Design and Implementation of a Cryptographic Unit for Efficient, Secure and Trusted Execution of Cryptographic Algorithms

Kazim Yumbul, Erkay Savaş, Övünç Kocabaş
Sabanci University
Orhanli, Tuzla Istanbul, 34956 Turkey
ovunc@su., erkays@sabanciuniv.edu

Johann Großschädl
University of Bristol
Merchant Venturers Building,
Woodland Road, Bristol, BS8 1UB, U.K.
johann@cs.bris.ac.uk

June 11, 2009

Abstract

Cryptographic algorithms, although being secure in theoretical construction, have many practical implementation challenges since they feature compute- and resource-intensive operations and handles sensitive information. When implemented in software on general-purpose processors, they are vulnerable to a multitude of attacks. In this paper, we design, implement, and realize a *cryptographic unit* that can easily be integrated to any RISC type processor for the safe and efficient execution of cryptographic algorithms. Design of the cryptographic unit takes a completely novel approach in the execution of cryptographic algorithms when compared to cryptographic accelerators and architectural enhancements. Although it is integrated to a pipeline of an embedded RISC processor, it is partially an autonomous unit with its own resources, which is analogous to the floating-point unit in this sense. It provides new instructions to accelerate cryptographic algorithms considerably and its associated cost in terms of area is acceptable and justified by the improvement in the performance and efficiency.

The cryptographic unit can also be instrumental in protecting the cryptographic computation against active and passive attacks, and other malicious processes running simultaneously. We demonstrate that the execution of AES encryption can be performed inside the cryptographic unit without any secret and/or sensitive information ever leaving the cryptographic unit during the entire computation. Thus, cryptographic keys and the entire cryptographic computation itself are protected against many powerful attacks such as cache-based side-channel and cold-boot attacks that would compromise the security in entirety otherwise.

1 Introduction

Implementation issues of cryptographic algorithms have been in the focal point of major research efforts for the last two decades. While the performance and efficiency in terms of speed and used resources had been *the* only focus in the early years of the discipline, the concerns about the secure implementation of cryptographic algorithms have become at least as important. A great deal of research work has already been dedicated to performance issues of cryptographic algorithms. Different metrics such as execution time (speed), implementation space (silicon area, code size, memory usage, etc.) and power usage/energy consumption are used to measure the performance quantitatively. Efficiency, which can be thought as a combined performance metric, is usually measured using area-time product that shows how effectively the design space is utilized.

Secure implementation of cryptographic algorithms is another (and increasingly more and more) important goal since the side-channel attacks [25, 26, 12] are shown to compromise the security in entirety independently from the theoretical/mathematical strength of the underlying cryptographic construction. While multitude of countermeasures at different levels from circuit to architectural to algorithmic are being proposed to harden implementations against side-channel attacks, the problem has not been fully solved. This is partly due to the difficulty of the problem and more importantly due to the fact that there is no panacea for absolute protection [32]. Every proposed remedy, therefore, has its own limitations and shortcomings.

A particular class of side-channel attacks concerning mostly software implementations exploits the computational residues left in the micro-architectural constituents [14, 1, 2, 6, 31]. This class of attack clearly underlines the fact that general-purpose processors with totally different design constraints are inadequate in providing the computational primitives for secure execution of cryptographic algorithms from purely software implementation point of view. This necessitates a paradigm shift in the design process of general-purpose microprocessors to accommodate the security requirements.

A general-purpose processors are designed to be versatile in fulfilling diverse set of tasks, which results in sharing of resources by different and simultaneously executing processes. The main protection that keeps processes from interfering with each other is known as *process isolation* enforced by the operating system. The operating system that implements the core protection mechanism, however, is overly complex and usually full of deficiencies (e.g. software bugs exploited by attackers) to provide an adequate level of security.

The cryptographic applications, in particular, require special care in process isolation since they have access to private/confidential information such as secret keys. There is plethora of work on secure execution of programs where the primary goal is to provide the programs with a secure environment free from the interference of other malicious programs. The common perception is that software-only solutions are inadequate and hence that hardware support is necessary. Major microprocessor manufacturers, (Intel, AMD, ARM) already introduced hardware extensions to their processor cores that allow isolated execution of programs [15, 13, 3]. This can be achieved by making the portions of memory, of cache, of TLB used by a program inaccessible to other programs. The techniques proposed in [28, 29] deals with performance problems in isolated execution of security-sensitive codes by minimizing trusted code base and by proposing some hardware extensions. They also allow fast and fine-grained attestation of the code executed. However, both the code base that manages the isolated execution mechanism and the code running in isolated environment have certain privileges (e.g. accessing sensitive information) and therefore must be trusted.

A highly favored practice is to implement cryptographic algorithms on dedicated hardware (e.g. cryptographic accelerators or trusted platform modules as defined in [38]). A single-purpose cryptographic processor can easily overcomes the aforementioned difficulties. Furthermore, being unquestionably free from overly complicated subtleties of process execution semantics on general-purpose processors, hardware implementations provide simple, yet perfectly isolated execution environment for cryptographic algorithms. And lastly, it may not be possible to overperform the hardware implementations in terms of time.

Hardware implementations, however, suffer from different set of disadvantages. First, a dedicated cryptographic processor is essentially a co-processor and relies on a host (general-purpose) processor with which it needs to communicate. The communication not only results in considerable overhead but also introduces new security risks, accrued in processor/co-processor settings. Second, the area of a cryptographic co-processor is generally large. And finally, the last (but not the least) issue is that dedicated hardware implementations are essentially fixed and do not provide sufficient level of flexibility and scalability, which are often needed in cryptographic applications as security requirements and awareness change over time.

The alternative to hardware implementations is modifying/enhancing/re-designing general-

purpose processors in such a way that they not only provide performance, flexibility, and scalability, but also introduce secure execution semantics into the computation core. The advantage is obvious and many-folds: i) tight integration to the processor core, no communication overhead and accrued security risks, ii) relatively moderate space compared to cryptographic accelerators and iii) high degrees of flexibility, scalability, and agility that go far beyond of fixed-function hardware such as a co-processor since the architecture can still be used for general-purpose computing with the potential benefit for other application domains as well. A perfect example for the need of flexibility and agility is the new algorithms [8, 7] for direct anonymous attestation protocols in the context of trusted computing that propose using elliptic curve cryptography and pairing operations. To best of our knowledge no available TPM can be programmed to implement these algorithms.

The main novelty in this paper is the idea of using a *cryptographic unit* (CU) in computation of cryptographic operations and in handling the secret information. The cryptographic unit (CU), a preliminary prototype of which has originally been proposed in [33], is relatively low cost executional unit that can be integrated to many RISC processors as its interface comply with the conventions of RISC-style instruction set architecture (ISA). It is different from all previous proposals since it is more akin to a floating point unit, which can be similarly integrated to the execution pipeline of general-purpose processor while providing different set of instructions and functional units. The cryptographic unit can also be implemented as a separate core in a multi-core design. While sensitive and speed-critical parts of the cryptographic computation can be executed in the CU, the other parts of the computation can be executed on the base processor. The cryptographic unit is flexible and can be programmed to implement any cryptographic algorithm with any key size.

In this paper, we design, implement and realize such a *cryptographic unit (*CU*)* and demonstrate its advantages. Firstly, we report the achieved speedups for basic arithmetic and cryptographic operations through the use of the CU. Later, we provide the area requirements of the SU when implemented in both ASIC and FPGA. To demonstrate its efficiency we provide the time-area product metric obtained for elliptic curve cryptography and RSA operations. We also demonstrate that a cryptographic algorithm (specifically AES) can be efficiently and securely executed in an isolated environment established through the cryptographic unit.

1.1 Related Work and Our Contribution

Previous works [20, 21, 39, 37, 17] propose various enhancements to accelerate cryptographic operations. For instance, the authors in [20] propose five custom instructions to accelerate arithmetic operations in both $GF(p)$ and $GF(2^n)$ on a MIPS32 core to benefit elliptic curve cryptography while ISA extensions in [39] aim to accelerate pairing-based cryptography.

Similarly, the authors in [17] explore the effect of on-chip memory on the execution time of s-box computations in symmetric key cryptography. A common feature of these works is that they focus on custom solutions for accelerating an individual cryptographic operation on general-purpose processors. The instructions that accelerate bit-sliced implementation of AES in [19] also protect the AES computation against known cache attacks since the AES implementation does not utilize any lookup tables.

In our earlier work appeared in [33], we proposed a cryptographic unit (CU) that potentially accelerates cryptographic algorithms such as RSA, elliptic curve cryptography (ECC) and AES. It has been shown that the proposed CU can easily be integrated to any RISC processor including the embedded end of the processor spectrum. In the design of the CU, we take a slightly different and holistic approach by designing/implementing/integrating a *cryptographic unit* (CU) to an extensible embedded processor core that benefits many cryptographic operations. The proposed cryptographic unit facilitates new and powerful custom instructions to accelerate multiplication and inversion in prime finite field GF(p), and potentially cryptographic operations of elliptic curve cryptography and RSA. The CU is also shown to be instrumental in implementing the AES in software, which is resistant against cache-based side-channel attacks. The CU was prototyped for embedded processor core by Tensilica [36] and some estimates for area usage and acceleration rates for RSA, ECC, and AES were given in [33].

In this paper, we improve our formerly proposed CU and show that it can be instrumental in isolated execution of security-sensitive programs besides accelerating cryptographic operations. To be more specific, our contributions can be summarized as follows:

- We fully implement and realize the proposed cryptographic unit and integrate it into an embedded processor core.
- We report on the cost of the CU in terms of area and time overhead in an embedded processor core by Tensilica [36] for both ASIC and FPGA implementations. Note that these values are not provided in [33], only estimated. We show that as far as the efficiency concerned the CU is superior to the base processor without the CU.
- We provide the speedup values obtained through the use of the CU for RSA and ECC operations. Note that the execution times for RSA and ECC given in [33] are only estimates.
- We compared our implementation with a hand-optimized assembly language implementation on a well-known embedded processor (ARM7TDMI) and demonstrate that significant speedups can be obtained using straightforward C language implementation on the CU.

- We also show that the CU is instrumental to provide a protected execution zone for secure and isolated execution of security sensitive programs (e.g. cryptographic algorithms). Put differently, the proposed infrastructure creates an isolated execution environment for a process to run without the intervention of any other process running simultaneously. In that respect, our approach provides a conceptually similar type of protection for cryptographic algorithms to what the Flicker infrastructure in [28] does for the general computation. Since our method proposes a more tightly-coupled *CU* to the processor core, it does not suffer from the communication overhead between the CPU and TPM as is the case in Flicker. Furthermore, the execution of other processes needs not be suspended (as it happens in Flicker) provided that the access to the CU by other processes is not allowed. Our tightly-coupled cryptographic unit follows the new paradigm in embedded system design as stated in [35]: security must be used as a design parameter.

- We implement the AES using the CU in a protected manner and show that confidential values (e.g. secret keys, intermediate values obtained in different phases of the computation) never leave the protected zone. During the computation, unprotected functional units can also be used for non-secret values of the computation. But memory, cache, and architectural registers cannot be used for confidential values at any time since memory is generally not a safe place, we do not have any control over the cache functioning, and architectural registers are subject to automatic spilling at any time.

- We provide the execution time and throughput values for AES algorithm running in protected zone, which are comparable to similar implementations.

As pointed out earlier, the CU alone cannot ensure the isolated execution zone for cryptographic applications; but it is a necessary component. We only implement AES algorithm in isolated fashion since it utilizes simple operations that can be performed entirely inside the CU. The other and more complicated cryptographic applications such as RSA and elliptic curve cryptography require a small amount of cache-like on-chip memory to keep the sensitive values during the calculation. In addition, software support is also necessary in protecting the sensitive values during the cryptographic computation since other simultaneously executing processes can in principle access them. Therefore, the CU usage may need to be restricted to the priviledged protection rings and access to the CU by other processes has to be strictly regulated. However, these issues are either beyond the scope of this paper or left as a future work.

2 General Architecture

In this section, we provide the details about the reconfigurable processor and our basic enhancements to it.

2.1 Configurable Processors

A typical configurable processor consists of a pre-defined processor core which can be enhanced to measure up to specific application requirements. Configuring these processor cores generally includes modifications, additions or removals to processor peripherals, memories, width of the memory bus and handshake protocols to deliver performance improvement for a given set of applications. Once finished with the configuration, configurable processors are synthesized as RTL code and can be mapped to ASIC or FPGA's. ARC, Improv, Tensilica are some of the major companies that offer configurable processor cores.

We prefer Tensilica's Xtensa configurable processor cores as the target embedded processor in our work, since they are one of the configurable cores that offer full software-development tool-chain. In addition, the Tensilica Xtensa cores are also *extensible* in the sense that totally new properties can be added to cores; a property that make them a superset of configurable processors, offering more flexible solutions compared to the other configurable-only processors.

Tensilica offers two types of Xtensa configurable cores: LX2 and Xtensa 7, which are intended for embedded applications. While Xtensa 7 is optimized for low power applications such as control operations, LX2 cores are more flexible and ideal for high performance, data-incentive applications. Therefore, we choose an LX2 core as our base processor since cryptographic applications employ multi-precision arithmetic in finite fields and performing these operations requires high performance.

Xtensa's LX2 32-bit processor architecture features a compact instruction set optimized for embedded system designs. The base architecture includes a 32-bit ALU, up to 64 general-purpose physical registers, 80 base instructions including 16 and 24-bit instead of RISC instruction encoding which enables significant code size reductions [36]. Furthermore an LX2 core has two essential features; namely configurability and extensibility, which will be utilized in the process of generating our *cryptographical unit*.

Configurability attribute of LX2 core offers designers to adjust their design for the specific applications where they can modify the processor core according to their design specifications. Modification of processor can be made by defining the width and number of execution units, data interfaces and optional data paths. Whereas with *extensibility* feature, custom execution units, registers, register files, single-instruction multiple-data functional units can be added to processor data path. Extensions to data path is achieved through Tensilica In-

struction Extension (TIE) language. TIE is a Verilog-like language which is used to describe instruction set extensions to processor core. Functional behaviors of desired extensions are defined in TIE and TIE compiler will generate and place the RTL equivalent blocks into processor data path.

2.2 *Cryptographically-Enhanced Processor*

Our purpose is to design a cryptographic unit *CU* that can be integrated to a processor core so that it not only accelerates public key algorithms such as RSA and elliptic curve cryptography, but also help provide secure and isolated zone for all cryptographic algorithms to run without any performance overhead. The processor core that incorporates the *CU* is referred throughout this paper as either *cryptographically-enhanced processor* or *enhanced processor*.

Design process of creating such processor consists of two steps. First, LX2 processor core is configured into so called *base processor* and then the *base processor* is extended with the *CU* by using TIE language to build final configuration.

Base Processor In order to demonstrate that it is feasible to integrate the proposed cryptographic unit any RISC processor core, we select to configure (resource-wise) a very conservative embedded processor as a base processor. To keep processor size as small as possible, unnecessary units are removed from LX2 core. For instance, floating point and 32-bit integer divider units are removed since they are not relevant in this context. Data and instruction caches are also chosen as reasonable sizes and direct-mapped cache. To increase the processor's performance, memory-cache interfaces and Processor Interface(PIF) are chosen as 128-bit (largest available) to increase bandwidth and word size of the processor. The configuration of the *base processor*is presented in Table 1.

Unit	**Configuration**
Multiply Unit	32 bit
Register File	32 × 32-bit
Data memory/cache interface	128-bit
PIF interface	128-bit
Data Cache	8Kb / direct-mapped / 16byte line size
Instruction Cache	8Kb / direct-mapped / 16byte line size

Table 1: Configuration of the *Base Processor*

Pipeline length of the LX2 core is also configurable and two versions of the *base processor* are generated with 5- and 7-stage pipeline length. The hardware costs of 5- and 7-stage

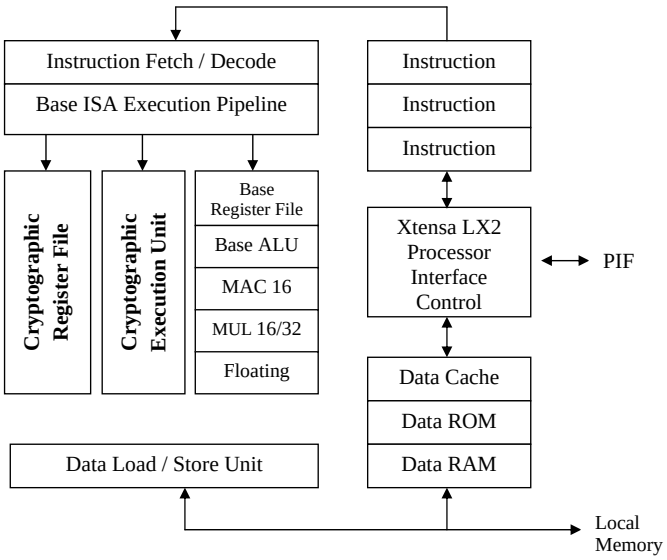

Figure 1: General Architecture of Enhanced Embedded Core

pipelined versions of *base processor* in $0.13m$ CMOS technology are approximately 102000 and 115000 equivalent gates, respectively.

In the design process of the cryptographic unit and in its integration to the base processor the following criteria are adopted:

- unacceptable increase in area is avoided
- change in instruction format and size is not considered
- easy integration with available tool-chain (e.g. compilers, debuggers, linkers) is ensured
- major change in the control circuitry and existing pipeline structure is avoided

Figure 1 shows the integration of the *CU* to the general architecture of Xtensa LX2. The *CU* consists of two parts: *cryptographic register file (CRF)* and *cryptographic execution unit (CEU)*. In the following sections, the *CRF* and the *CEU* are explained in detail.

2.2.1 *Cryptographic Register File (CRF)*

The *CRF* is an array of 32 registers each of which has 128-bit width and is used to store operands and temporary results of arithmetical operations. Storing these values in the *CRF* will significantly reduce the execution time since the number of time-consuming memory access operations will be reduced. Besides, the *CRF* can be used to store sensitive information

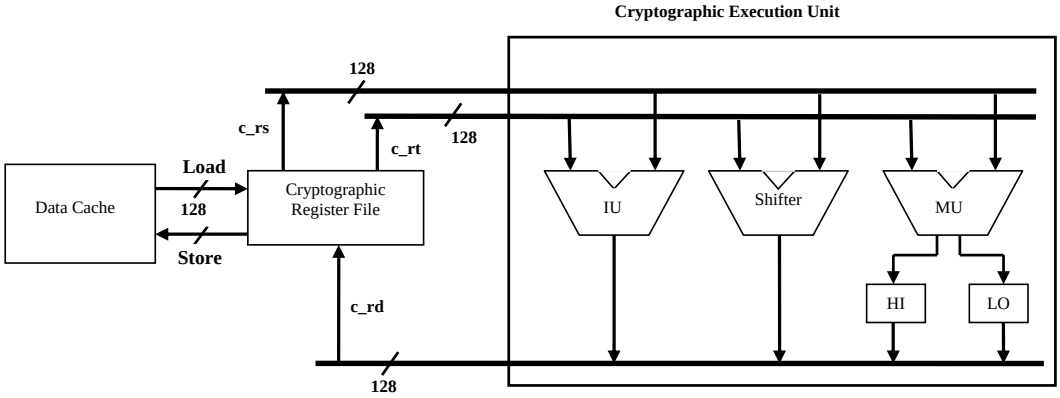

Figure 2: Detailed Architecture of the Cryptographic Unit (*CU*)

such as secret keys and small look-up tables for increasing security level of cryptographic algorithms. In subsequent sections, we will show that the *CRF* will be of crucial importance for protecting software implementation of AES from side-channel attacks; e.g. cache attacks.

2.2.2 *Cryptographic Execution Unit (CEU)*

The *CEU* is the new execution unit designed to utilize 128-bit width processor interface and the *CRF* during cryptographic operations. By choosing interface precision as 128-bit we simply increase our word length to 128-bit for cryptographic operations instead of 32-bit word size of general purpose processors. Using 32-bit ALU in the core processor will be inefficient for these operations, therefore the *CEU* is designed to be used as functional unit for cryptographic operations. Functional units of the *CEU* will now take their operands from the *CRF* instead of 32-bit physical registers of the core processor.

The *CEU* is composed of three parts: an integer unit, a shifter circuit and a multiply unit. While *Integer Unit (IU)* is capable of adding/subtracting and comparison of two 128-bit integers, shifter circuit performs shift operation on both directions on a 128-bit register. Final functional unit in the *CEU* is multiply unit which performs 128-bit multiplication, and generates 256-bit result and stores the most and least significant 128 bit of the result on special purpose registers `HI` and `LO` respectively. Figure 2 shows the detailed architecture of the *CU* and functional units inside the *CEU*.

2.3 Multiply Unit

Multiply unit is the most crucial functional unit of the *CEU* for accelerating modular multiplication operations which is performed excessive number of times in RSA and elliptic curve cryptography as well as many other public key algorithms.

A 128-bit multiplication can be decomposed into 16 32-bit multiplications in general. One can choose to instantiate 16 multipliers to calculate all 32-bit multiplications in parallel and a following cycle to add the partial products to get the final result. Yet, using 16 32-bit multipliers will severely increase the processor area. Instead we prefer to implement 128-bit multiplication by utilizing four 64-bit multiplications and add the aligned partial products to get 256-bit result. In each 64-bit multiplication, four 32-bit multiplications is performed. For this, we instantiate four 32-bit multipliers to execute them parallel. By using 4 parallel multipliers instead of 16 we can still obtain a significant speed up at the expense of acceptable hardware cost.

A 128-bit multiplication can be decomposed into four 64-bit multiplications as shown in Figure 3. Each 64-bit multiplication produces a partial product of 128-bit and in the end all partial products are aligned and summed to compute final product. Final product, 256-bits, is stored on `HI` and `LO` special purpose registers as presented in Figure 4. In what follows, computation of the partial products in parallel by using four multipliers is explained first and then the alignment and addition of partial products to the 256-bit final product is shown in detail.

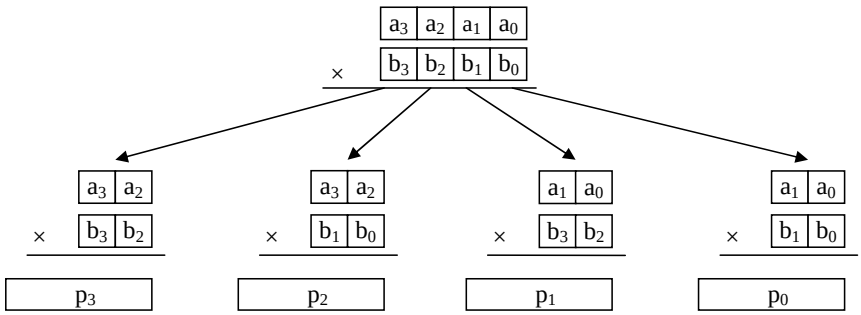

Figure 3: Dividing 128-bit multiplication into four 64-bit multiplication

2.3.1 Computing Partial Products

In a 64-bit multiplication, four 32-bit multiplications are performed in parallel in one (1) clock cycle using four 32-bit multipliers, which are shown in Figure 5. As seen in the figure,

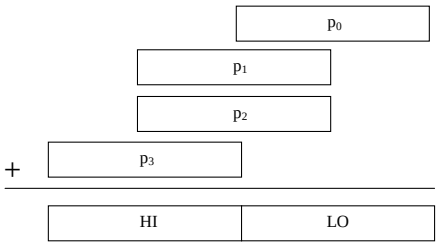

Figure 4: Computing the Final Product

the partial product calculation can be finished in three clock cycle, two of which are spent on the subsequent alignment and addition operation within the partial product. `HI` register stores the t_l and t_h of the results while `LO` register stores t_{int1} and t_{int2}. Before calculating the partial product, which is 128 bits, two more operations have to be performed. First, the intermediate results are added (t_{int1} and t_{int2}) and then the sum is aligned and added to the value in `HI` register. After these operations the partial product is calculated and stored in a 128-bit register.

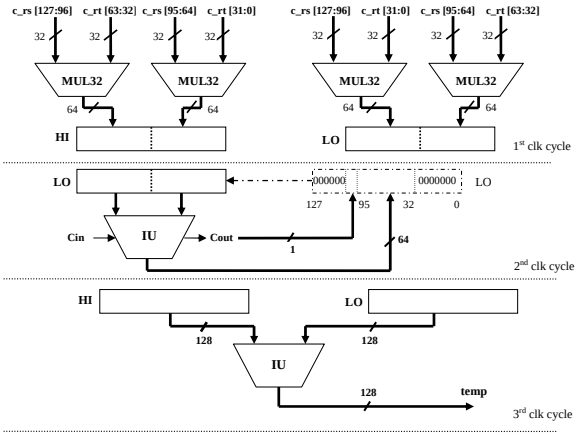

Figure 5: Partial Product Computation

2.3.2 Alignment and Addition of Partial Products

Four partial products of 128-bit each, namely p_0, p_1, p_2, p_3 (cf. Figure 6), which are calculated in the previous step, are stored temporarily in four 128-bit registers. Final product is computed after three iterations which is composed of successive additions of partial products

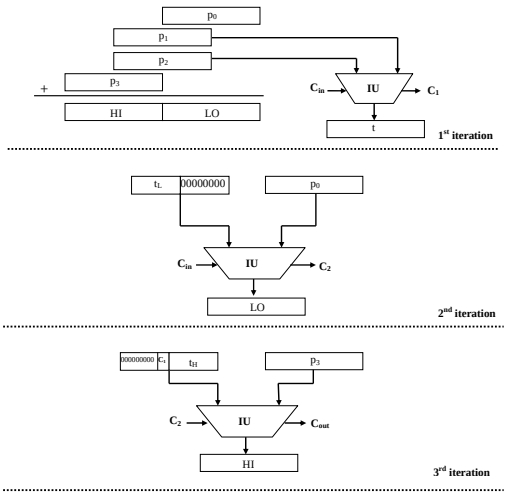

Figure 6: Alignment and addition of Partial Sums

into `HI` and `LO` registers. These iterations are also summarized in Figure 6 and explained below.

1st iteration: Partial products p_1 and p_2 are added and the result (t) is stored temporarily in a register (In following iterations, t will be divided into 2 halves, t_H and t_L, and each half will be used as operands of addition to `HI` and `LO` registers). Also the carry of the addition, C_1, is stored in 1-bit carry register as it is used in the calculation of result on `HI` register in the final iteration.

2nd iteration: In this iteration, lower half of the partial sum calculated in 1st iteration, t_L, is added with the p_0 and result will be the lower half of the final product and stored in the `LO` register. Again the carry out from this step, C_2, is stored in a carry register and is used in the final iteration.

3rd iteration: With the final step, final product is calculated and stored in `HI` and `LO` registers. In this iteration, upper half of the partial sum of the first iteration, t_H, is concatenated with C_1 and summed up with the p_3. During the addition, the carry of the second iteration, C_2, is used as the carry-in value. Finally, the result of the addition is stored in `HI` register.

2.4 Proposed Instructions

A new family of instructions is introduced to the processor ISA to fully employ the *CEU*. These instructions operate on 128-bit operands and conform to instruction type and formats of LX2 core which uses RISC instruction encoding. Therefore new instructions are encoded

as RISC instructions with a slight difference. Common notations of source, target and destination registers (denoted as *rs*, *rt* and *rd* respectively) in RISC encoding are adjusted to reflect changes such that functional units in the *CEU* uses operands stored in the *CRF*. Therefore source, target and destination registers of the *CRF* are designated as `c_rs`, `c_rt` and `c_rd`.

The proposed instructions are presented in Table 2. `ADD_CREG` and `SUB_CREG` operations perform unsigned addition and subtraction respectively. Both operations take their operands from `c_rs` and `c_rt` registers and write result back to `c_rd` register. `COMP_CREG` operation compares the values of `c_rs` and `c_rt` registers and if the value of `c_rs` register is greater than `c_rt` register than it writes 1 to `c_rd` otherwise it writes 0. `SHL_CREG` and `SHR_CREG` operations perform 1 bit shift operation however, the `CRF` has 2 read and 1 write port only the value of `c_rs` register can be changed while the value in `c_rt` register remains unchanged. `MUL_CREG` operation performs 128-bit unsigned multiplication and writes product to `HI` and `LO` special purpose registers. Finally, `LOAD_CREG` and `STORE_CREG` operations perform data transfer operations between memory and the *CRF* for given memory address.

Format	**Description**	**Operation**
`ADD_CREG c_rd, c_rs, c_rt`	Unsigned Addition	(`C`$_{out}$`, c_rd) := c_rs+c_rt+C`$_{in}$
`SUB_CREG c_rd, c_rs, c_rt`	Unsigned Subtraction	(`B`$_{out}$ `, c_rd) := c_rs-c_rt-B`$_{in}$
`COMP_CREG c_rd, c_rs, c_rt`	Comparison	`c_rd = c_rs > c_rt ? 1 :0`
`SHL_CREG c_rs, c_rt`	Shift together left	`c_rs := c_rs[126:0] || c_rt[127]`
`SHR_CREG c_rs, c_rt`	Shift together right	`c_rs := c_rt[0] || c_rt[127:1]`
`MUL_CREG c_rs, c_rt`	Unsigned Multiplication	(`HI / LO`) := `c_rs` × `c_rt`
`LOAD_CREG c_rd`	Load data from memory	`c_rd := Memory [address]`
`STORE_CREG c_rd`	Store data to memory	`Memory [address] := c_rd`

Table 2: List of New Instructions

3 Software Implementations of Arithmetic Operations on Enhanced Core

In this section, we explain the implementation details of two important arithmetic operations for our enhanced embedded processor; namely multi-precision modular multiplication and modular inversion.

The proposed processor is based on the extensible, embedded processor core [36], released as Xtensa LX2 by Tensilica. We use the term *base processor* for Xtensa LX2 with basic instructions, 8 KB separate instruction and data cache memories and a 32-bit multiplier.

Table 3: Implementation Results for Modular Multiplication in 5-stage Pipeline

Precision	**CIOS (base)**	**SOS (with *CU*)**	**Speedup**
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

We enhance the base processor with tightly-coupled cryptographic unit, hence the name *enhanced processor*. We run our algorithms on both the base and enhanced processors and reported the results in the subsequent sections.

3.1 Modular Multiplication

Koc et al. [24] proposed five algorithms to implement the Montgomery modular multiplication (the fastest method for modular multiplication) [30] operation in software. The CIOS method in [24] seems to be the best choice since it has a regular execution pattern and needs the least memory space among others. However, we use the SOS method that basically consists of two phases: i) the schoolbook multiplication of two big integers and ii) the Montgomery reduction. Even though the SOS doubles the memory space, it does not use all the variables at the same time. Since each phase in isolation requires less memory than the CIOS, which executes the two phases interleaved, all the operands needed in the SOS phases can fit in the *CRF* for up to 1024-bit multiplications. Table 3 summarizes the implementation results of the CIOS algorithm on the base processor and the SOS algorithm on the enhanced processor in terms of number of clock cycles.

3.2 Modular Inversion

Inversion is a relatively slow operation needed in both RSA (e.g. key generation or CRT method) and the elliptic curve cryptography. While it is possible to avoid 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 binary extended Euclidean algorithm and its variation, the Montgomery inversion algorithm [23]. We elect to implement the Montgomery inversion both on the base and the enhanced processors. The results are enumerated in Table 4.

Table 4: Implementation Results for Modular Inversion in 5-stage Pipeline

Precision	**Inversion (base)**	**Inversion (with *CU*)**	**Speedup**
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 5: Implementation Results for Modular Multiplication in 7-stage Pipeline

Precision	**CIOS (base)**	**SOS (with *CU*)**	**Performance Loss**
160	3,032	1,132	9.66% / 8.12%
192	3,747	1,282	−3.25% / 7.19%
256	7,310	1,013	9.25% / 8.81%
512	27,856	2,598	8.83% / 9.85%
1024	108,493	8,418	8.16% / 9.98%

3.3 Modular Multiplication and Inversion on Seven-Stage Pipeline

In order to show that the proposed *CU* can easily be integrated to RISC processors with different properties, we implement it on a seven-stage pipelined version of Xtensa processor. Especially, it is important to demonstrate that having a long latency instruction such as 128-bit integer multiplication does not cause a significant performance degradation due to true data dependencies[1]. The timing results of modular multiplication and inversion for both the base processor and enhanced processor with the *CU* are given in Table 5 and Table 6, respectively. The last columns in the tables enumerate the performance loss in algorithm executions in clock counts due to the deeper pipeline when compared to five-stage pipeline.

As can be observed from Tables 5 and 6, the modular multiplication and inversion operations take longer in seven-stage pipeline than in five-stage pipeline. This result is expected since the true data dependencies have a higher negative impact in deeper pipelines. This fact is confirmed by our implementations of Montgomery multiplications and inversions on the seven-stage pipeline. It is a well-established fact that the modular multiplication is much easier to parallelize than modular inversion operation. In our software implementations in the base processor with seven-stage pipeline, Montgomery inversion operation accrues more overhead due to the true data dependencies in the algorithm; on average twice the overhead of the modular multiplication. However, when implemented in the proposed *CU* that are integrated to the seven-stage pipeline both algorithms suffer equivalently (i.e. about 10%). This results clearly shows that the long latency instructions implemented in the *CU* have

[1]An instruction that needs the result of a 128-bit multiplication instruction may have to wait for the result and this may lead to long stalls in the pipeline (hence data dependency).

Table 6: Implementation Results for Modular Inversion in 7-stage Pipeline

Precision	**CIOS (base)**	**SOS (with CU)**	**Performance Loss**
160	94,174	40,789	20.27% / 10.31%
192	127,296	48,485	19.98% / 10.53%
256	207,939	63,354	20.61% / 10.82%
512	698,104	153,821	20.39% / 8.45%

Table 7: Implementation Results for Elliptic Curve Point Multiplication

Precision	**Point mult. (base)**	**Point mult. (with CU)**	**% of modular multiplication**	Speedup
160	5,684,844	2,695,097	87.00%	2.11
192	9,774,069	3,673,000	90.17%	2.66
256	21,509,576	4,412,633	92.49%	4.87
512	160,109,439	19,798,812	96.51%	8.08

no particular negative effect in longer pipelines.

4 Timing Results for Elliptic Curve Cryptography and RSA

In this section, we provide the speedup values obtained for both RSA and elliptic curve cryptography.

We implemented a simple 1024-bit RSA using two methods: windowing method with 4-bit window size and no-windowing. On the base processor, 1024-bit RSA with 4-bit windows takes on average 132,361,636 clock cycles, 97.48% of which is spent on modular multiplication. Consequently, the speedup on the processor with the CU is found as 11.26. 1024-bit RSA with no windowing method takes 156,812,860 clock cycles, 97.86% of which is spent on modular multiplication. The speedup for this case is found out to be 11.47.

Similarly, we implemented elliptic curve scalar point multiplication with jacobian coordinates [11] and the implementation results are given in Table 7.

It is a common tendency to think that there is no need to speedup inversion operation due to the projective coordinates (e.g. jacobian coordinates) [11]. The projective coordinates eliminate all but one inversion from elliptic curve point operations at the expense of more multiplications; only a single inversion operation is needed for converting the resulting point from projective coordinates to affine coordinates. We demonstrate, in this section, that the time spent even on a single inversion might be significant especially when the modular

multiplication is performed in our enhanced processor.

Using projective coordinates, one elliptic curve scalar point multiplication takes approximately 2,695,097 clock cycles for 160-bit elliptic curve on the enhanced processor. Our implementation of Montgomery inversion for 160-bit operands would consume, on the other hand, 76,692 clock cycles if the CU is not utilized. The inversion consumes only about 2.8% of all clock cycles spent on scalar point multiplication including conversion. This does not call for a need to speedup the inversion operation since any improvement on inversion will marginally speedup the entire operation.

There are, however, pre-computation techniques that significantly improve the elliptic curve point operations. For example, with fixed-base comb method it is possible to perform one scalar point multiplication in 342,901 clock cycles on the enhanced processor. This time, the inversion operation would consume about 22.36% of clock cycles without the CU; which is a good motivation for speeding up inversion operation. Consequently, this would be translated into 11.78% speedup in one point multiplication due to the improvement in inversion calculations.

5 A Secure and Isolated Implementation of AES Algorithm

[2]

Efficient software implementations of many symmetric key ciphers rely on heavy use of lookup tables, which naturally makes them vulnerable to cache-based side-channel attacks [14, 2, 6, 31]. The lookup tables are used to capture the input-output mapping of nonlinear functions (s-box) used in cryptographic algorithms. These tables usually fit in the first or second-level caches of modern processors. The most efficient AES implementation in software to best of our knowledge is due to Barreto [4] where four 1 KB tables are used for the first nine rounds of 128-bit AES. Another table of the same size is used for the last round. Many cache-based attacks [14, 1, 2] exploit access patterns of cryptographic process to cache lines, which may contain the desired table item (cache hit), or not (cache miss). A spy process running simultaneously to the cryptographic computation can find out the access patterns of cryptographic process by creating carefully timed access patterns of its own to the same cache.

For instance, one particular type of cache attacks can be described as follows. In a multitasking operating system where simultaneously executing processes take their turn in their time slices, a spy process, when it is switched in by the operating system, can cause all

[2]The results in this section of the paper are to be presented in a conference

cache lines to retire to memory by generating memory accesses that change every cache line. Next time the cryptographic process switches in again, it generates accesses to some of the cache lines, where these accesses are to a great extent determined by the secret information. The spy process which will be switched in afterward will generate memory accesses that cause changing every cache line. Some of memory accesses will result in a cache miss if previous access of cryptographic process to the same cache line retires spy's process data to the memory. Considering the fact that a cache miss introduces a significant and observable delay to the computation due to the fact that cache memory is much faster than the main memory, the spy process can understand which parts of the cache lines are accessed by the cryptographic process by timing every access of its own. We naturally assume that the spy process knows the virtual address map of the cryptographic process (e.g. it knows the virtual addresses of lookup table entries, perhaps those of secret keys but not the content). For a formal model of cache attacks, one can refer to [6, 31].

Majority of general-purpose processors including embedded systems support multi-tasking execution and micro-architectural residues in cache memories or branch prediction units are natural side-effects of the thread-level parallelism exploited during the computation. With multi-core architectures pointing out the near-future trend in processor technology, resource sharing will remain as the indispensable feature of computation. The techniques such as shared memory, cache coherency, and fast communication infrastructure connecting cores only further emphasize the opportunities of the new era in processor technology. Yet, resource sharing instigates an insecure environment for processes to execute. The processes, on the other hand, frequently handle confidential data and therefore need a secure, perhaps isolated, zone to execute. Being aware of this fact, major processor manufacturers equips their processors with extensions to provide such an isolated execution environment [13, 15, 3]. To provide isolated execution of a security sensitive code, the following requirements need to be addressed:

- The secret information can only be in trusted part of the memory. However, a common practice is to implement the entire virtual address space of a process in an isolated section of the physical memory. This naturally requires a considerable hardware support and results in divergence from classic execution semantics. A significant negative impact on the time performance is a natural consequence most of the time. One final disadvantage of this approach is that it is vulnerable to so-called *cold boot memory attack*, where the secret information can be recovered from the memory from which the power is cut off during the computation [22].

- Cache memory must implement a mechanism that prevents processes from using the same cache lines. The mechanism distinguishes the data blocks coming from protected

and regular parts of the physical memory. The partitioned cache [34] is proposed to thwart side-channel attacks.

- The registers are storage elements where the data is actually processed. Naturally, confidential data will be in some of the registers in some point of the computation. However, the registers are of limited quantity and their contents are frequently saved in the stack implemented in the physical memory when the system runs short of registers (*spilling*). Even within a process, one cannot have any control over spilling process (e.g. rotating registers in IA 64 architectures). When the entire address space is implemented in protected regions of memory, the register contents will never leave the protected region since the stack is also part of the virtual address space. Yet again, memory protection is usually costly and vulnerable to cold boot attacks.

Our cryptographic unit fulfills these requirements using a slightly different approach as explained in the following:

- The confidential values, including secret keys and intermediate results (e.g. the state variable of 128-bit in standard AES implementation), never leaves the processor and are never stored in memory. The confidential values stay always within the cryptographic unit in special registers. The register requirements for these values are relatively low.

- The lookup tables are implemented in the cryptographic register file (CRF). Since the content of the cryptographic register file is never spilled to the memory they are never cached, hence the AES implementation is not vulnerable to cache-based attacks.

- The architectural registers are generally not used to hold confidential values. They are only used to keep public data such as loop indices and some temporary variables. If temporary variables are somewhat stored in architectural registers even for a while, they are used in such a way that they are not spilled to the memory (e.g. no function call is made before the used register is reset in rotating register architectures). A relatively few number of special registers are sufficient to hold all the secret values during the computation.

To summarize, in our approach, secret and confidential values always stay in the protected zone inside the processor, i.e. the cryptographic unit. Here, the assumption is that the cryptographic unit is protected against tampering and other types of physical attacks (tamper-proof). In the following, we outline the architecture of the cryptographic unit that allows the protected execution of AES rounds.

Table 8: Hardware Cost of Cryptographic Unit

Block cipher algorithm	**Lookup table size**
DES/3DES	256 Bytes
AES	256 Bytes
Twofish	512 Bytes (two 8×8-permutation tables)
Serpent	256 Bytes

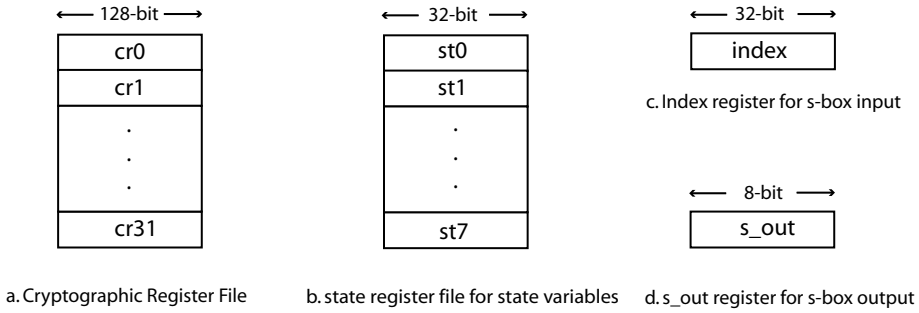

Figure 7: Cryptographic register file used to implement secure lookup table for s-box

5.1 Architectural Enhancements Allowing Protected Execution of AES

Many block cipher algorithms utilize one or more highly-nonlinear functions usually referred as s-boxes. Larger s-boxes are generally better since they increase the security of the block cipher algorithm. However, due to implementation concerns s-boxes are usually of moderate sizes in practice. Table 8 shows the lookup table sizes needed to implement the s-boxes of some of the well known block cipher algorithms. The lookup tables for these block cipher algorithms can be implemented in our cryptographic register file (*CRF*) shown in Figure 7.a since the *CRF* is 512 bytes in total sizes. We, indeed, use the first 16 cryptographic registers (from `cr0` to `cr15`) to hold the lookup table of AES in our implementation.

Since the *CRF* is originally designed to accelerate multi-precision arithmetic for public key algorithms, it is possible to perform 128-bit integer arithmetic (e.g. addition, subtraction, and multiplication) and bitwise logical operations (e.g. and, or, xor) with a single native instruction. Especially, bitwise logical operations are very useful in AES implementation. On the other hand, some other functional units and instructions are needed to perform table lookup operation through the *CRF*, which are shown in Figure 7.

A cryptographic register in the *CRF* can be considered as a register capable of holding 16 bytes of s-box lookup table as demonstrated in Figure 8. The register (`crx`) represents any one of the cryptographic register in the *CRF* and (`crx[i]`) stands for an individual byte

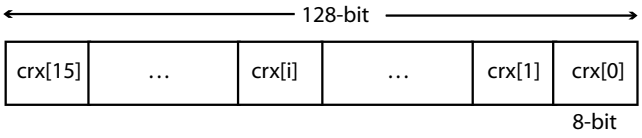

Figure 8: One cryptographic register holding 16 Byte of lookup table

within the register (`crx`). The individual bytes of cryptographic registers are not directly accessible. As explained in the following, only special instructions can transfer an individual byte of a cryptographic register into a special register (`s_out`) in Figure 7. Direct access to an individual byte and its transfer to any other register would overly complicate the design and incur high performance penalties.

Figure 7.b shows a small register file that can hold two versions of the state variables (each of which is 128-bit) in each round; one for the old and one for the newly computed AES block (or state variable). Recall that each AES round takes 128-bit block as input and generates a new 128-bit block as output. For 192, and 256-bit implementations of AES, only state register file needs to be modified, whose area overhead is negligible compared to overall area of the processor. In every round, we perform 16 s-box operations for 128-bit AES. The state variable of AES is in (`st0, st1, st2, st3`) at the beginning of each round. The first 32-bit part of the state variable is transferred from (`st0`) to (`index`) register shown in Figure 7.c using the instruction (`mv_st2index st0`). The least significant byte of the (`index`) register is used to access the s-box output which is stored in the *CRF*. Note that one cryptographic register file can hold 16 bytes of the AES lookup table. For instance, the cryptographic register (`cr0`) stores the s box outputs for the input bytes from 0 to 15.

The upper four bits of the least significant byte of (`index`) register is used to determine which cryptographic register holds the desired s-box output. The least significant four bit of (`index`) gives the offset of the s-box output within the cryptographic register file. Once the cryptographic register that holds the s-box output is known, the instruction (`rd_tab_creg crx`) reads the s box output from the cryptographic register (`crx`) and puts it in another register (`s_out`), shown in Figure 7.d. The instruction (`mv_sout2st stx`) first rotates (`stx`) to right by one byte and put the content of (`s_out`) register in the most significant byte of the state register (`stx`).

When one table lookup operation for one byte of the AES block is completed, the content of (`index`) register is shifted to the right by one byte. As a result of this, the table lookup operation for the second byte can start. When lookup operations for four bytes read from the state register to the (`index`) register are completed, the next four bytes are transferred from the next state register (i.e. `st1`) to the (`index`) register and the same operations are applied

Table 9: Special instructions for AES implementation

Instruction name	**Syntax**	**Definition**
rd_tab_creg	rd_tab_creg crx	s_out:=crx[index ∧ 0xF]
shlmod	shlmod std, sts, arx	std[i]:=sts[i] ⊕ (std[i]_7) ∧ arx i = 0,1,2,3
rowop	rowop str, sts	s_out := index_7∧sts[3] ⊕ index_6∧sts[2] ⊕ index_5∧sts[1] ⊕ index_4∧sts[0] ⊕ index_3∧str[3] ⊕ index_2∧str[2] ⊕ index_1∧str[1] ⊕ index_0∧str[0]
mv_st2index	mv_st2index std	index := (0,0,0,std[3]) and std := (0,std[3],std[2],std[1])
mv_sout2st	mv_sout2st std	std := (sout,std[3],std[2],std[1])
mv_cr2st	mv_cr2st std, crs	std := index_0∧(crs[3],crs[2],crs[1],crs[0] ⊕ index_1∧(crs[7],crs[6],crs[5],crs[4]) ⊕ index_2∧(crs[11],crs[10],crs[9],crs[8]) ⊕ index_3∧(crs[15],crs[14],crs[13],crs[12])

to (index) register content as well. The table lookup operation finishes for one round when 16 table accesses are completed. Note that except for the *CRF*, other registers are of special type and they cannot be accessed directly. Only special instructions shown in Table 9 can access the content of these registers.

Note that the instructions discussed so far are not designed to benefit particularly the AES algorithm. They benefit all block cipher algorithms that utilize relatively small s-boxes such as DES/3DES, AES (Rijndael), Serpent, Twofish. Only instruction that may be considered as specific to AES is (shlmod std, sts, arx *cf.* Table 9.) that can perform four simultaneous shift left operations by one bit in binary extension field $GF(2^8)$. If the irreducible polynomial of $GF(2^8)$ we work in is $p(x) = x^8 + r(x)$ then the architectural register (arx) in the instruction is initialized to $r(x)$. For instance, the irreducible polynomial of $GF(2^8)$ used in AES is $x^8 + x^4 + x^3 + x + 1$ and therefore (arx := 0x1B). In Table 9, std[i] stands for the ith byte of the destination state register std and sts[i]_7 for the most significant bit of the ith byte of the source state register (sts). The instruction (shlmod) works for any irreducible polynomial and can benefit the applications using $GF(2^8)$ arithmetic. When (arx := 0x0), the instruction can perform four regular left shift operations in one clock cycle.

Another instruction used in our AES implementation is (rowop str, sts), that takes two words (32-bit variable) stored in (str) and (sts) registers and xor certain bytes of these

two words which are determined by the bits of the (`index`) register, where (`index_i`) stands for the ith least significant bit of (`index`) register (*cf.* Table 9) . The resulting byte is stored in the (`s_out`) register. This instruction is useful in matrix arithmetic where the elements of the matrix are in $GF(2^8)$.

The rest of the new instructions in Table 9 are moving data between the special registers and cryptographic registers. They have generic usage since we need to move the data around if we want to use the cryptographic unit. They are easy to implement, do not incur significant overhead in area, and definitely are not in the critical path of the processor.

As can be observed from the discussions in this section, our approach is not to integrate powerful instructions that can provide superior performance, specific to the cryptographic algorithm in question and very expensive to implement. Our design principle is to propose simple and inexpensive instructions that can benefit a wide range of cryptographic algorithm implementations while providing a secure and isolated execution.

5.2 Time Performances of the Different Implementations of AES Algorithm

In this section, we compare the time performances of four different (and state-of-the-art) implementations of AES.

The first implementation is due to Barreto [4] that is one of the most efficient (i.e. the fastest) implementation of AES in software. The implementation uses relatively large tables that fits in the first and second level caches and naturally vulnerable to cache-based side channel attacks. The second implementation is known as *standard* implementation and uses a 256 B lookup table. The standard implementation is a straightforward implementation and not vulnerable to the cache-based attack, but utilizes memory, cache, and architectural registers to perform operations involving confidential data. The third implementation, referred as *hardened* in [33], utilizes the *CRF* to store the lookup table. It is secure against cache attacks, but do not run in an isolated zone.

The hardened implementation in [33] give the overhead in number of clock cycles per round to protect a particular round. Most powerful attacks focus either on the first round of AES as in [31] or on the last round as in [1] since these rounds directly interact with outside world by taking the plaintext and outputting the ciphertext, which are easily observable by an adversary. Therefore, it is of utmost importance to protect the first and last rounds. Implementing even one round without using any lookup tables (in order not to leave any trace in the cache) can be very slow in software due to involved bit manipulation operations. The idea is naturally is to use the lookup table in the *CRF* for any round we want to protect and otherwise use the large tables in memory for other rounds. Table 10 lists the overhead

(in number of clock cycles) round wise in Table 10 for a single block encryption of 128 bits.

Table 10: Overhead of protecting the rounds of AES against cache attacks (in clock cycles)

[4]	$1st$	last	$1st$+last	per round
796	171 (21.5%)	33 (4.5%)	199 (25%)	178 ($\approx$ 22.4%)

Finally, the fourth implementation, as we prefer to call *isolated* uses the lookup table in the *CRF* and does not use memory, cache or architectural register to store confidential data. It is not only secure against cache-based side channel attacks but also run in an isolated zone without any interference from any other simultaneously executing process. None of the confidential values such as secret key, round key, and intermediate blocks from AES rounds will leave the protected zone during the AES computation. Under the assumption that the *CU* is manufactured as tamper-proof, the isolated implementation of AES can even withstand cold-boot attacks.

The time performances of the four aforementioned AES implementations are given in Table 11. As can be observed from the figures in the table, Barreto's implementation performs much better than the other three implementations. This is due to the fact that Barreto's implementation mainly consists of lookup operations to the five large tables stored in memory. As long as the processor provide fast memory accesses through the use of first or second level caches, it is almost impossible to provide a better performance than the Barreto's implementation. This implementation, however, has been demonstrated to be vulnerable to cache attacks.

Table 11: Time performance of the four software implementations of AES

Implementation	**Time performance (clock cycles)**	**Characteristics**
Barreto [4]	796	Fast, Insecure
Standard	2654	Moderate Speed, Secure, No Isolation
Hardened [33]	2246 (est.)	Moderate Speed, Secure, No Isolation
Isolated (this work)	2620	Moderate Speed, Secure, Isolated

The standard implementation, which is generally considered as secure against cache attacks[3] provide a moderate performance while the hardened implementation provides expect-

[3]It may still succumb to so-called *synchronized attacks* through a spy process that can evict cache lines during the AES computation with a very fine-grained precision.

edly 16% improvement over the standard implementation. One can always selectively protect the AES rounds in order to increase performance of hardened AES implementation.

The final AES implementation executes AES encryption operation in complete isolation within the cryptographic unit (CU). Its performance is comparable to standard and hardened implementations of AES. Since it does not use any other shared functional units (memory, cache, architectural registers) than those inside the CU for computations involving security sensitive values, it provides a very extensive security for a software implementation. A slight performance degradation when compared to hardened implementation is unavoidable since the latter uses the best of both the CU and the other existing (and speed-optimized) functional units. Since the execution time of the new AES implementation is almost the same as the execution time of the standard implementation, the additional benefit of isolated execution comes almost without much overhead. Another point that needs to be recalled is that the execution time can always be made shorter if highly specialized functional units are designed into the CU. However, this, besides being incompatible with our design principles, would incur high overhead in terms of time and area and would be difficult to integrate to a processor data path.

5.3 A Note on the Trusted/Authenticated Code Execution

With trusted computing it is usually meant that the computer will consistently behave in specific ways, and those behaviors will be enforced by hardware and software[4]. Having noted the adverse criticism to particular implementations of trusted computing, it is essential to assure that the programs execute to their specifications without the outside interference and vulnerability to (active or passive) attacks. Especially in cases where the computation involves sensitive information such as secret keys, the necessity of providing assurance becomes more obvious. Looking at the trusted computing from this angle, we can list two important requirements for trusted execution of computer programs as follows:

- The authenticity of the program instructions and initial (constant) data should be able to be checked dynamically during execution since they are stored in the main memory which is not considered to be a safe place. The memory encryption and authentication mechanisms to protect the entire address space of a program can be a solution to ensure that instructions and process data be not modified by unauthorized parties [9, 18, 10, 41, 40]. However, since the address space of a process contains dynamically changing data elements, memory encryption and authentication can be costly and require some changes to the underlying processor architecture.

[4]The definition is partially taken from Wikipedia to reflect the common perception on the trusted computing.

- For ultimate protection, the execution of a program should be completely isolated from the other programs; either logically or physically. This prevents other processes and agents from gaining access to the sensitive program data and/or residues left on the architectural side channels.

Our proposal of executing cryptographic algorithms in *complete, physical* isolation fulfills the second requirement of trusted computing. The physical process isolation to this extent is naturally neither required nor possible for all types of programs. However, the cryptographic applications that both handles sensitive data and provide a security framework for other applications definitely benefit from physical isolation as proposed here. Once a cryptographic algorithm is executed in complete physical isolation, the only other requirement to fulfill is to authenticate the program instructions and initial data (e.g. input and global variables, configuration parameters). This final requirement can be fulfilled by a lightweight dynamic authentication scheme for static data and instructions of a program [16].

6 The Implementation Results

Adding new instructions and functional units inevitably introduces additional costs in terms of area and time complexity. For an embedded processor, it is essential that the extra hardware cost not exceed the benefits of the enhancements. It is, however, difficult to give a complete perspective on cost-benefit issues when some of the benefits are of qualitative nature. The advantages of the proposed *CU* can indeed be three-fold: i) speed, ii) security, and iii) isolated execution. Not all cryptographic algorithms utilizing the *CU* can benefit from all the advantages. For instance, RSA and elliptic curve cryptography gain up to ten times acceleration since the *CU* is originally designed to support multi-precision arithmetic. AES, on the other hand, can mainly benefit from the latter two advantages: security and isolated execution without increasing the execution time.

Although RSA and ECC can also benefit from these two advantages, the current implementation does not execute RSA and ECC operations in complete isolation since they require a relatively small amount of cache-like on-chip memory to hold sensitive information during the cryptographic calculations. While cryptographic calculations definitely benefit from fast memory accesses via on-chip storage, the particular replacement algorithms is not especially the best option for the cryptographic applications. The topic of on-chip memory for cryptographic computation is an important subject per se, therefore beyond the scope of this work and left as a future work.

We synthesized our design into both ASIC and FPGA target devices using Xtensa LX2 embedded processor core by Tensilica [36]. For ASIC implementation, we used the estimates

Table 12: Hardware Cost of Cryptographic Unit in ASIC Implementation

Area utilization	**without *CU***	**with *CU***
base processor	77,000	77,000
new operations	0	27,195
new state registers	0	10,004
new register files	0	60,496
new functions	0	4,399
total area	77,000	179,094
max clock frequency	270 Mhz	270 Mhz

Table 13: Hardware Cost of Cryptographic Unit in A Implementation

Logic utilization	**without *CU***	**with *CU***
total number of slice registers	9,582	17,939
number of occupied slices	14,363	31,381
total number of four-input LUTs	22,487	49,089
total equivalent gate count	224,014	475,308
max clock frequency	50,914 Mhz	50,594 Mhz

provided by Tensilica tools for $0.13\mu m$ CMOS technology. Table 12 shows the hardware costs of various functional units in terms of equivalent gate count.

The base processor in Table 12 refers to a simple 32-bit embedded processor optimized for ASIC implementation. Note that the figures in the table are estimates and additional units in the *CU* are not optimized for ASIC implementation. Full ASIC realization of reconfigurable Tensilica processors requires further work and the required tools are not available to research community.

There is an area increase for 7-stage pipelined version of both base and enhanced processors. For base processor, the increase is approximately 11.72% while the increase for the CU is 6.82%. Consequently, the increase in the total area of the enhanced processor is 9.31% if it is implemented as a 7-stage pipeline.

We also synthesized the design into FPGA target device, and generated the bit files to program the Avnet LX200 board that features Xilinx Virtex-4 type FPGA. This basically implies that the implementation figures are obtained after placement-and-routing. Timing constraint is chosen as 30 ns as suggested by Tensilica. The implementation results are listed in Table 13.

Since the maximum clock frequency that can be applied to Avnet LX200 boards are specified as 50 Mhz, the proposed cryptographic unit is not on the critical path of processor which is one of our design goals. There is a significant increase in area usage for which we

have several justifications:

- The cryptographic unit provides extensive and multifaceted support for cryptographic algorithm implementations in software. It has its own execution pipeline, functional units, register files, special registers, and more importantly execution semantics that ensure security and isolation.
- The *CU* is not optimized for the FPGA in question. A better analysis of the proposed changes and careful mapping to FPGA resources can decrease the area overhead.
- Increase in area is compensated by the increase in the speedup in certain class of cryptographic algorithms such as ECC and RSA where the time performance is much more important.

In order to clarify the last point, we use `time`×`area` metric to measure the true advantage of the *CU* for ECC and RSA algorithms as far as the time performance is concerned. We choose ECC and RSA algorithms since the unit is primarily designed to accelerate these algorithms. Table 14 provides the `time`×`area` metric of 1024-bit RSA implementation for both no windowing and 4-bit windowing options. The `time`×`area` metric is normalized to the 4-bit windowing implementation of RSA on the processor with the *CU*.

Table 14: Time × Area product for RSA

Core	**Operation**	**Area (Slices)**	**Clock Cycles**	`time` × `area` **(Normalized)**
Base	RSA (4-bit w.)	14,363	132,361,636	**5.15**
With CU	RSA (4-bit w.)	31,381	11,753,299	**1.0**
Base	RSA (no w.)	14,363	156,812,860	**6.11**
With CU	RSA (no w.)	31,381	13,642,547	**1.16**

Table 15 provides the time `time` × `area` values of point multiplication operations for elliptic curve cryptography (ECC). Note that, the `area`×`time` metric is normalized to the 160-bit point multiplication operation on processor with the *CU*.

Finally, Table 16 presents the absolute performance improvements of 1024-bit RSA and ECC point multiplication operation. Except for 160-bit ECC cryptography benefit of the *CU* far exceeds its associated cost.

6.1 Comparing Against Implementations on Other Architectures

In order to give an idea as to how our implementations in the *CU* compares with those on more known and popular architectures, we implemented the Montgomery multiplication and elliptic curve point operations on ARM7TDMI processor. The CIOS version of the

Table 15: Time × Area for ECC

Core	**Operation**	**Area (Slices)**	**Clock Cycles**	`time` × `area` **(Normalized)**
Base	160-bit ECC	14,363	5,684,844	**0.97**
With CU	160-bit ECC	31,381	2,695,097	**1.00**
Base	192-bit ECC	14,363	9,774,069	**1.66**
With CU	192-bit ECC	31,381	3,673,000	**1.36**
Base	256-bit ECC	14,363	21,509,576	**3.65**
With CU	256-bit ECC	31,381	4,412,633	**1.64**
Base	512-bit ECC	14,363	160,109,439	**27.19**
With CU	512-bit ECC	31,381	19,748,196	**7.33**

Table 16: Improvements for RSA and ECC

Operation	**Improvement**
RSA (4-bit w.)	**5.15**
RSA (no w.)	**6.50**
160-bit ECC	**0.97**
192-bit ECC	**1.22**
256-bit ECC	**2.23**
512-bit ECC	**3.71**

Montgomery multiplication operation is written in ARM Assembly language and highly optimized. Since our implementation on Xtensa processor is written in only C language, it is not optimized and there is a room for more improvement as far as the execution time is concerned. As can be observed from Table 17, our implementations of ECC and RSA operations on the *CU* provides significant speedups over those implemented on ARM7TDMI. The precomputation method in the table refers to the fixed-based comb method which uses relatively high amount of lookup tables.

Finally, to give an idea how fast the elliptic curve operations would execute in real applications scenario, we provide the actual times in milliseconds for an FGPA device running on a relatively low clock speed of 50 Mhz. If the precomputation method is used, a 160-bit EC point multiplication operation takes about 6.86 ms in the *CU* implemented on Xtensa LX2 processor.

Table 17: Comparison with ARM7TDMI

Operation	**Assembly on ARM7TDMI**	**C on the *CU* @ XTensa**	**Improvement**
160-bit Mont. Mult.	1,464	1,047	1.40
192-bit Mont. Mult.	2,008	1,196	1.68
256-bit Mont. Mult.	3,384	931	3.63
1024-bit Mont. Mult.	45,600	7,654	5.96
1024-bit RSA	70,041,600 (est.)	11,753,299	5.96
160-bit ECC Point Mult.	3,554,000	2,695,097	1.32
192-bit ECC Point Mult.	5,688,000	3,673,000	1.55
256-bit ECC Point Mult.	11,696,000	4,412,634	2.65
512-bit ECC Point Mult.	84,352,000	19,798,723	4.26
160-bit ECC Point Mult. with precomputation	688,000	342,901	2.01
192-bit ECC Point Mult. with precomputation	1,096,000	462,840	2.37
256-bit ECC Point Mult. with precomputation	2,320,000	545,860	4.25
512-bit ECC Point Mult. with precomputation	15,448,000	2,409,272	6.41

6.2 A Note on AES Time Performance and Comparison with Other Implementations

As pointed out earlier, the speed is not the main design goal for AES implementation; hence we only add instructions to better exploit the *CU* for secure and isolated execution of AES. From another standpoint, however, the implementation should also give a decent time performance. In order to evaluate time performance of our AES implementation, we enumerate the time performances of other state-of-the art AES implementations in Table 18.

Time performances of different implementations differ due to several reasons. Firstly, different processor architectures can render certain optimizations easier. Secondly, whether the implementation is secure against cache attacks incur overhead in cycle count (e.g. [27] utilizes the bit-slicing technique to protect AES against cache attacks). Thirdly, the types and extent of the hardware support play a very decisive role in time performance. For instance, the implementation in [19] combines three hardware-based techniques to accelerate the AES implementation to a great extent. Our implementation, even though not providing the best time performance, does not suffer a considerable deterioration in speed. Further-

Table 18: Comparison of AES Implementations

Implementation	**Hardware support**	Performance (cycles)
[5] on ARM7TDMI	-	1675
[27] on AMD Opteron	-	2699
[19] on CRISP	Bit-sliced + lookup tables	2203
[19] on CRISP	Bit-sliced + lookup tables + bit-level permutation	1222
this work	on cryptographic unit	2620

more, if the same hardware-based techniques such as bit permutation are applied in our design, a considerable speedup would be obtained.

Comparing our design to another scheme [28] that provides a general support for process isolation can be beneficial in assessing some qualitative aspects of our proposal. As mentioned earlier, Flicker scheme in [28] allows any process to run in complete isolation from all other simultaneously running processes; hence giving a high level of security to applications. However, since it utilizes an available TPM chip in the system, the applications may observe significant delays due to the incurred overhead in communication. In addition, when a process in isolation is running, the other applications basically suspend and all the interrupts are disabled. Not only the computer becomes unresponsive to user inputs (keyboard inputs, mouse movements), but also there may be packet losses as well. Since the Flicker sessions cannot be too long, the overhead cannot be amortized and long sessions for bulk encryption cannot be supported.

The cryptographic unit in our proposal provides support only for cryptographic applications. There is basically no overhead in execution time compared to other schemes and cryptographic computation can run as long as necessary without suspending the other applications sharing the same processor. Naturally, only one cryptographic operation can be actively running at a given time and the access of all other processes to the CU must be prevented during the execution of a cryptographic process.

7 Conclusion and Future Work

We designed and implemented a cryptographic unit (CU), for secure and fast execution of a wide range of cryptographic algorithms, which can be integrated into any RISC processor architecture. To show the design's efficiency and applicability, we integrated the proposed CU into the execution pipeline of a low cost, extensible, embedded processor core. We obtained considerable speedups for basic multi-precision arithmetic operations such as modular

multiplication and inversion that are the dominant operations in many public key cryptosystems. We found out that the speedup values gained through the *CU* for ECC and RSA, are up to 8 and 10 times, respectively. We demonstrated that the *CU* can also be used to harden software implementations of symmetric ciphers with low overhead against certain side-channel attacks (i.e. cache attacks). We also established that the hardware overhead of the proposed *CU* in terms of chip area is acceptable even for embedded processors. A comparison of the obtained speedup values and incurred hardware overhead clearly confirms that the benefits of the *CU* far exceed its cost.

The main structure of the CU resembles a integer unit of a RISC-style general-purpose processor and its interface is the same as a typical RISC ISA. For instance, instructions are register-based, and take at most three operands. In order to demonstrate that the CU can provide a similar benefits for different implementations of a RISC processor, we integrated it to both 5-stage and 7-stage pipelined versions of the same embedded processor. Although there is a slight degradation in area and speed (in terms of clock cycles count) for 7-stage pipeline, which is expected, the degradation is less in the CU than in the other part of the processor. This clearly shows that the CU design is indeed generic and especially beneficial for cryptographic operations.

We realized the enhanced processor on an FPGA device and determined that the CU unit is not in the critical path of the processor; therefore we were able to achieve the maximum clock frequency of 50 MHz for the target device.

We demonstrated, in the particular case of the AES, that a cryptographic algorithm can be executed inside the CU in complete isolation from the other processes/threads simultaneously running on the same processor. Process isolation has been on the agenda of several major processor manufacturers and some processors provide hardware support when the need arises. Our approach, similar in concept but different in approach, provides this property for only cryptographic algorithms with virtually no performance degradation. Combined with dynamic code authentication, the CU is instrumental to the trusted computing.

References

[1] Onur Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *LNCS*, pages 112–121. Springer Verlag, Berlin, Germany, 2006.

[2] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache-Based Remote Timing Attacks on the AES. In MAsayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 271–286. Springer Verlag, Berlin, Germany, 2007.

[3] ARM. *TrustZone Technology Overview*. `http://www.arm.com/products/security/trustzone/`.

[4] P. Barreto. The AES Block Cipher in C++. Website, 2003. `http://planeta.terra.com.br/informatica/`.

[5] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient software implementation of aes on 32-bit platforms. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 159–171. Springer, 2002.

[6] Johannes Blömer and Volker Krummel. Analysis of Countermeasures Against Access Driven Cache Attacks on AES. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *LNCS*, pages 96–109. Springer, 2007.

[7] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *TRUST*, volume 4968 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2008.

[8] Liqun Chen, Paul Morrissey, and Nigel P. Smart. Pairings in trusted computing. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing*, volume 5209 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2008.

[9] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207. Springer-Verlag, 2003.

[10] Dwaine E. Clarke, G. Edward Suh, Blaise Gassend, Ajay Sudan, Marten van Dijk, and Srinivas Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE Symposium on Security and Privacy*, pages 139–153. IEEE Computer Society, 2005.

[11] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In Kazuo Ohta and Dingyi Pei, editors, *ASIACRYPT*, volume 1514 of *LNCS*, pages 51–65. Springer, 1998.

[12] Jean-Sébastien Coron, David Naccache, and Paul C. Kocher. Statistics and secret leakage. *ACM Trans. Embedded Comput. Syst.*, 3(3):492–508, 2004.

[13] Intel Corporation. *LeGrande technology preliminary architecture specification.* Intel Publication no. D52212, May 2006.

[14] D. Bernstein. Cache-Timing Attacks on AES. Website, 2005. `http://cr.yp.to/papers.html#cachetiming`.

[15] Advanced Micro Devices. *AMD64 virtualization: Secure virtual machine architecture manual.* AMD Publication no. 33047 rev. 3.01, May 2005.

[16] A. O. Durahim, E. Savas, T. B. Pedersen, B. Sunar, and O. Kocabas. Transparent code authentication at the processor level. IET Computers and Digital Techniques, to appear, 2009.

[17] A. Murat Fiskiran and Ruby B. Lee. On-Chip Lookup Tables for Fast Symmetric-Key Encryption. In *ASAP*, pages 356–363. IEEE Computer Society, 2005.

[18] Blaise Gassend, G. Edward Suh, Dwaine E. Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity. In *Proceedings of Ninth International Symposium of High Performance Computer Architecture (HPCA 2003)*, pages 295–306, February 2003.

[19] Philipp Grabher, Johann Großschädl, and Dan Page. Light-weight instruction set extensions for bit-sliced cryptography. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2008.

[20] Johann Großschädl and Erkay Savas. Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *LNCS*, pages 133–147. Springer, 2004.

[21] Johann Großschädl, Stefan Tillich, and Alexander Szekely. Performance Evaluation of Instruction Set Extensions for Long Integer Modular Arithmetic on a SPARC V8 Processor. In *DSD*, pages 680–689. IEEE, 2007.

[22] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc. 17th USENIX Security Symposium (Sec '08), San Jose, CA*, July 2008.

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

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

[25] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[26] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[27] Robert Könighofer. A fast and cache-timing resistant implementation of the aes. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2008.

[28] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In Joseph S. Sventek and Steven Hand, editors, *EuroSys*, pages 315–328. ACM, 2008.

[29] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go?: recommendations for hardware-supported minimal tcb code execution. In Susan J. Eggers and James R. Larus, editors, *ASPLOS*, pages 14–25. ACM, 2008.

[30] P. L. Montgomery. Modular multiplication without trial division. *Math. Computation*, 44(170):519–521, April 1985.

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

[32] Elisabeth Oswald and Stefan Mangard. Template attacks on masking - resistance is futile. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 2007.

[33] Övünç Kocabaş, Erkay Savaş, and Johann Groβschädl. Enhancing an embedded processor core with a cryptographic unit for speed and security. In *RECONFIG '08: Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, pages 409–414, Washington, DC, USA, 2008. IEEE Computer Society.

[34] D. Page. Partitioned cache architecture as a side channel defence mechanism. Cryptography ePrint Archive, Report 2005/280, August, 2005. `citeseer.ist.psu.edu/page05partitioned.html`.

[35] Srivaths Ravi, Anand Raghunathan, Paul C. Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embedded Comput. Syst.*, 3(3):461–491, 2004.

[36] Tensilica. Xtensa LX2 Embedded Processor Core. Website. `http://www.tensilica.com/products/xtensa_LX2.htm`.

[37] Stefan Tillich and Johann Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *LNCS*, pages 270–284. Springer, 2006.

[38] Trusted Computing Group, Incorporated. *TCG Software Stack (TSS), Specification Version 1.2, Level 1. Part1: Commands and Structures*, January 6 2006. `https://www.trustedcomputinggroup.org/specs/TSS/TSS_Version_1.2_Level_1_FINAL.pdf`.

[39] Tobias Vejda, Dan Page, and Johann Großschädl. Instruction Set Extensions for Pairing-Based Cryptography. In Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors, *Pairing*, volume 4575 of *LNCS*, pages 208–224. Springer, 2007.

[40] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. In *ISCA*, pages 179–190. IEEE Computer Society, 2006.

[41] Jun Yang, Lan Gao, and Youtao Zhang. Improving memory encryption performance in secure processors. *IEEE Trans. Computers*, 54(5):630–640, 2005.