR–34956 Istanbul, Turkey
E-mail: ovunc@su.sabanciuniv.edu, erkays@sabanciuniv.edu

Johann Großschädl
University of Bristol
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom
E-mail: johann@cs.bris.ac.uk

Abstract

We present a set of low-cost architectural enhancements to accelerate the execution of certain arithmetic operations common in cryptographic applications on an extensible embedded processor core. The proposed enhancements are generic in the sense that they can be beneficially applied in almost any RISC processor. We implemented the enhancements in form of a cryptographic unit (CU) that offers the programmer an extended instruction set. The CU features a 128-bit wide register file and datapath, which enables it to process 128-bit words and perform 128-bit loads/stores. We analyze the speed-up factors for some arithmetic operations and public-key cryptographic algorithms obtained through these enhancements. In addition, we evaluate the hardware overhead (i.e. silicon area) of integrating the CU into an embedded RISC processor. Our experimental results show that the proposed architectural enhancements allow for a significant performance gain for both RSA and ECC at the expense of an acceptable increase in silicon area. We also demonstrate that the proposed enhancements facilitate the protection of cryptographic algorithms against certain types of side-channel attacks and present an AES implementation hardened against cache-based attacks as a case study.

1. Introduction

Optimizing the resources of general-purpose processors to fit the needs of a specific application or application domain is nowadays widely employed in embedded systems [9]. Many microprocessor vendors developed architectural enhancements for fast multimedia processing (e.g. Intel's MMX, AMD's 3DNow, or Motorola's AltiVec). Similar to multimedia applications, also public-key cryptosystems are suitable for processor specialization since most software algorithms for multiple-precision arithmetic employed in public-key cryptography spend the overwhelming majority of their running time in a few performance-critical sections (e.g. inner loops). Speeding up these critical code sections through architectural enhancements can therefore result in a significant performance gain.

In this paper, we explore the benefits of architectural enhancements for fast and secure computation of cryptographic operations on an embedded RISC processor. Such enhancements are typically realized by 1) augmenting the existing base-ISA with new instructions and 2) adding new functional units with reasonable overheads. Extending a general-purpose processor with a set of custom instructions for operations that dominate cryptographic applications in terms of execution time and resource usage has a number of advantages over using a hardware accelerator such as a cryptographic co-processor. First, performing cryptographic operations within a processor core eliminates the communication overhead (and possibly associated security risks) accrued in processor/co-processor settings. Second, the area of a cryptographic co-processor is usually larger than the area overhead of a functional unit that is tightly coupled to the processor core and directly controlled by the instruction stream. Third, architectural enhancements offer a degree of flexibility and scalability that goes far beyond of what is possible with fixed-function hardware (i.e. co-processors) since the base instruction set can be used to implement the control flow of any cryptographic algorithm for operands of arbitrary length, while the domain-specific instructions allow one to speed up the performance-critical operations.

In practice, a number of criteria have to be considered when designing architectural enhancements for embedded processors. Most notably, the enhancements should not entail 1) an unacceptable increase in area, 2) a change in instruction format or size, 3) a difficult integration into the available tool-chain, and 4) a major change in the control circuitry and pipeline structure.

*This work was supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project number 105E089 (TUBITAK Career Award) and, in part, by the EPSRC under grant EP/E001556/1.

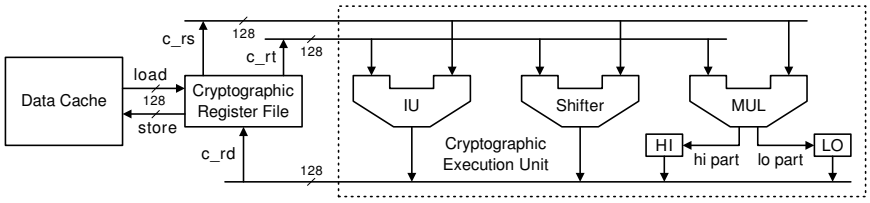

Figure 1. Detailed Architecture of the Cryptographic Unit (CU)

1.1. Related Work and Our Contributions

Various architectural enhancements to speed up the execution of cryptographic operations have been proposed in the recent past [6, 7, 8, 16, 17]. For example, the authors of [7] propose a set of five custom instructions to accelerate arithmetic operations in GF(p) and GF(2^m) on a MIPS32 core to benefit elliptic curve cryptography, while the ISA extensions in [17] aim to support pairing-based cryptosystems. On the other hand, the authors of [6] explore the effect of on-chip memory on the execution time of S-box computations in symmetric-key cryptography. A common characteristic of all these approaches is that they focus on architectural enhancements for accelerating an individual cryptographic operation.

In this paper, we take a slightly different and holistic approach by designing, implementing, and integrating a *cryptographic unit (CU)* into an extensible embedded processor core with the goal to support many cryptographic operations through not only acceleration but also secure execution. The CU provides a set of new and powerful custom instructions to speed up multiplication and inversion in prime fields, as well as other operations carried out in elliptic curve cryptography and RSA. The CU is also shown to be beneficial for the implementation of AES software hardened against certain types of side-channel attacks. Our experiments with an extensible processor platform from Tensilica [15] demonstrate that the proposed CU can be easily integrated into a general-purpose RISC core with an acceptable hardware overhead.

2. General Architecture

The cryptographic unit, shown in Figure 1, consists of two parts: i) a *cryptographic register file (CRF)* organized as an array of 32 registers, each 128 bits wide, and ii) a *cryptographic execution unit (CEU)*. The CRF can be used to store operands as well as temporary results of arithmetic operations with the purpose of increasing the performance since a register file of such size considerably reduces the number of memory accesses. It can also be used to store sensitive information (e.g. secret keys) and small look-up tables for security and speed. The CEU can be considered

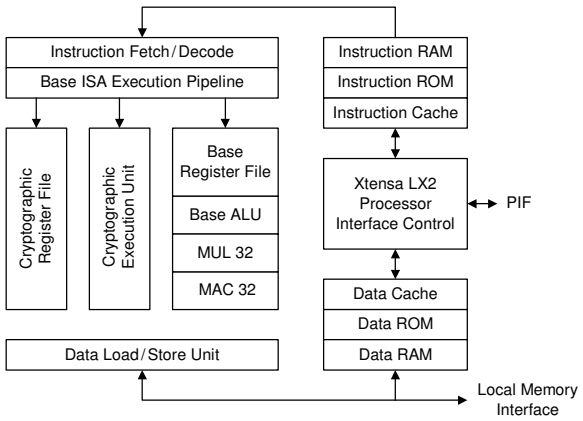

Figure 2. Architecture of the Enhanced Core

as cryptographic data path containing functional units that are better capable of dealing with relatively large operands than a classical integer unit. Figure 2 shows that the CU is tightly integrated into the processor core and utilizes its pipeline structure and interconnection infrastructure. Both the instructions of the base-ISA and the new instructions are executed in the existing pipeline.

Figure 1 shows the functional units inside the CU, where data transfer and arithmetic/logic operations are performed on 128-bit words. The integer unit (IU) is capable of adding and subtracting two 128-bit integers, while the shifter can shift a 128-bit register in both directions. The shifter is also able to shift the least or most significant bit from one register to another by taking two registers as argument (but only one register is overwritten; see Table 1). The third functional unit is a 128-bit multiplier which takes two 128-bit integers as input and produces a 256-bit result. As can be seen from the figure, the data path and interconnection structure for cryptographic instruction execution is similar to the integer data path of a typical RISC processor.

2.1. Multiplier Unit

The most important functional unit with respect to the performance of public-key cryptographic operations is the multiplier unit, which needs to be carefully designed and integrated into the micro-architecture. Our design is based

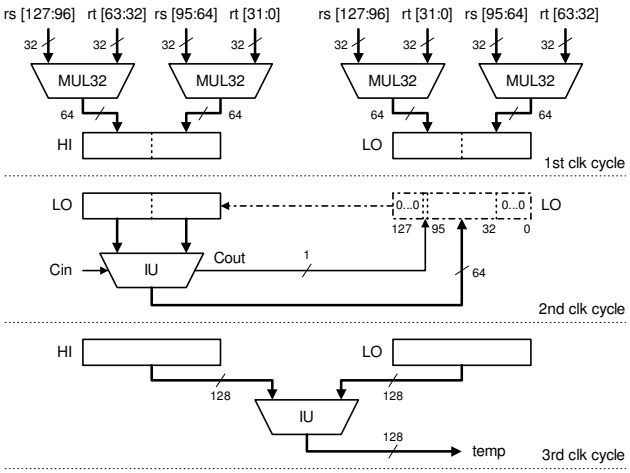

Figure 3. Multiplier Unit

on four 32-bit multipliers that are instantiated from the existing multiplier unit in the integer data path. The result of the multiplication is a 256-bit integer, which is produced in 15 clock cycles to avoid an increase of the critical path delay. A full 128-bit multiplication consists of four 64-bit multiplications, each of which itself consists of four 32-bit multiplications. The computation of a 64-bit multiplication is shown in Figure 3; it completes in three clock cycles. In the first cycle, four 32-bit multipliers generate the partial products and write them into the 256-bit HI/LO register pair. In the next clock cycle, an addition is performed on the aligned partial products; finally in the last cycle all the partial products are added up and the result is placed in a temporary register. This process is repeated three times for the other 64-bit multiplications. A final addition phase is needed to generate the 256-bit result, which consumes another three clock cycles.

2.2. Cryptographic Register File

The CRF can be shared among different processes if the operating system supports multi-tasking. However, in order to alleviate security concerns and reduce the cost of context switches, we propose a *transactional* use of the CRF. The content of the CRF is not saved by the operating system on context switches; therefore, any process wishing to use the CRF does not automatically assume that a register's content remains intact forever. Each process is provided with a consistent view of the CRF for only a short duration (e.g. for the time of one multiple-precision multiplication). A process can lock the CRF for this duration (so that no other process can access the CRF) if the context switches occur too frequently. The operating system assists processes for a fair schedule of the CRF usage in order to prevent starvation or attacks by malicious processes. Therefore, a smart scheduling algorithm is very important to solve the afore-mentioned problems.

Format	Description
ADD_CREG(c_rd, c_rs, c_rt)	$c\_rd \leftarrow c\_rs + c\_rt$
SUB_CREG(c_rd, c_rs, c_rt)	$c\_rd \leftarrow c\_rs - c\_rt$
COMP_CREG(c_rd, c_rs, c_rt)	$c\_rd \leftarrow c\_rs > c\_rt\ ?\ 1 : 0$
SHL_CREG(c_rs, c_rt)	$c\_rs \leftarrow c\_rs_{126:0} \| \ c\_rt_{127}$
SHR_CREG(c_rs, c_rt)	$c\_rs \leftarrow c\_rt_{0} \| \ c\_rt_{127:1}$
MUL_CREG(c_rs, c_rs)	$(HI/LO) \leftarrow c\_rs \times c\_rt$
LOAD_CREG(c_rd, address)	$c\_rd \leftarrow$ Memory[address]
STORE_CREG(c_rd, address)	Memory[address] $\leftarrow c\_rd$

Table 1. New 128-bit Instructions

2.3. Custom Instructions

Table 1 summarizes the custom instructions added to the base-ISA. Note that the new instructions operate on 128-bit values and conform to the instruction types and formats of conventional RISC architectures.

3. Software Implementation of Arithmetic Operations on the Enhanced Processor

In the following, we explain the implementation of two important arithmetic operations on our enhanced embedded processor, namely multiple-precision modular multiplication and modular inversion. Our processor is based on the Xtensa LX2 [15], a configurable and extensible RISC core developed by Tensilica. We use the term *base processor* to denote the Xtensa LX2 with its base instruction set, 8 kB of separate instruction and data cache memories, and a 32-bit multiplier. We enhanced the base processor with the tightly-coupled CU described in the previous section and refer to it as *enhanced processor*. All algorithms of which we report the execution times were executed on both the base processor and the enhanced processor.

3.1. Modular Multiplication

Koç et al. [11] proposed five algorithms to implement the Montgomery modular multiplication [12] (which is the fastest method for modular multiplication) in software. The CIOS method [11] seems to be the best choice since it has a regular execution pattern and requires the least memory space among the examined candidates. However, we found the SOS method [11] more suitable for execution on our enhanced processor. The SOS method is performed in two steps: i) the schoolbook multiplication of two big integers and ii) the Montgomery reduction. Even though the SOS method doubles the memory space, it does not use the all the variables at the same time. Since each step in isolation

Precision	CIOS (base)	SOS (enhanced)	Speed-up
160	2,765	1,047	2.6
192	3,873	1,196	3.2
256	6,691	931	7.2
512	25,605	2,365	10.8
1024	100,304	7,654	13.1

Table 2. Timings for Modular Multiplication

Precision	Inv. (base)	Inv. (enhanced)	Speed-up
160	78,174	36,978	2.11
192	106,082	43,864	2.42
256	172,407	57,168	3.02
512	579,878	141,836	4.09

Table 3. Timings for Modular Inversion

requires less memory than the CIOS method (which executes the two steps interleaved), all the operands needed in the two SOS steps fit into the CRF if the operands are no longer than 1024 bits. Table 2 summarizes the execution times (in clock cycles) of the CIOS method on the base processor and the SOS method on the enhanced processor.

3.2. Modular Inversion

Inversion is an extremely slow operation needed in both RSA (e.g. for key generation or CRT method) and elliptic curve cryptography (ECC). While it is possible to avoid the inversion operation in many cases, there will be other situations where fast inversion is useful. The best way to compute multiplicative inversion is to use what is known as the binary extended Euclidean algorithm and a variation of it, the Montgomery inversion algorithm [10]. We decided to implement the Montgomery inversion on both the base processor and the enhanced processor; the execution times (in clock cycles) are summarized in Table 3.

4. Implementation Results

In this section, we present the estimated speed-up values obtained for both RSA and elliptic curve cryptography. We implemented a 1024-bit RSA using two different exponentiation techniques: a window method with a 4-bit window size and the the conventional binary method (the so-called "square and multiply" algorithm). On the base processor, a 1024-bit RSA exponentiation with 4-bit windows takes on average 132,334,584 clock cycles, of which some 97.5% is spent on modular multiplication. Consequently, the speed-up factor due to the proposed enhancements is estimated to be about 10.10. A 1024-bit RSA using the binary method takes 156,812,860 clock cycles, of which 97.9% is spent on modular multiplication. Therefore, the estimated speed-up factor is found to be 10.48. We also implemented elliptic curve point multiplication based on Jacobian coordinates as described in [5]; the results are given in Table 4.

Precision	Point mult. (base)	% of modular multiplication	Estimated speed-up
160	5,814,161	87.00%	2.15
192	9,924,418	90.17%	2.63
256	22,342,893	92.49%	4.91
512	158,199,923	96.51%	8.05

Table 4. Timings for Point Multiplication

It is widely believed that there is no need to accelerate the modular inversion when projective coordinates, such as Jacobian coordinates [5], are used. Projective coordinates eliminate all but one inversion from the elliptic curve point multiplication at the expense of more multiplications; this single inversion is needed for the conversion of the result from projective to affine coordinates. However, as we will show in the following, the time spent on the inversion can make a considerable contribution to the overall execution time, especially when the modular multiplications are sped up by our architectural enhancements.

Using projective coordinates, a point multiplication over a 160-bit prime field takes roughly 2,701,349 clock cycles on the enhanced processor. On the other hand, our implementation of the Montgomery inverse for 160-bit operands needs about 78,174 clock cycles if the enhancements are not utilized. Thus, the inversion accounts for merely 2.9% of the total cycles spent on point multiplication including conversion. This does not justify to speed up the inversion operation since any improvement on inversion will only marginally accelerate the entire operation. However, there exist a number of advanced point multiplication techniques with significantly better performance, especially when the base point is fixed and known a priori. For example, using the fixed-base comb method cuts the execution time of a point multiplication down to an estimated 343,905 cycles on our enhanced processor. In this case, the inversion would constitute 22.73% of the total time if it is executed with the base instructions, which makes a strong case for speeding up the inversion operation. The enhanced processor is able to compute the inversion in roughly 36,798 cycles, which translates to 11.98% of the overall execution time.

4.1. Hardware Cost

Adding new instructions and functional units inevitably introduces additional costs. For an embedded processor, it is essential that the extra hardware costs do not exceed the benefits of the enhancements. Table 5 shows the hardware cost of each functional unit in terms of gates in a 0.13 μm CMOS technology. Considering that these enhancements

Functional unit	Gate count (0.13 µm)
Base processor	101,694
CRF	34,004
Multiply unit	38,209
Integer unit	4,524
Shifter	35
Other	18,727
CU total	95,499

Table 5. Hardware Cost of Functional Units

speed up certain cryptographic operations by a factor of up to 10, their benefit far exceeds the associated costs.

5. An AES Implementation Hardened against Cache Attacks

Efficient software implementations of many symmetric ciphers are vulnerable to so-called cache attacks [14] since they usually utilize look-up tables for non-linear function (S-box) evaluations, whereby these tables generally fit into the first or second-level cache of modern processors. The most efficient software implementation of the Advanced Encryption Standard (AES) is due to Barreto [3] and uses four 1 kB tables for the first nine rounds of the 10-round variant of the algorithm. Another table of the same size is used in the final round. Many cache-based attacks on AES software exploit the access patterns to cache lines, which may contain the desired table entry (i.e. cache hit), or not (i.e. cache miss) [1, 2]. Considering the fact that a cache miss introduces a significant and observable delay to the computation (since cache memory is usually much faster than main memory), the differences in the execution time of AES encryption leaks information on the secret (round) keys. We refer the reader to [4, 13] for a formal description of cache attacks.

Practical cache attacks on AES focus either on the first round (as in [13]) or the last round (as in [1]) since these rounds directly interact with the "outside world" by taking the plaintext as input and outputting the ciphertext, both of which are observable by an attacker. Therefore, it is very important to protect the first and last rounds. Implementing even a single round without using look-up tables (in order to not leave any trace in the cache) can be very slow in software due to bit-manipulation operations.

A fast AES implementation secure against cache attacks could use a combination of the countermeasures proposed in [4, 13]. Our enhanced processor can be beneficial when applying these protection methods. For example, the Cryptographic Register File (CRF) sketched in Section 2 can be used to store a part of the look-up tables. Even though the CRF with its 32 128-bit registers is relatively large, it is not capable to accommodate all tables, which are 5 kB in size. However, the CRF can easily hold a 256-byte table with pre-computed values for the byte substitution of one AES round. Since table look-ups now result in accesses to the register file (and not to the data cache), the requested byte is always returned in constant time. Furthermore, as the CRF is not time shared (at least while a cryptographic process has locked it), other (possibly spy) processes can not observe the trace left by the cryptographic process.

[3]	First	Last	First+last	Per round
796	171 (21.5%)	33 (4.5%)	199 (25%)	178 ($\approx$ 22.4%)

Table 6. Overhead (in clock cycles) of protecting the rounds of AES

A small look-up table, specifically placed for AES implementation and optimized for fast access, allows for secure computation of each round of the AES at the expense of a slight degradation in performance compared to a standard (i.e. unprotected) implementation like the one in [3]. This is even the case if the register file is not especially designed for the implementation of look-up tables. The CRF of our processor is geared towards public-key algorithms and thus organized as a one-dimensional array of 32 registers of 128 bits. A drawback of this approach is a certain overhead in accessing the desired bytes in the register file. However, one can always arrange the CRF as a byte array if the execution time of symmetric ciphers is more important.

We modified the implementation from [3] by replacing the look-up table-based round functions with their secure counterparts; the resulting overhead (in number of clock cycles) is listed round-wise in Table 6 for the encryption of a 128-bit block. To give an idea as to how expensive it is to protect even one round of AES, we also implemented an AES version on the base processor without any look-up tables and optimized it for speed. The resulting overhead of protecting only the first round of AES turns out to be as high as 36,125 clock cycles.

As explained in [4] for a standard implementation of the AES, a small table can be used to protect the rounds. We also implemented the standard method for different rounds and found that protecting one round results in an overhead of some 29%, which is higher than ours. Consequently, the standard implementation with all rounds protected requires 2654 cycles, while our fully-protected implementation on the enhanced core takes 2246 cycles, which translates into a 16% improvement over the standard implementation. Note that the standard implementation may still be vulnerable to *synchronized attacks* through a spy process that can evict cache lines during AES computation in a very fine-grained fashion. On the other hand, our AES implementation provides perfect protection against cache attacks and costs no overhead in hardware since we utilize resources which are already included in the core for public-key operations.

The case study of AES shows that the CRF is a good example for a generic architectural enhancement which can be used for many purposes, e.g. speeding up cryptographic operations, storing secret data (e.g. keys), and securing the implementation of non-linear functions (S-boxes).

6. Conclusions and Future Work

We designed and implemented a generic cryptographic unit (CU) that i) facilitates fast and secure execution of a wide range of cryptographic algorithms, and ii) adheres to the 3-address instruction format and can be integrated into almost any general-purpose processor.

To demonstrate the efficiency and applicability of the proposed CU, we integrated it into the execution pipeline of an extensible, embedded RISC processor. We obtained considerable speed-up factors for basic multiple-precision arithmetic operations such as modular multiplication and inversion, which are the the dominant operations of many public-key cryptosystems. The estimated speed-up factors for ECC and RSA gained through the CU are up to 8 and 10, respectively. We showed that the CU can also be used to effectively harden software implementations of symmetric ciphers against certain side-channel attacks such as cache attacks. Our experimental results indicate that the hardware overhead of the proposed CU in terms of silicon area is acceptable for embedded processors. A comparison of the obtained speed-up values and incurred hardware overhead makes clear that the benefits of the CU exceed its cost.

We leave an actual implementation of our enhanced processor on reconfigurable hardware (i.e. FPGA) as a future work. All enhancements introduced by the CU are designed in such a way that they do not incur significantly adverse effects on the critical path of the base processor. For example, the multiply unit consists of four 32-bit multipliers working in parallel; therefore its critical path is similar to that of a 32-bit multiplier. Also the other functional units in the CU were designed with special consideration of the critical path. There will, of course, be a certain penalty in the maximum applicable frequency due to the increase in the total chip area. However, a moderate reduction of the clock frequency is acceptable for embedded applications where relatively low clock speeds are adopted. Exploring the (possibly negative) effect of the CU on the maximum applicable frequency and optimizing the functional units to minimize this effect is left as a future work.

References

[1] O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on AES. In *Information and Communications Security — ICICS 2006*, LNCS 4307, pp. 112–121. Springer Verlag, 2006.

[2] O. Acıiçmez, W. Schindler, and Ç. K. Koç. Cache-based remote timing attacks on the AES. In *Topics in Cryptology — CT-RSA 2007*, LNCS 4377, pp. 271–286. Springer Verlag, 2007.

[3] P. S. Barreto. The AES Block Cipher. Source code, available for download from `http://planeta.terra.com.br/informatica/paulobarreto/EAX++.zip`, 2003.

[4] J. Blömer and V. Krummel. Analysis of countermeasures against access driven cache attacks on AES. In *Selected Areas in Cryptography — SAC 2007*, LNCS 4876, pp. 96–109. Springer Verlag, 2007.

[5] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology — ASIACRYPT '98*, LNCS 1514, pp. 51–65. Springer Verlag, 1998.

[6] A. M. Fiskiran and R. B. Lee. On-chip lookup tables for fast symmetric-key encryption. In *Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2005)*, pp. 356–363. IEEE Computer Society Press, 2005.

[7] J. Großschädl and E. Savaş. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In *Cryptographic Hardware and Embedded Systems — CHES 2004*, LNCS 3156, pp. 133–147. Springer Verlag, 2004.

[8] J. Großschädl, S. Tillich, and A. Szekely. Performance evaluation of instruction set extensions for long integer modular arithmetic on a SPARC V8 processor. In *Proceedings of the 10th Euromicro Conference on Digital System Design (DSD 2007)*, pp. 680–689. IEEE Computer Society Press, 2007.

[9] P. Ienne and R. Leupers. Customizable Embedded Processors: Design Technologies and Applications. Morgan Kaufmann Publishers, 2006.

[10] B. S. Kaliski. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, Aug. 1995.

[11] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[12] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[13] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology — CT-RSA 2006*, LNCS 3860, pp. 1–20. Springer Verlag, 2006.

[14] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[15] Tensilica, Inc. Xtensa LX2 Configurable Processor Core. Product brief, available for download from `http://www.tensilica.com/products/xtensa_LX.htm`, 2007.

[16] S. Tillich and J. Großschädl. Instruction set extensions for efficient AES implementation on 32-bit processors. In *Cryptographic Hardware and Embedded Systems — CHES 2006*, LNCS 4249, pp. 270–284. Springer Verlag, 2006.

[17] T. Vejda, D. Page, and J. Großschädl. Instruction set extensions for pairing-based cryptography. In *Pairing-Based Cryptography — PAIRING 2007*, LNCS 4575, pp. 208–224. Springer Verlag, 2007.