OXFORD UNIVERSITY PRESS The Computer Journal

Enhancing an Embedded Processor Core for Efficient and Isolated Execution of Cryptographic Algorithms

Journal:	*The Computer Journal*
Manuscript ID:	COMPJ-2013-05-0284.R2
Manuscript Type:	Original Article
Date Submitted by the Author:	n/a
Complete List of Authors:	Savas, Erkay; Sabanci University, Yumbul, Kazim; ASELSAN,
Key Words:	Cryptographic Unit, Instruction Set Extension, FPGA, Cryptography, Attacks

Enhancing an Embedded Processor Core for Efficient and Isolated Execution of Cryptographic Algorithms *

Kazim Yumbul, Erkay Savaş

Sabanci University

Orhanli, Tuzla Istanbul, 34956 Turkey

kyumbul@gmail.com, erkays@sabanciuniv.edu

February 27, 2014

Abstract

We propose enhancing a reconfigurable and extensible embedded RISC processor core with a *protected zone* for isolated execution of cryptographic algorithms. The protected zone is a collection of processor subsystems such as functional units optimized for high-speed execution of integer operations, a small amount of local memory for storing sensitive data during cryptographic computations, and special-purpose and cryptographic registers to execute instructions securely. We outline the principles for secure software implementations of cryptographic algorithms in a processor equipped with the proposed protected zone. We demonstrate the efficiency and effectiveness of our proposed zone by implementing the most-commonly used cryptographic algorithms in the protected zone; namely RSA, elliptic curve cryptography, pairing-based cryptography, AES block cipher, and SHA-1 and SHA-256 cryptographic hash functions. In terms of time efficiency, our software implementations of cryptographic algorithms running on the enhanced core compare favorably with equivalent software implementations on similar processors reported in the literature. The protected zone is designed in such a modular fashion that it can easily be integrated into any RISC processor. The proposed enhancements for the protected

*Preliminary version of this paper has been presented at Reconfig 2009 [1].

zone are realized on an FPGA device. The implementation results on the FPGA confirm that its area overhead is relatively moderate in the sense that it can be used in many embedded processors. Finally, the protected zone is useful against cold-boot and micro-architectural side-channel attacks such as cache-based and branch prediction attacks.

Keywords: Cryptography, Cryptographic Unit, Isolated Execution, Instruction Set Extension, Secure Computing, Attacks.

1 Introduction

Secure and efficient implementations of cryptographic algorithms have become more of a focal point for research in cryptographic engineering since various attacks [2, 3, 4, 5] (i.e., timing, power analysis, fault, and branch prediction attacks, respectively) successfully compromise *realizations* of many cryptosystems which are believed to be secure under computational or similar assumptions in theory. Since general-purpose processors fulfill neither the timing nor the security constraints of cryptographic applications due to different set of design considerations, special-purpose cryptographic co-processors are built to remedy this problematic. Nevertheless, cryptographic co-processors turn out to be not entirely free from security concerns and furthermore, introduce their own problems such as security risks and speed considerations accrued in host processor/co-processor setting.

Aware of inadequacy of software-only solutions, computer manufacturers already introduced hardware extensions to their processor cores to accelerate cryptographic computations such as Intel's AES instruction set [6] and to provide secure execution environment. A notable development is that new architectures introduced by three major manufacturers [7, 8, 9] allow that security-sensitive applications execute in an environment strictly free from the intervention of other simultaneously running processes. This feature is known as process isolation and enforced by the hardware. A strictly enforced process isolation is definitely beneficial in thwarting an important class of attacks known as micro-architectural side-channel attacks [10, 5]. However, without hardware support many practical attacks [5, 11, 4, 10, 2, 3, 12], in fact, cannot be easily prevented by software-only countermeasures. This is such a general understanding that many counter-measures

proposed in the literature [13, 14, 15, 16, 17, 18] are implemented below the software level. From these observations, developments and results, the need for further research in new computer architectures that support efficient and secure implementations of cryptographic algorithms becomes obvious.

In this paper, we investigate the realization of a protected zone in a reconfigurable embedded processor that provides cryptographic algorithms with a highly secure execution environment. The protected zone consists of architectural subsystems of a local memory, registers, and functional units and enables a much more strict process isolation in the sense that sensitive information never leaks outside the zone. For acceleration of basic arithmetic operations, we use and improve the design principles presented in [19, 20]. The goal of accelerating many cryptographic algorithms (AES, hash functions, elliptic curve cryptography, RSA, Pairing-based cryptography) has a major influence on the design of the functional units and the organization of the protected zone. For instance, the functional units are designed to enable fast modular arithmetic for numbers in the range of [160, 2048] bits (which are the precisions used in the public key algorithms; i.e., elliptic curve cryptography, RSA, Pairing-based cryptography) without an unacceptable increase in chip area and any decrease in clock frequency. Similarly, an optimum number of cryptographic registers and an optimum amount of local memory are determined to accelerate these public key algorithms.

The organization of the protected zone is highly modular, and complies with the design principles of RISC processors, therefore, it can be incorporated into any RISC processor. We also demonstrate that well-known cryptographic algorithms, RSA, ECC, pairing-based cryptography, AES, SHA-1, and SHA-256 can be implemented on an embedded processor equipped with the protected zone with superior time performance, efficiency, and high-level security. A similar approach is used in [21] only for AES implementation; however the proposed technique in [21] cannot easily be extended to more complicated public key algorithms. We provide a complete, generic approach that is readily applicable to any cryptographic algorithm and does not necessitate Assembly language implementation, which is essential in [21].

The rest of the paper is organized as follows: We summarize the related work and our contri-

butions in Section 2. Section 3 outlines our methodology used in the design of the protected zone. Section 4 introduces the general architecture and how the protected zone is incorporated into a classic embedded RISC processor. A small amount of local memory, which is an essential part of the protected zone is explained in Section 5. In software implementations of cryptographic algorithms, general-purpose registers, which are not a part of the protected zone but the base processor core, need to be used. We show how to use these general-purpose registers securely in Section 6. Section 7 presents the new instructions, which are realized on the protected zone and useful in secure execution of certain operations in cryptographic applications. The timing results for the software implementations of major cryptographic algorithms are given, when they are implemented on our processor enhanced with the protected zone, in Section 8. The implementation results of the protected zone and the base processor core on FPGA and ASIC are presented in Section 9. The paper is concluded in Section 10 by summarizing the achievements.

2 Related Work and Our Contribution

While modifying microprocessor architectures for speed and security is a common method in the literature, the objectives and approaches in particular solutions differ. The works in [13], [15], [17], [22], and [18] aim to provide a secure computing environment for protecting all applications and their data against software and hardware attacks. In [13] and [17] the internal state of the processor is protected. In [15], the Secret-Protecting (SP) architecture ensures that all data and codes of a software module are encrypted when they are off the chip. The secure processor architecture in [22] protects a trusted hypervisor, which protects other trusted software modules. Finally, the architecture in [18] adopts a secure processor model, where CPU core and cache are protected using encryption and memory integrity verification modules. The model envisages that the computing system is divided into two parts: i) trusted on-chip modules (e.g., CPU core, cache memory, registers, encryption/decryption engine, and memory integrity verification module) and ii) untrusted off-chip modules (external memory and external peripherals). Any data that goes out of the chip

is encrypted and any data coming from the off-chip modules into the trusted chip is verified for tamper resistance. The architecture features an AES engine and a true random number generator as cryptographic units.

Other architectures modify or extend processor cores mostly for accelerating the cryptographic algorithms [23], [24]. There are also other architectures, which propose solutions for both secure and fast execution of cryptographic algorithms [19], [21], [1] and [20]. A recent work in [24] explores design possibilities for accelerating cryptographic algorithms while secure execution is not considered. We compare our architecture and that in [24] in terms of the execution speed of cryptographic algorithms and the relative overhead of the processor extensions.

The approach in our previous work in [1], which also provides a generic support for many cryptographic algorithms for speed and security, is the closest to the approach adopted in this work. While the work in [1], which presents our preliminary results, introduces the essentials of our approach, this work provides substantially new contributions, which can be summarized as follows:

- We introduce a novel secure table lookup technique that benefits the s-box computation in all block ciphers. We implement the AES algorithm using this new technique; and the new implementation outperforms the implementation in [1] by a large margin.
- Besides RSA and elliptic curve cryptography, we implement the Tate pairing operation for pairing-based cryptographic protocols, which are common in many security protocols. Our implementation outperforms a comparable implementation on an extended embedded processor in [25] by about a factor of 2. The software implementations of many new arithmetic operations in different algebraic fields required for Pairing-based cryptography are also obtained.
- We show how to implement a cryptographic hash function securely in our new processor. We implement SHA-1 and SHA-256 (256-bit version of SHA-2) algorithms using the secure execution principles and list the performance results.
- We include implementation details for new instructions for secure execution of cryptographic

algorithms and a rationale behind them, which are omitted in [1] due to space considerations.

- We demonstrate as to how to use general-purpose registers, which are already available in the base processor in a secure manner in cryptographic computations.

- We introduce the use of a local memory for secure execution and show that only a small amount of local memory is sufficient for the implementations of a wide range of cryptographic algorithms. We also include some remarks about the feasibility of implementing it on-chip in an embedded microprocessor.

- Developing secure implementation of cryptographic algorithms requires neither advanced Assembly programming nor expert level knowledge in micro-architectural details of the processor. The development can be done in high-level languages (C and C++ in our case) with only a minor exception where a couple of inline Assembly statements are added to track the use of general-purpose registers for sensitive data.

- Optimizations are performed to improve the performance and the efficiency of the proposed architecture. One important example is in the reduction of the number of the cryptographic registers. In our earlier design, there are 32 cryptographic registers. Now, the current design uses only 8 cryptographic registers, which results in a considerable reduction in area. We show that reducing the number of the cryptographic registers did not have any adverse effect on the time performance of cryptographic algorithms for considerably large key sizes. We also reduce the number of the predicate registers from two to one, used in the protection against branch prediction attacks.

This work is a crucial step in an attempt to build a secure architecture for the execution of cryptographic algorithms. Since it essentially proposes an isolated execution space for cryptographic algorithms, it can provide protection against a wide range of attacks when combined with other types of countermeasures in the literature. For instance, arithmetic codes in [26, 27] can be used in the design of the functional units in our architecture to protect a wide range of public key

cryptography algorithms against fault attacks. Similarly, countermeasures at logic gate-level as proposed in [14], when applied in our protected zone, will provide protection against differential power analysis attacks. After all, the protected zone can be implemented in a chip area, on which various gate-level countermeasures can be applied effectively. Since no secret or sensitive data leaves the zone, they can be protected against attacks such as side channel and fault attacks.

3 Principles and Requirements of Secure and Isolated Execution

Software implementations of cryptographic algorithms are vulnerable to various forms of attacks that can be grouped into two main classes: side-channel [2, 3, 5] and fault-injection attacks [4]. The side-channel attack in [2] takes advantage of key-dependent variations in execution times of cryptographic algorithms while the differential power analysis attack in [3] utilizes variations in power usage during cryptographic computations. The side-channel attack in [5] is a timing attack using the time variations due to branch mispredictions. In the second category, fault-injection attacks [4] utilize incorrect outputs of a cryptographic algorithm due to faults deliberately induced by an adversary to find out the secret key. Different countermeasures from circuit- [14] through architectural- [16] to algorithmic-levels [28] have been proposed. It has, however, been well-understood that ultimate protection against all kinds of attacks seems to be difficult and different counter-measures need to be deployed against different attacks for reasonably secure implementations of cryptographic algorithms.

In this paper, we deal with mainly *architecture-level* attacks and countermeasures. In particular, the proposed countermeasures provide resilience against cache-based, branch prediction, and to some extent simple power analysis attacks. Therefore, countermeasures proposed for lower levels (e.g., circuit level) against fault analysis and differential power analysis attacks are beyond the scope of this work. We emphasize that a good protection against known attacks (mainly side-channel and fault induction attacks against hardware and software implementations of cryptographic algorithms) should combine countermeasures for all levels; i.e. from circuit through architecture (as in our

approach) to algorithm levels. Notwithstanding, our proposed architecture can still be useful in the implementation of algorithmic countermeasures as shown in Section 8 such as a countermeasure for simple power analysis, secure implementation of cryptographic hash functions, and secure lookup tables. Moreover, circuit level countermeasures can be built into the functional units such as the multiplier, which definitely broadens the variety of attacks, for which the proposed architecture provides protection.

A preeminent example of architecture-level attacks is cache-based attacks. Many cryptographic algorithms utilize lookup tables for fast execution, which makes them vulnerable to cache-based side-channel attacks [10]. Another form of side-channel attacks that utilizes processor micro-architecture is named as branch prediction attacks [5]. The main reason that these attacks are effective is the fact that a majority of general-purpose processors (including many embedded processors) support multi-tasking and resource sharing as in the cases of cache memories, branch prediction and target buffers. The processes running simultaneously cannot directly access each other's data since the operating system enforces process isolation. However, processes inadvertently (and inevitably to a certain degree) leave *residue* data in shared resources (cache memories and branch buffers). Another process cannot directly use or learn the residue data; however, it can make inferences through carefully timed accesses to these shared resources. The residue data in shared resources does not have to be secret or confidential per se; but its presence may say something about the secret that is used to access it. Naturally, during the execution of cryptographic algorithms, secret keys are used to access lookup tables (hence cache attacks) and to make decisions in the program execution flow (hence branch prediction attacks).

Worse yet, the bugs and flaws in operating systems (OS) render OS-implemented process isolation ineffective against sophisticated attacks that allow ill-intentioned programs to gain access to secret information through the violation of process isolation. This situation calls for a much stronger, and inevitably hardware-based, mechanism for process isolation. Supporting this claim, major processor manufacturers such as Intel, AMD, and ARM, introduced extensions to their processor cores to fortify the process isolation [7, 8, 9]. The basic principle is to make certain parts of

memory, of cache, and of TLB used by a process strictly inaccessible by other processes. However, the isolation is still virtual rather than physical since the data from different processes still occupy the shared resources. For instance, the confidential data such as secret keys and temporary variables will still be present in physical memory in certain points of execution. Recently demonstrated cold-boot attacks [12] efficiently recover secret keys used in cryptographic operations.

Therefore, to provide an even stronger type of process isolation where the cryptographic algorithms execute free from vulnerabilities against the aforementioned attacks, the processor architecture needs to provide support for keeping all confidential information in physically protected zones. Confidential information not only includes secret keys, but all intermediate values obtained during cryptographic computations. For example, an AES block in an intermediate round is also confidential since its compromise may reveal important information on the secret key[1]. Similarly, an intermediate elliptic curve point obtained during elliptic curve scalar multiplication needs to be protected, since it is possibly a smaller multiple of the base point which gives away certain bits of the secret integer (possibly the private key). Therefore, there is a need for a protected zone where we can keep the confidential information before, during, and after the cryptographic computation. The protected zone includes functional units, a small, protected local memory, a cryptographic register file that we can use during operations, and some special registers to keep intermediate variables. In what follows, we explain the components of the protected zone.

- **Functional units** execute the instructions needed in cryptographic computations, which basically implement simple arithmetic/logic operations. Some operations are needed for secure execution of cryptographic algorithms to prevent branch prediction attacks as well as to avoid confidential variables appearing in general-purpose registers of the processor.

- **Local memory** is used to implement a scratch pad for temporary variables and lookup tables as well as to keep secret keys. The local memory can be implemented either on-chip or

[1]The compromise of an intermediate AES block is equivalent to using AES with fewer number of rounds than specified. It is well-known that many block ciphers are shown to be weak if they are executed with fewer number of rounds.

outside of the chip; but the important feature is that it is physically protected and not a part of the memory hierarchy to avoid it from being backed up on higher levels of the hierarchy. Its implementation is much easier than a cache memory since placement scheme is straightforward. The cache memory usage is always problematic in cryptographic algorithms and not necessarily as beneficial as a possible local memory. Some commercially available processors such as graphic processors and the Cell Broadband Engine Architecture (CBEA) [29] also feature local memories. It is important to note that the SPE cores (Synergistic Processing Elements) in CBEA use on-chip local memories in isolation.

- **Cryptographic Registers** are organized as a register file, from which the functional units can read their operands. The confidential values (secret keys and sensitive temporary values) are kept and operated on while they are in these registers. Important feature of these registers is that they are not spilled onto the main memory but to the local memory.

- **Special registers** are used to keep some temporary values during the long-latency cryptographic computations such as multi-precision modular multiplication and block cipher round operations.

In the next section, we provide more details about the processor architecture that incorporates such a protected execution zone for cryptographic operations.

4 General Architecture

The architecture in Figure 1 is proposed to fulfill the requirements of secure and isolated execution of cryptographic algorithms stated in Section 3. The base architecture is essentially a 32-bit embedded processor core based on Xtensa LX3 architecture by Tensilica [30] that provides the most basic integer functionality. The architecture is both reconfigurable and extensible. A basic pipeline structure with five stages, a register file of 32 32-bit registers and a simple ALU are default resources in what is referred as the *base* architecture whose components are shown in dark in Fig-

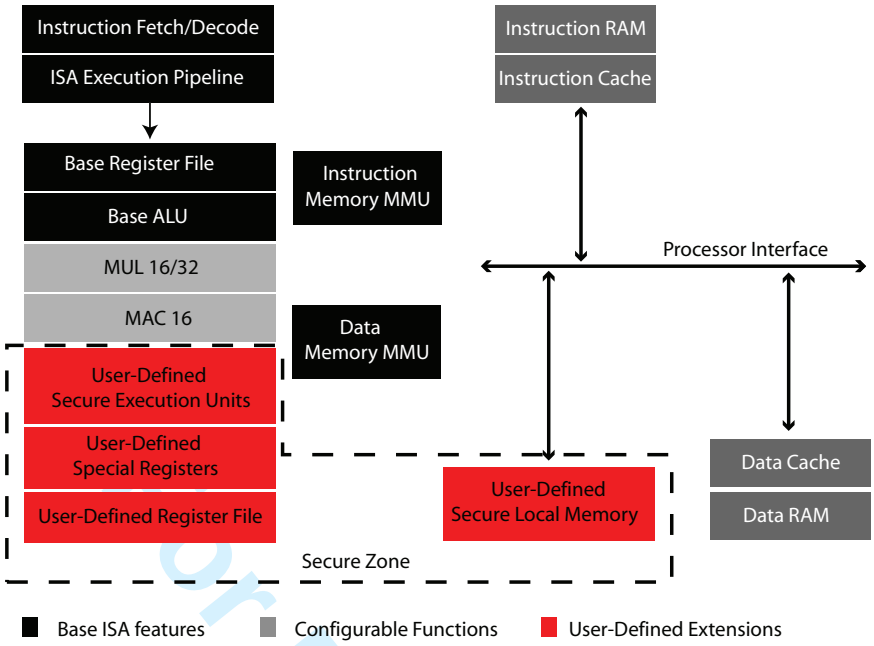

Figure 1: General Architecture

ure 1. The resources shown in the lightest represent configurable parts, which simply means that a developer/designer can choose to add/remove/configure units already available in the Xtensa LX3 architecture. For instance, a 16- or 32-bit multiplier and a multiply-and-accumulate unit (MUL16/32 and MAC 16 in Figure 1, respectively) can be added to the base architecture. The cache memory size and configuration can also be determined by the designer/developer.

The architecture is extensible in the sense that the designer can add units of her/his own design such as multi-cycle execution units, register files, special registers for multi-cycle instructions, even make the basic RISC pipeline into a multi-issue VLIW processor. It is the extensibility feature that we use to realize our protected zone to execute cryptographic operations as illustrated in Figure 1 (enclosed within the dashed area).

Figure 2 shows the details of the protected zone where we can perform cryptographic operations safely. The organization of the zone is very similar to an ordinary RISC processor core with the exception of the 128-bit data path and the block cipher unit. The register file consists of eight 128-

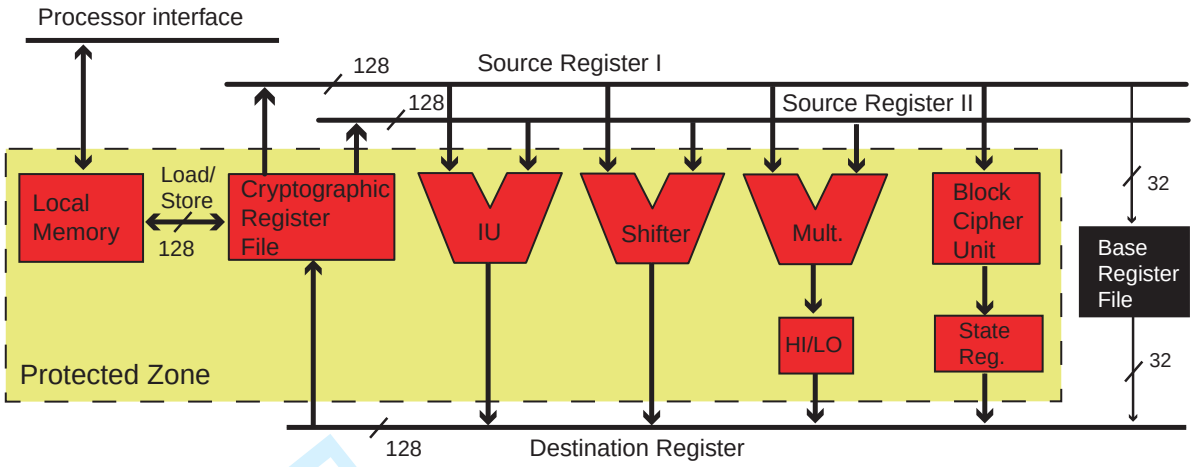

Figure 2: Organization of Protected Zone

bit registers, which we refer as *cryptographic registers* henceforth, and are used to hold operands during the computation. The execution units, namely integer unit (IU), shifter, and multiplier, are responsible for executing arithmetic/logic operations common in cryptographic computations in an efficient and secure manner. While 128-bit shift and arithmetic/logic operations are single-cycle, the 128×128 multiplication is a multi-cycle operation. For the details of these instructions, see [19, 20].

The block cipher unit (BCU) is novel in this design and incorporates various operations common in many block cipher algorithms. In the beginning of each round of a block cipher algorithm, the block is in one (or more depending on the block length) of the cryptographic registers. Once the round starts, the block is first transferred into special registers in the BCU. One of the important operations performed in the BCU is index calculation for *secure* table lookup operation that is employed in many block cipher implementations to accelerate s-box computation. The lookup table is formed inside the local memory in order to avoid cache-based side-channel attacks.

While accessing the lookup tables, most RISC-based processors use architectural (general-purpose) base registers to compute the address of the location of the desired data, which may be directly related to the secret. However, the architectural registers are not safe places to keep

confidential information since they are backed up on the main memory (*register spilling*); a process that may leak secret information. Therefore, they must be used carefully. A straightforward approach is to reset the architectural register used to keep confidential data after they are no longer needed and before they are spilled to the main memory, which is easy to do in Assembly programming. However, this is not an easy task in higher level language implementations since it is up to the compiler to decide which registers are used in address calculation, which is hard to predict beforehand for software developers. We use basically two techniques to reset architectural registers' secret content, using high level language constructs that allow inline assembly instructions and defining local variables on specified registers. This way, it is easy to keep track of registers that are used to handle sensitive information to reset them afterward.

Note that certain operations either take multi-cycle or multi-instruction to complete, therefore temporary values are kept in special-purpose registers. This resembles the multiplication operation common in RISC processors that puts the high and low parts of the result in two special-purpose registers, namely `HI` and `LO`, respectively. In order to further operate on the result of a multiplication, instructions such as `mfhi` and `mflo` are used to move the results to the general-purpose registers. We adopt the same approach; however, it is required that these special-purpose registers be not saved in the main memory before process switch operation, which is supervised by the operating system. Thus, operating system support is necessary in secure and isolated execution to time carefully the context switching in order not to lose data.

5 Local Memory

As mentioned before, we propose to use a small amount of non-cached local memory as a scratch pad. While the local memory can be implemented as off-chip as well as on-chip memory, it is preferable to implement it as an on-chip memory since this way it will be much easier to protect it against threats such as cold-boot attacks [12]. Furthermore, an on-chip local memory is faster. Using a local on-chip, non-cached memory for protection of processes is not new and is already em-

ployed in the Cell Broadband Engine Architecture (CBEA) [29] used in PlayStation game consoles by Sony and BladeServers by IBM.

The local memory is accessed the same way as the main memory, whereby a memory address is processed by a memory management unit. The memory management unit simply treats any address in the address range of local memory as a special memory access and sends it to the local memory. Since the local memory may contain sensitive data belonging in ongoing cryptographic computation, its content must be protected on context switch where the operating system schedules another process to run. We identify four methods that can be used to achieve the protection of sensitive data in local memory. The first and most straightforward approach is to erase its content on every context switch, which may raise efficiency concerns. The second method can be a partitioning technique to allocate different parts of the local memory to different processes, whereby usage is enforced by checking process identifiers. This method can require an increase in the size of the local memory to accommodate the space requirements of every active process. The third method can be to give exclusive usage of local memory to a single process at a given time. This method alleviates the efficiency and size concerns while it deprives the other processes of the local memory. The last method is that only a privileged process can use the local memory and all others are not allowed to use it. Depending on the implementation and the usage scenario, one of the proposed methods can be adopted. We do not specify a preference here.

Compilers use registers and memory (stack or heap) to store variables used in a computer program. When possible, variables are kept in registers for fast execution of instructions. The register contents are *spilled* to main memory when the compiler runs out of registers, which are limited in number. The mapping between registers and memory location is not fixed for a variable and hence every time it is accessed a different register can be used for the same variable. In our processor, we utilize the properties of Tensilica architecture and the associated tool chain to establish a mapping between the memory locations and cryptographic registers for sensitive data processing.

When a variable is declared to keep sensitive data, it is defined as a new data type which is

mapped to a cryptographic register. This basically means that whenever a variable of this new data type is to be processed it is placed in one of the cryptographic registers. If the variable is of array type, more than one cryptographic registers are used for this purpose. Since we have only eight cryptographic registers in our cryptographic register file, they are also subject to register spilling. Therefore, we need to force this spilling to use a predefined memory location in our local memory for security purpose. Otherwise, the contents of the cryptographic registers would be written to any location in main memory, which forms a security risk under our threat model. Fortunately, Tensilica tools allow to initialize a variable of a new type based on the cryptographic register file by pointer assignment to a pre-initialized memory location as follows:

```
unsigned long modulus_data[32];
crypto_register *modulus = (crypto_register *) modulus_data;
```

Here, `crypto_register` is a new data type associated with the cryptographic register file integrated to the processor core as follows:

```
regfile crypto_register 128 8 cr.
```

Also, the keywords (`regfile 128 8`) indicate that we extend the core processor with a new cryptographic register file that contains eight 128-bit registers. The keyword `cr` is an internal name used for the individual registers; namely compiler accesses these registers as `cr0`, `cr1`, ... `cr7`.

To summarize, we keep sensitive data in the cryptographic registers that are mapped to locations in our local memory. The sensitive data kept in variables of new type is normally stored in the local memory. They are placed in the cryptographic registers before processing and when the processing is finished or when the system runs short of cryptographic registers they are moved (*spilled*) to the same location in the local memory. Since both the local memory and the cryptographic registers are protected under our assumptions, no leakage occurs for sensitive data.

Since the tools provided by Tensilica do not allow to realize the actual local memory, in our implementations we employ a part of the global memory address space to simulate the local memory. Therefore, it is, at this point, imperative to discuss the feasibility of realizing on-chip local

memory, especially for embedded processors. The non-cached local memory for each processor core is implemented in CBEA processors [29], which has eight cores each featuring 256 KB (kilobyte) of on-chip local memory. When run in so-called *isolated mode*, a process running on a core completely isolates itself from the system (main memory, system bus, other cores, etc.) and relies only on local memory for data and instructions. The example of the CBEA processor clearly shows that 256 KB $\times$ 8 = 2 MB of on-chip memory is feasible to implement on high-end processors.

Nevertheless, our architecture is proposed also for embedded applications and therefore, we need to develop a deeper insight to the cost of an on-chip local memory in embedded processors. Apparently, large on-chip memories cannot be supported in embedded processor due the limited budget in chip area and power dissipation. Fortunately, as we explain in subsequent sections the cryptographic algorithms we implement require a surprisingly small amount of local memory. Even 2048-bit RSA algorithm, which is the most memory-intensive implementation in our experiments, needs only about 5,700 B of local memory at most. Therefore, we basically estimate that about 10 KB of local memory is sufficient for many symmetric and asymmetric algorithms in use today if expensive precomputation techniques are *not* used for acceleration.

Considering one bit of SRAM memory takes about 6-10 transistors to manufacture, 5,700 B of memory requires 5700 B $\times$ 6(10) = 34,200 (57,000) transistors. Considering also that a NAND gate, the number of which is used as a metric called gate-equivalent (GE) to estimate the area complexity of a design on ASIC, requires four transistors, a local memory of 5,700 B expectedly takes as much space as a circuit of 8,550 (14,250) GE on ASIC. Note that this is a rough estimate, thus real figures can only be given after actual implementations. All the same, this simple analysis shows that realizing a small amount of on-chip memory does not require prohibitively high amount of chip space.

6 Secure Use of General-Purpose Registers

In software implementations of cryptographic algorithms, general-purpose registers available to programmers are needed for various purposes. For example, addresses in memory access instructions are usually calculated and kept in general-purpose registers. General-purpose registers are, in general, not secure locations since they are saved in the main memory at special points of execution (i.e., *spilling* process) when the operating system runs out of general-purpose registers. As previously shown in cache-based attacks, it is important to hide the access patterns to memory. Therefore, general-purpose registers are required to be erased of the addresses (or part of it) after they are used to access memory.

Since compilers may map a variable to a different general-purpose register, and this mapping can change dynamically every time the variable is used in the program, the developer cannot keep track of registers used to store sensitive data. To use the same register for a sensitive variable, we use *inline Assembly* feature that exists in high-level languages such as C/C++. For example, a pointer variable that is used to access a lookup table and contains a sensitive address can be defined as follows in Xtensa LX3 processor:

```
register unsigned int *table_ptr asm("a13");
```

Here, the register (`a13`) is declared as the pointer to hold the address of a particular element of a lookup table that is being accessed. When the pointer variable is defined in this manner, the compiler will always use the register (`a13`) to hold the corresponding address. Note that forcing to use always the same register for a variable may have performance implications.

7 The New Instruction Set Architecture

Basic integer arithmetic and logic operations are implemented in the protected zone to provide a wide range of cryptographic algorithms with a secure and efficient execution environment. Some of these operations are implemented as simple single-cycle instructions such as integer addition and

various shift operations while more sophisticated operations such as 128-bit multiplication take multiple cycles. All instructions comply with RISC conventions such as using maximum three operands per instruction, simple addressing modes stipulating register-to-register arithmetic, etc. These conventions help keep the data path as simple and regular as possible. For instance, the latency for instruction fetch can be minimized for short and regularly formatted instructions.

The implementation of instructions for efficient integer and logic operations has been explained in detail in our previous works [19, 20]. Therefore, we focus only on explaining the new instructions that allow secure execution of cryptographic operations; but only a subset of new instructions, which we think are the most representative of the adopted methodology, is explained for space considerations.

7.1 Predicate Registers and Associated Instructions

Firstly, we mention a special register that plays a key role in secure computations. We use a one-bit predicate register, namely (p)[2], to allow predicated (or conditional) execution of certain instructions. The predicated instructions are known; however as a novelty we allow arithmetic operations to be performed on a predicate register so that more sophisticated conditions can be evaluated before the completion of an instruction. Five instructions associated with handling the predicate register are given in Table 1.

Instruction name	Arguments	Definition
set_predicate	p	p := 1
reset_predicate	p	p := 0
read_predicate	p and ar	ar := p
or_predicate	p and carry	p OR carry
mf_creg2predicate	cr and p	p := cr[127]

Table 1: Instructions pertaining to predicate registers

[2]Two predicate bits in [1] are not really necessary since predicated execution can be performed using only a one-bit predicate register.

Here, (ar) and (cr) stand for general-purpose and cryptographic registers, respectively. Another one-bit register carry is set when a previous addition operation produces a carry bit. This carry bit is copied to the predicate register via or_predicate instruction. This way, a carry can be used as predicate for conditional execution of subsequent instructions.

The instruction mf_creg2predicate in Table 1 moves the most significant bit of the cryptographic register cr to the predicate register (p) while (cr) is shifted to the left by one bit. The instruction is useful in modular exponentiation and elliptic curve scalar multiplication operations where the secret exponent (or integer) is kept in the cryptographic register and moved to the predicate registers when needed. In particular, in classical left-to-right binary exponentiation algorithm, the exponent is kept in a cryptographic register. The exponent bit which is moved to the predicate register, determines whether a modular multiplication is computed or not.

Three instructions in Table 2 are conditional instructions using the predicate register. The conditional instructions eliminate the need for conditional branches dependent on sensitive information. The first two instructions cond_mv and cond_mv_c conditionally move the content of a cryptographic register to another depending on the value of the predicate register. These instructions are useful again in the exponentiation and elliptic curve scalar multiplication operations, where certain operations are performed (e.g., modular multiplication) depending on the current value of the exponent bit, which is currently in the predicate register. For instance, in the Montgomery ladder algorithm [28] for exponentiation, the result of the modular multiplication $R0 \times R1$ is assigned either to $R0$ or $R1$ depending on the value of the current exponent bit. The conditional move instructions are useful in performing this assignment operation without using branch prediction circuit that leaks information about the secret key [5]. By performing a secret key-dependent move instruction using the predicate register, the branch prediction attacks are thwarted.

The instruction acc_carry is used to perform logical-OR operation on the two special purpose registers, namely carry and carry2. Having two carry registers is useful in modular arithmetic operations. For instance, the final subtraction operation in the Montgomery multiplication algorithm [31, 32] can be securely handled by utilizing two carry bits. In our implementation of the

Instruction name	Arguments	Definition
cond_mv	p, cr_d, cr_s	if p=1 then cr_d:=cr_s;
cond_mv_c	p, cr_d, cr_s	if p=0 then cr_d:=cr_s;
acc_carry	p, carry carry2	p := carry OR carry2;

Table 2: Special instructions for conditional executions of operations

algorithm, the register `carry2` contains the carry out of the Montgomery multiplication operation before the final subtraction. We always perform the final subtraction operation and save the result in a temporary cryptographic register using `carry`. The subtraction operation can generate another carry which is written in `carry`. If either of these carry registers is set, then the final subtraction is necessary. By setting the predicate register (`p`) to logical-OR of two carry registers, we can perform a conditional move from the temporary register to the result register. This way, the execution time does not vary because of the final subtraction and additional conditional branch, which is necessary in a conventional implementation, is removed. Therefore, the implementation becomes more resilient against the side-channel attacks based on branch mispredictions [5].

7.2 Block Cipher Related Instructions

Another special register, `sbox_in`, is useful in table lookup operations used in implementing s-box computations in block cipher algorithms. Three new instructions pertaining to sbox computation are given in Table 3.

The special register, `sbox_in` is 32-bit in length and holds a part of the state of a block cipher during s-box calculations. Before every round, the block (i.e., the current state of the block cipher) is held in cryptographic registers. The instruction `shlcr_2sbox_in` in Table 3 moves the highest 32-bit of `cr` to the special register `sbox_in`, where the round operations are applied.

The so-called substitution box (s-box) in a block cipher algorithm is a non-linear function whose evaluation is usually achieved using lookup tables. This is due to the fact that it requires algebraic

Instruction name	Arguments	Definition
`shlcr_2sbox_in`	`cr, sbox_in`	`sbox_in:= cr[127:96];` shift left `cr` by 32 bits;
`lookup_table_op`	`addr, base_addr, sbox_in`	`addr:=base_addr + sbox_in[31:24];` shift left `sbox_in` by 8 bits;
`lookup_table_op_word`	`addr, base_addr, sbox_in`	`addr:=base_addr + (sbox_in[31:24]<<2);` shift left `sbox_in` by 8 bits;

Table 3: Special instructions for block cipher related operations

manipulations that are usually too slow to implement in software. Depending on the available memory, different table lookup methods can be considered. The s-box of AES is an 8×8 function and the basic table lookup technique makes use of a 256 B (byte) memory. A more sophisticated method suggested by the designers of AES uses about 5 KB of memory to store lookup tables to achieve higher speedup values. However, as pointed out in many works in the literature [10, 11], cache-based attacks reveal a fundamental weakness in table lookup methods due to the fact that the lookup tables are kept in cache memory. Even though these tables are public, key-dependent access patterns to them in cache memory results in variations in the execution time due to cache misses.

We address the security concerns pertaining to the usage of lookup tables in s-box evaluations utilizing the architectural support in our processor. First of all, we propose to use a limited amount of non-cached local memory to store the lookup tables. This approach basically thwarts all cache-based attacks. However, there is another security concern that arises due to the fact that we use classical memory access mechanism to read the local memory. Therefore, we develop a novel, secure method of accessing relatively small lookup tables in the local memory without using general-purpose registers in address calculations. For this, we use two instructions in Table 3: `lookup_table_op` and `lookup_table_op_word`. The instruction `lookup_table_op` computes

the address of the s-box output value precisely, which allows to fetch the desired item from the lookup table securely, assuming that the table contains 8-bit entries. The calculated address and s-box output are placed in general-purpose registers which need to be properly handled and erased afterward. To support larger s-boxes in other block cipher algorithms or larger lookup tables for faster evaluation of s-box functions, the instruction `lookup_table_op_word` in Table 3 is used to return 32-bit entries in the lookup tables.

7.3 Registers in Protected Zone

In a general-purpose processor, there are two kinds of registers used in data processing: i) general-purpose registers that keep operands for normal instructions and are visible to developers and ii) special-purpose registers, which are not directly accessible by normal instructions (e.g., program counter, condition codes/flags, temporary registers for multi-cycle instructions, etc.). General-purpose registers are part of the register file, which is an array of registers within the processor. The protected zone, modeled after RISC-based general-purpose processors, also contains both general- and special-purpose registers. A cryptographic register file (*cf.* *User-Defined Register File* in Figure 1) contains eight 128-bit cryptographic registers that are general purpose in the sense that they can be used with all user-defined cryptographic instructions in the protected zone.

Other registers in the protected zone are special purpose. Four 128-bit registers are used to keep temporary data during the execution of arithmetic instructions such as 128-bit addition and multiplication. Two 128-bit registers, namely `crypto_HI` and `crypto_LO` store higher and lower 128-bit halves of a 128-bit multiplication instruction, respectively (similar to `HI` and `LO` registers in many RISC processors). The predicate register is a one-bit register that allow predicated execution of instructions, which is needed as a protection against branch prediction attacks. Both of the two one-bit carry registers (i.e., `carry` and `carry2`) are used in big arithmetic operations as explained in Section 7.1. Finally, there is one register for secure s-box computation: `sbox_in`, which is a 32-bit register used to obtain the address used to access the lookup tables. Table 4 lists all registers used in the protected zone.

Register Name	General- or Special-Purpose	Size (bits)	Number
Cryptographic registers	General-Purpose	128	8
Temporary registers	Special-Purpose	128	4
`crypto_HI`	Special-Purpose	128	1
`crypto_LO`	Special-Purpose	128	1
predicate register	Special-Purpose	1	1
carry registers	Special-Purpose	1	2
`sbox_in`	Special-Purpose	32	1

Table 4: General and special-purpose registers in the protected zone

7.4 Extending the Protected Zone with New Features and Functional Units

Thanks to the *extensibility* feature of the reconfigurable architecture, new functional units with new features, registers, and register files, etc., can be added to the processor data path. An extension to the data path of the processor is possible using Tensilica instruction extension (TIE) language. TIE is, in fact, a hardware description language, similar to VDHL and Verilog, which is used to describe instruction set extensions to the processor core. The functional behaviors of desired functional units are defined in TIE language, and TIE compiler will generate and place the RTL (register transfer level) equivalent blocks into the processor data path. In Appendix A, we demonstrate how a one-cycle 128-bit adder is encoded in TIE language as RTL. The TIE code in fact implements a 128-bit fast adder whose block diagram is given in Figure 4.

After TIE compiler generates the RTL blocks of the extensions, the processor is synthesized with the new RTL blocks. Figure 5 illustrates how the integration of the new 128-bit adder would look like in the five-stage pipeline of the reconfigurable processor core. This is a tight integration since a 128-bit addition operation is now performed in five clock cycles as in the case of all other existing instructions of the reconfigurable processor.

After synthesis stage, the configuration file of the extended processor that can be used to program the target FPGA device. ASIC realization requires vendor's assistance in the manufacturing process. The vendor also provides a tool chain (compiler, debugger, linker, loader, etc.) for software

development for the extended processor.

The reconfigurable architecture can always be extended with more powerful functional units for more security and further acceleration. For instance, we can have two multiplication units in the protected zone to take advantage of instruction level parallelism. Two multiplication units would perform two multiplication operations in elliptic curve cryptography and RSA algorithms, simultaneously, resulting in significant speedup in overall computations. However, two multiplication units naturally requires more chip space and more complicated control circuity in the pipeline of the processor.

Similarly, the reconfigurable processor could feature functional units tailored to perform instructions specific to a certain cryptographic algorithm. For instance, we can always design a highly optimized hardware module to perform multiple s-box operations of the AES algorithm at the same time. However, this module would not be used to implement any other block cipher algorithms. Therefore, the architecture would lose its generic nature.

The proposed architecture adopts two design approaches to increase its feasibility in embedded applications: i) simple architecture with acceptably low hardware cost and significant acceleration of cryptographic algorithms, and ii) generic functional units to support as many cryptographic algorithms as possible. As will be demonstrated in the subsequent sections, the proposed architecture can be implemented with relatively moderate hardware cost without a decrease in the maximum applicable clock frequency.

8 Implementation of Cryptographic Algorithms

In this section, we explain our adopted approach to implement major symmetric and asymmetric cryptographic algorithms and present the timing results in terms of clock cycle count for their performances. We selected three well-known and widely used public key cryptosystems: RSA, elliptic curve cryptosystems, and Pairing-based cryptographic algorithms. For symmetric cryptographic algorithms we implemented AES as the block cipher and SHA-1 and SHA-256 as the representatives

of cryptographic hash functions. Below, we start by presenting timing results for the basic modular arithmetic operations common in many public-key cryptosystem.

8.1 Modular Arithmetic Operations for Big Numbers

To accelerate the basic arithmetic operations in fields with prime characteristics, we use the functional units presented in [20, 21]. Table 5 lists the complexity of the basic arithmetic operations used in the computation of the pairing operation in Pairing-based cryptography in number of clock cycles.

Operation	160-bit	192-bit	256-bit	512-bit
F_p addition	167	174	170	239
F_p subtraction	171	178	174	243
F_p multiplication	905	1044	830	2167
F_p multiplication by -2	295	309	300	418
F_p inversion	36,565	42,925	55,478	140,049

Table 5: Timings of modular arithmetic operations in number of clock cycles

Similarly, in Table 6, we present the timing results for the prime extension field arithmetic used in pairing-based cryptography. We provide the timing results only for two extension degrees of 2 and 4, namely F_p^2 and F_p^4 since we use pairing operations defined for these two fields. The timings of additions and subtractions for prime extension fields are not included here as they can be accurately estimated from modular addition and subtraction operations in Table 5. F_p^4 inversion and multiplication timings are not given for 512-bit since our pairing implementation uses only the embedding degree of 2 over 512-bit prime field. One obvious observation from Tables 5 and 6 is that the major operation in F_p^2 and F_p^4 inversion calculations is the inversion in the prime field F_p.

We use irreducible polynomials of the form $X^2 - \beta$ and the tower field approach in the construction of prime extension fields for a faster computation of the basic field arithmetic. Note also that these implementations are developed using our secure programming technique in the protected zone.

Operation	256-bit	512-bit
F_p^2 multiplication	3,775	7,836
F_p^4 multiplication	13,541	-
F_p^2 inversion	60,983	152,738
F_p^4 inversion	77,819	-

Table 6: Timings of arithmetic operations in extension fields in number of clock cycles

Algorithm	**Base Architecture**	**Fast on protected**	**Secure protected**
RSA-1024	132,334,584	9,215,168	14,831,132
RSA-2048	NA	66,728,848	107,173,686
ECC-160	5,684,844	2,524,498	4,683,325
ECC-256	21,509,576	3,649,338	7,213,678
ECC-512	160,109,439	16,979,307	33,893,033

Table 7: Clock count for RSA and ECC

8.2 RSA and Elliptic Curve Implementations

The timing results of an RSA exponentiation and an ECC scalar point multiplication are given in Table 7 in terms of number of clock cycles. Note that all implementations are done in C language with some lines in inline Assembly and implementations in full Assembly are expected to yield better performance.

For both RSA and ECC, we list the timing results for three implementations: base, fast, and secure Table 7. The base implementation is implemented without using the cryptographic acceleration and therefore it is not secure (not to mention the fact that the implementations are prohibitively slow). Fast implementations execute completely in isolated manner, whereby all computations are performed in the protected zone. But they are vulnerable to simple side-channel attacks since we use the binary left-to-right exponentiation algorithm in the computations of RSA exponentiation as well as elliptic curve scalar multiplication, which is vulnerable to the simple power analysis (SPA). In order to harden these operations against the SPA and branch prediction attacks,

we employed the Montgomery Ladder algorithm [28] along with conditional move instructions (*cf.* Section 7.1) in our secure implementations of RSA and ECC, hence the *secure* implementation.

The Montgomery Ladder algorithm is a typical example of algorithm level countermeasures against the simple power analysis attacks. It is used in RSA exponentiation and elliptic curve scalar point multiplication operations. Independent of the exponent (the scalar integer in ECC), which is the secret information, the algorithm always performs the same operations (one modular multiplication and one modular squaring for every bit of the secret exponent in RSA). Therefore, no information leaks due to secret key dependent operations. We further eliminate any remaining dependency on the secret key by using conditional move instructions, which remove any conditional statement such as the `if` statement that checks the bits of the secret exponent.

The differences in the clock cycle counts of the fast on protected and the secure protected implementations (see the third and the fourth columns in Table 7) are due to two factors. Firstly, the Montgomery ladder algorithm always performs a multiplication (point addition in ECC) for each bit of the exponent while binary exponentiation algorithms perform a multiplication only if the corresponding exponent bit is 1. Secondly, conditional move instructions based on predicate registers require a higher number of clock cycles.

For comparison, our implementations, both fast and secure, greatly outperform another 1024-bit RSA implementation on the same Xtensa processor in [33] where one exponentiation operation takes about 24.32 million clock cycles. In a more recent work [24], in which the FPGA realization of the extended processor runs at 24 MHz and execution times are given for modular multiplication only, one 1024-bit modular multiplication takes $25,418$ clock cycles. In comparison, the same operation takes $7,654$ clock cycles in our processor. Considering our architecture can run at a twice faster clock frequency of 50 MHz, the speedup is about 6.92.

It turns out that the required sizes of the local memory are surprisingly low for RSA and ECC implementations; maximum 5,700 Byte (B) is needed for the fast implementation of RSA and it is only $1,936$ B for ECC. RSA memory requirement in fast implementation can be reduced to as low as $1,860$ B at the expense of $17-18\%$ deterioration in speed. The secure RSA imple-

mentation requires only 2,112 B memory space. Considering our former disccussion about the feasibility of impelementing local memory on-chip in Section 5, the local memory requirements of our implementations are very low.

8.3 AES Implementations

In encryption (symmetric as well as asymmetric), we can assume that the plaintext is usually placed first in the main memory, which is not a secure place. Our implementations take a block of plaintext (128-bit or 16 B in AES) from the main memory and place it in the local memory, which is a secure place. During the computation of the AES rounds, anything computed remains in the protected zone; i.e. local memory, cryptographic or special purpose registers. After the final round of the block cipher is executed, the resulting ciphertext block is transferred to the main memory. One can argue that an adversary who already compromised the main memory can easily obtain the plaintext while it is in the main memory. Therefore, it can further be argued that the proposed protection does not help to secure the encryption operation. However, the primary goal of our architecture is to protect the secret key used in the encryption process. Even if the main memory is compromised, our architecture can still carry on encryption (or decryption) operations without leaking sensitive information.

We developed two C implementations of the AES algorithm and the results are given in Table 8 along with those of other implementations (some of them on similar embedded platforms). In the table we provide operating clock frequencies for the designs, for which FPGA implementations are provided. Our first implementation, referred as the *limited memory* version, utilizes a lookup table with 256 entries where each entry is 8-bit, which is stored in the local memory. It is basically used in the direct evaluation of 8×8 s-box of the AES algorithm. As seen in the table, our implementation is outperformed by those in [24], [34] and [35]. The implementation in [35] is a bit-sliced implementation. Our implementation does not use the bit-slicing technique, therefore, it can work in any mode of operation. A bit-sliced implementation in our architecture would possibly yield a better performance, which we leave as a future work. Note also that the implementation

in [24] uses a slower clock frequency.

Implementation	Hardware support	Performance (cycles)
[36] on ARM7TDMI	-	1675
[37] on AMD Opteron	-	2699
[35] on CRISP	Bit-sliced + lookup tables	2203
[35] on CRISP	Bit-sliced + lookup table + bit-level permutation	1222 (@30 MHz)
[33] on Xtensa	-	1400
[24] on LEON 3	Hardware support	463 (@24 MHz)
Reference implementation [34] on XTensa	No hardware support	859 (@50 MHz)
this work - limited memory	**on protected zone**	**1334 (@50 MHz)**
this work - fast	**on protected zone**	**863 (@50 MHz)**

Table 8: Comparison of AES Implementations

The implementation [34] uses five lookup tables of size 1 KB each resulting in about 5 KB memory for table lookup approach. However, this implementation has been shown to succumb to cache-based attacks. Our second implementation modifies the implementation [34] by using the local memory for lookup tables and secure memory access techniques. As seen in Table 8, our fast implementation provides the same performance as the reference implementation [34] without sacrificing the security.

The AES algorithm requires only 464 B of scratch pad memory for the limited memory version while the requirement is $5,328$ B for the fast version. The scratch pad memory required by the fast implementation of RSA can also be used to enable the execution of the fast version of AES.

8.4 Implementing Pairing-Based Cryptography and Protocols Based on Pairing

Elliptic curve based pairing operation recently emerged as a very important cryptographic primitive and already became an essential part of many cryptographic schemes and protocols [38, 39,

40, 41, 42, 43, 44]. Therefore, it is important to implement pairing operation on embedded devices efficiently. However, pairing operation is costly and generally prohibitively slow on embedded processors [45]. Therefore, full or partial hardware support for acceleration is a frequently adopted methodology [25]. In this section, we demonstrate that it is possible to accelerate pairing operation significantly if implemented in our protected zone. But, we will first provide a brief introduction to pairing operation defined over elliptic curves to enable a better understanding of our implementations.

8.5 Bilinear Pairing and Tate Pairing Operation

Bilinear pairing is a function that maps two points in elliptic curve groups to a subgroup of the extension field F_{p^k}; more formally $e : G_1 \times G_2 \rightarrow G_3$, where $G_3 = F_{p^k}$, k is the embedding degree, and p is the prime characteristic of the field F_p, over which the elliptic curve group G_1 is defined. In practice, all three groups have the same order r and the embedding degree k is a small integer. An important property that makes the pairing operation interesting for cryptographic applications is its *bilinearity*

$$e(aP, bQ) = e(P, Q)^{ab} = e(bP, aQ) = e(aP, Q)^b = e(P, bQ)^a,$$

where P and Q (i.e., uppercase letters) are elliptic curve points while a and b (i.e., small case letters) are integers. The Tate pairing, which is one of the most efficient pairings widely used in cryptography can be computed using Miller's algorithm, followed by a final exponentiation operation. The BKLS algorithm [46], described in Algorithm 1, is an efficient method to compute the Tate pairing.

The function g in Algorithm 1 takes two elliptic curve points from G_1 and one point from G_2 and returns an element in F_{p^k}. In our implementations we use two values for embedding degree, namely, $k = 2$ or 4. In addition, we use the same elliptic curve for the groups G_1 and G_2. Specifically, while G_1 is the elliptic curve over F_p, G_2 is the same curve defined over F_{p^k}. Since we use the quadratic twist [47], [45] of the elliptic curve for $k = 4$, the elements of G_2 have coordinates from F_{p^2} in both

Algorithm 1 BKLS Algorithm for Computing Tate Pairing $e(P, Q)$

Require: r, p, P, Q
Ensure: $f = e(P, Q)$
  // *Miller's Loop*
  $f = 1$
  $T = P$
  $n = r - 1$
  for i **from** $\lfloor \log_2 r \rfloor - 2$ **downto** 0 **do**
    $f = f^2 \cdot g(T, T, Q)$
    if $n_i = 1$ **then**
      $f = f \cdot g(T, P, Q)$
    end if
  end for
  $f = f^{(p^k-1)/r}$ // *Final Exponentiation*

cases (i.e., $k = 2$ or 4). The irreducible polynomials $x^2 + 1$ and $x^2 + 2$ can be used to construct the quadratic field F_{p^2}. Consequently, the elements of F_{p^2} can be represented as $x + iy$, where $i = \sqrt{-1}$ or $\sqrt{-2}$ and $x, y \in F_p$. For more information, see [47].

The function g can be computed using Algorithm 2, where the elliptic curve points $A, B \in G_1$ are given in projective coordinates (e.g., $A = (X_a, Y_a, Z_a)$) while we use the affine representation for the point $Q \in G_2$ (i.e., $Q = (x_q, y_q)$). Note that the point A is modified by the function g.

Algorithm 2 Computation of g Function in Tate Pairing

Require: $A = (X_a, Y_a, Z_a), B = (X_b, Y_b, Z_b), Q = (x_q, y_q)$, where $A, B \in G_1$ and $Q \in G_2$
Ensure: $m = g(A, B, Q)$
  $C = A + B$ // *elliptic curve point addition*
  $\lambda = Z_a^3 Y_b - Y_a$
  $m = Y_a Z_c - \lambda(x_q Z_a^3 + X_a Z_a) - i(y_q Z_a^3 Z_c)$
  $A = C$

The final exponentiation $f^{(p^k-1)/r}$ can be computed using the Frobenius as explained in [45]. Moreover, two methods are utilized to accelerate the pairing operation. The first method is a precomputation technique that can be used when the the first elliptic curve point P in $e(P, Q)$ is fixed. Note that the Miller's loop in Algorithm 1 computes the same multiple of the P independent of Q. Thus, we can precompute the elliptic curve additions in the computation of g and

save them in a lookup table. This eliminates costly elliptic curve operations during the Miller's loop. The second acceleration method (i.e., Lucas sequence method) can be applied in the final exponentiation operation in case $k = 2$ (*cf.* [48] for the application of the Lucas sequences to the final exponentiation).

8.6 Our Implementations

We implemented the Tate pairing operation to see if it can be computed efficiently on our architecture. For elliptic curve arithmetic and pairing operations we use two elliptic curves provided in MIRACL library [49]. The first elliptic curve uses a 256-bit prime field F_p with embedding degree $k = 4$ for pairing operations. The extension fields F_p^2 and F_p^4 are constructed using tower field approach and irreducible polynomials of the form $X^2 - \beta$. Since the base point order of the elliptic curve group is a 160-bit number and embedding degree is 4, this selection of pairing operation provides 80-bit security level, which is suitable for embedded applications.

The second curve is constructed over 512-bit prime field with the embedding degree $k = 2$ for pairing operation and provides also 80-bit security. The second curve is used for the pairing operation with precomputation for the case when the first point in the pairing is fixed. Results for both implementations are given in Table 9. Note that the results for the precomputation case are about two times faster than those in [25], which is also based on instruction set extension technique.

Operation	256-bit	512-bit
Point addition	13,509	32,915
Point doubling	9,793	21,581
Point multiplication	2,478,410	5,122,758
Miller's Loop	10,928,255	29,77,028 (with precomputation)
Final Exponentiation	5,764,617	1,889,324 (with Lucas sequence)
Pairing (Total)	16,692,872	4,866,352

Table 9: Timings of elliptic curve and pairing operations in number of clock cycles

In Table 9, the timing results for elliptic curve point arithmetic are also included since they are used together with pairing operations in many cryptographic schemes. Note that a scalar point multiplication in Table 9 is much faster than those in Table 7 for the same precision since elliptic

curves used in pairing operations have a much smaller base point order, which determines the required number of point addition and doubling operations.

One pairing operation on our processor running at 50 MHz takes about 334 ms for 256-bit case while it is 97.3 ms for 512-bit case where we use precomputation. Note that the reason we choose 50 MHz as the clock frequency is that it is typical for embedded systems and that we are able use 50 MHz for the synthesis of our processor on our FPGA board. In the precomputation case, two coordinates of the point P in Algorithm 1 and integer λ in Algorithm 2 are precomputed. The order of the curve used in the implementation is a 160-bit integer, and λ and elliptic curve coordinates are 512-bit numbers. Consequently, the precomputation method requires about 10 KB of memory.

Many pairing-based cryptographic algorithms and protocols also make use of other operations such as elliptic curve arithmetic besides pairing operation. For example, the privacy-preserving authentication protocol introduced in [44] uses one pairing and seven elliptic curve multiplication operations for a wireless user to authenticate to an access point. This workload on a user who may have an embedded system to connect to the network can be prohibitively heavy. However, when implemented on our proposed architecture the computation by a user takes about only $36,520,152 = 16,692,872 + 2,478,410 \times 7$ clock cycles. If a clock frequency of 50 MHz is used for the processor, the computation of all cryptographic operations can be completed in less than a second; i.e. approximately 731 ms, which is quite acceptable for such an involved computation.

One pairing operation requires that a diverse set of operations be executed such as the basic arithmetic operations in F_p and F_{p^2}, elliptic curve arithmetic operations, etc. The detailed call graph for our implementation of the pairing operation is given in Figure 6 in Appendix B. These operations in the call graph require local variables to store temporary results, which are sensitive and therefore kept in the local memory. For this, we calculate and allocate the required amount of memory needed throughout the call graph in advance. Note that operations in the same level of the call graph can share the same memory area. The total amount of the memory needed for the Pairing operation is less than 5 KB (not including the memory space for the precomputation method in Table 9).

8.7 Secure Implementation of Hash Functions on Cryptographic Unit

In cryptographic hash computation for a given message, the confidentiality is not an issue since anything computed is public. Therefore, it is important to guarantee the *correct* computation of a hash value; in a sense what is important is the integrity. For instance, if the intermediate values during the hash computation are stored in the main memory, an attacker who has access to the address space of a cryptographic application, can modify them and tweak the computation that will produce an incorrect hash value, which may be the value intended by the attacker. In our architecture, it is easy to secure the hash computation by simply saving the hash block in the protected local memory.

General-purpose registers can be used for the hash operations and hence the hash block can temporarily be kept in these registers. However, these registers must be backed up on the secure memory before these registers are spilled into the memory. The backing up the hash block can be performed as many times as required. Since the processing of one message block can be finished relatively fast, and implementing the core hash functionality for a message block is possible in a single function (method), it is usually sufficient to back up the hash block after the processing of one message block is completed. Figure 3 illustrates the secure computation of the hash of a message. In the figure, IV and MB_i stand for the initial vector and the i-th message block, respectively. In hash computations, the message is partitioned into fixed length blocks, each of which is processed within the core function of the hash known as the digest. If the message is not a multiple of the block length, a padding scheme is applied for the last message block. After the finalization step the hash value is written back to the local memory. The *backup* operation in Figure 3 represents spilling all registers holding the current hash block to the local memory. If, for instance, the processor context switches to another process after computing a hash block corresponding to the message block $(i-1)$, for the resumption of hash computation (for the block i and onward) on the next context switch back to the hash computation, the hash block $(i-1)$ is read from the local memory.

During the computation of hash functions, general-purpose registers of the base processor are

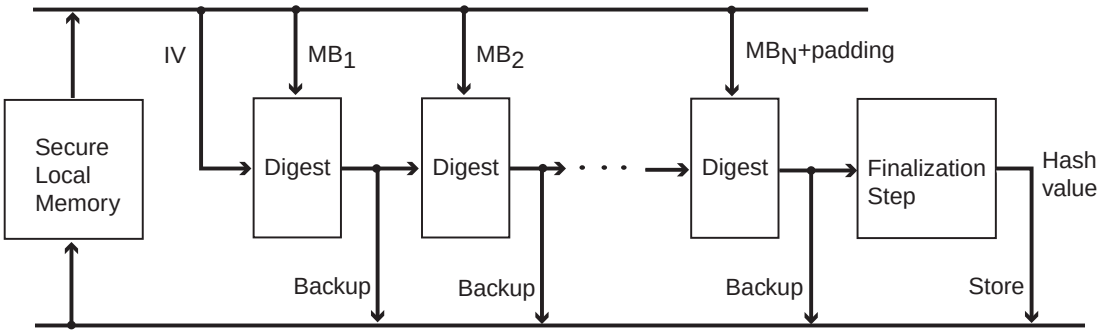

Figure 3: Secure hash computation using local memory

used. These registers must be mapped to the secure local memory in order for them to be backed up in the local memory when needed.

In our secure implementation of the SHA-1 function, computing the hash of a 512-bit message takes 1529 clock cycles. The same computation takes 1,602 clock cycles in the architecture in [35] (another implementation in the same architecture [35] using lookup tables takes 1441 clock cycles). Note that the maximum applicable clock frequency in [35] is 30 MHz while we can apply 50 MHz to our architecture. We also implemented SHA-256 (256-bit version of SHA-2) in the same manner to provide the performance result for 128-bit security level and we found out that the hash of a 512-bit message is computed in 7,550 clock cycles. In comparison, the same operation takes 6,255 clock cycles in the architecture in [24] at a maximum clock frequency of 24 MHz. In both cases, our overall execution times are better than those in [35] and [24].

For hash-based MAC computations, in order to protect secret key, hash blocks should also be protected in the same way as the blocks of AES, which will slow down the computation.

9 Implementation Results

We implemented our protected zone and integrated it into an embedded processor core, namely Xtensa LX3 by Tensilica [30]. Design choices for the extensions are made in such a way that the applicable clock frequency is not affected. The Tensilica tool chain estimates that the protected zone takes a chip area of 100,279 equivalent gate count in an ASIC realization while the base

processor takes about 88,000 equivalent gate count. Therefore, the total size of the processor is estimated as 188,279 equivalent gate count in an ASIC realization.

The results for an ASIC realization are rough estimates and the tools do not report reliable figures for maximum applicable clock frequency (The base processor runs at a maximum clock frequency of 320 MHz). Therefore, we also synthesized the design into an FPGA target device, and generated configuration files to program Avnet LX200 board that features Xilinx Virtex-4 type FPGA. This basically means that the reported implementation figures are obtained after placement-and-routing. The number of slices used in the design is reported to be 27,401; and 14,432 slices of the total number are used to implement the protected zone. Note that there are 89,088 slices in the FGPA device. No degradation in clock frequency is reported by the synthesis tools; 50 MHz maximum clock frequency is achieved for the design. Note that, the implementation figures do not include the area which will be required for the local memory.

We compare the implementation results with the architecture in a recent work [24], which reflects the state-of-the-art in instruction set extensions for cryptographic algorithms. The design in [24] is based on LEON 3 microprocessor architecture and realized on Xilinx Virtex 5 FPGA. Since the processor cores and target devices (FPGA) are different, a direct comparison is not fair. However, relative increases in areas (i.e., number of slices) can give an idea whether the overhead is reasonable and/or acceptable. In [24], when all extensions are considered, the relative increase in the number of slices is approximately 207% (the base processor and extensions take 2,338 and 4,830 slices, respectively) as opposed to an increase of about 111% in our architecture. The relative increase incurred in the area due to the extensions, while significant, is typical and acceptable. Another architecture [35], claiming to be lightweight and supporting only block ciphers and SHA-1 algorithm, takes 9,500 slices on Virtex II architecture.

The equivalent gate count of the base Xtensa LX3 microprocessor (i.e., 88,000 equivalent gates) reflects the fact that its architecture is optimized for ASIC realizations. This also explains relatively high slice count on the FGPA device, for which the processor is not optimized. The gate count of the base processor (88,000) and the additional gate count due to the protected zone (100,279)

are relatively low figures, which are feasible for embedded applications. For comparison with an equivalent embedded core, we can inspect the *deeply-embedded* ARM processor core, Cortex-R4 [50], which has an equivalent gate count of 290,000 for the maximum clock frequency of 934 MHz. The gate count can be as low as 150,000 for lower clock frequencies.

To compare the hardware cost of the proposed architecture with cryptographic accelerators, we select two high performance designs specialized for elliptic curve cryptography to give an idea of their hardware costs. The first design is a high-speed elliptic curve accelerator [51], which takes 750,000 gate count operating at 200 MHz. A low-power power design [52] for elliptic curve cryptography takes about 34,000 gates and operates at 200 MHz.

In summary, our design features an embedded processor core with a generic support for many cryptographic algorithms and its hardware budget is similar to those of similar embedded systems.

10 Conclusion

We designed, implemented and realized a protected zone in an embedded processor that supports efficient and secure execution of cryptographic algorithms. We estimated the area overhead of the protected zone for the ASIC implementation. We also provided area usage on an FPGA device after placement-and-routing for the full design including an embedded base processor and the protected zone. Since the number and organization of the subsystems in the protected zone are carefully designed, we observed that the area overhead is acceptable for an embedded processor while no deterioration in maximum applicable clock frequency is reported in FPGA realization. We outlined the principles of software implementations of cryptographic algorithms so that the resulting executables run in a secure and isolated manner. Since the protected zone is specifically tailored for the cryptographic application domain, we achieved superior time performance of major cryptographic algorithms compared to both those reported in the literature and those in the base processor. The protected zone benefits software implementations of many cryptographic algorithms since the selected design methodology is not aimed at favoring a particular class of algorithms.

Acknowledgment

This work is supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project number 105E089 (TUBITAK Career Award).

References

[1] Durahim, A. O., Savas, E., and Yumbul, K. (2009) Implementing a protected zone in a reconfigurable processor for isolated execution of cryptographic algorithms. *Proceedings of ReConFig'09*, Cancun, Mexico, 9-11 December, pp. 207–212. IEEE Computer Society.

[2] Kocher, P. C. (1996) Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Proceedings of CRYPTO 1996*, Santa Barbara, California, USA, 18-22 August, LNCS, **1109**, pp. 104–113. Springer, Berlin.

[3] Kocher, P. C., Jaffe, J., and Jun, B. (1999) Differential power analysis. *Proceedings of CRYPTO 1999*, Santa Barbara, California, USA, 15-19 August, LNCS, **1666**, pp. 388–397. Springer, Berlin.

[4] Boneh, D., DeMillo, R. A., and Lipton, R. J. (1997) On the importance of checking cryptographic protocols for faults (extended abstract). *Proceedings of EUROCRYPT 1997*, Konstanz, Germany, 11-15 May, LNCS, **1233**, pp. 37–51. Springer, Berlin.

[5] Aciiçmez, O., Ç. K. Koç, and Seifert, J.-P. (2007) Predicting secret keys via branch prediction. *Proceedings of CT-RSA Conference 2007*, San Francisco, CA, USA, 5-9 February, LNCS, **4377**, pp. 225–242. Springer, Berlin.

[6] Gueron, S. (2012) *Advanced Encryption Standard (AES) Instructions Set - Rev 3.01.* Website (online). `http://software.intel.com/en-us/articles/advanced-encryption-standard-aes-instructions-set/`.

[7] Intel Publications no. 315168-009 (2006) Intel trusted execution technology (intel

TXT). Technical Report. Intel Corporation. http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html.

[8] AMD Publication no. 33047 rev. 3.01 (2005) Amd64 virtualization: Secure virtual machine architecture manual. Technical Report. AMD, Inc. http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.

[9] ARM Publications no. PRD29-GENC-009492C (2009) Arm security technology: Building a secure system using trustzone technology. Technical Report. ARM Limited. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[10] D. Bernstein (2005). Cache-Timing Attacks on AES. Website (online). http://cr.yp.to/papers.html#cachetiming.

[11] Aciiçmez, O., Schindler, W., and Çetin Kaya Koç (2007) Cache based remote timing attack on the AES. *Proceedings of CT-RSA Conference 2007*, San Francisco, CA, USA, 5-9 February, LNCS, **4377**, pp. 271–286. Springer, Berlin.

[12] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. (2008) Lest we remember: Cold boot attacks on encryption keys. *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, USA, 28 July - 1 August, pp. 45–60. USENIX Association.

[13] Gassend, B., Suh, G. E., Clarke, D. E., van Dijk, M., and Devadas, S. (2003) AEGIS: architecture for tamper-evident and tamper-resistant processing. *Proceedings of the 17th Annual International Conference on Supercomputing, ICS 2003*, San Francisco, CA, USA, 23-26 June, pp. 160–171. ACM, New York, NY, USA.

[14] Tiri, K. and Verbauwhede, I. (2003) Securing encryption algorithms against DPA at the logic level: Next generation smart card technology. *Proceedings of CHES 2003*, Cologne, Germany, 8-10 September, LNCS, **2779**, pp. 125–136. Springer, Berlin.

[15] Lee, R. B., Kwan, P. C. S., McGregor, J. P., Dwoskin, J. S., and Wang, Z. (2005) Architecture for protecting critical secrets in microprocessors. *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA 2005)*, Madison, Wisconsin, USA, 4-8 June, pp. 2–13. IEEE Computer Society.

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

[17] Suh, G. E., O'Donnell, C. W., and Devadas, S. (2007) AEGIS: A single-chip secure processor. *IEEE Design & Test of Computers*, **24**, 570–580.

[18] Szefer, J., Zhang, W., Chen, Y.-Y., Champagne, D., Chan, K., Li, W. X. Y., Cheung, R. C. C., and Lee, R. B. (2011) Rapid single-chip secure processor prototyping on the OpenSPARC fpga platform. *Proceedings of the 22nd IEEE International Symposium on Rapid System Prototyping, RSP 2011*, Karlsruhe, Germany, 24-27 May, pp. 38–44. IEEE.

[19] Kocabaş, O., Savaş, E., and Groβschädl, J. (2008) Enhancing an embedded processor core with a cryptographic unit for speed and security. *Proceedings of ReConFig'08*, Cancun, Mexico, 3-5 December, pp. 409–414. IEEE Computer Society.

[20] Yumbul, K., Savas, E., Kocabas, Ö., and Großschädl, J. (2014) Design and implementation of a versatile cryptographic unit for risc processors. *Security and Communication Networks*, **7**, 36–52.

[21] Yumbul, K. and Savaş, E. (2009) Efficient, secure, and isolated execution of cryptographic algorithms on a cryptographic unit. *Proceedings of the 2nd International Conference on Security of Information and Networks, SIN 2009*, Gazimagusa, North Cyprus, 6-10 October, pp. 143–151. ACM, New York, NY, USA.

[22] Champagne, D. and Lee, R. B. (2010) Scalable architectural support for trusted software.

Proceedings of the 16th International Conference on High-Performance Computer Architecture, HPCA-16 2010, Bangalore, India, 9-14 January, pp. 1–12. IEEE Computer Society.

[23] Großschädl, J. and Savaş, E. (2004) Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). *Proceedings of CHES 2004*, Cambridge, MA, USA, 11-13 August, LNCS, **3156**, pp. 133–147. Springer, Berlin.

[24] Grabher, P., Großschädl, J., Hoerder, S., Järvinen, K., Page, D., Tillich, S., and Wójcik, M. (2012) An exploration of mechanisms for dynamic cryptographic instruction set extension. *J. Cryptographic Engineering*, **2**, 1–18.

[25] Scott, M., Costigan, N., and Abdulwahab, W. (2006) Implementing cryptographic Pairings on smartcards. *Proceedings of CHES 2006*, Yokohama, Japan, 10-13 October, LNCS, **4249**, pp. 134–147. Springer, Berlin.

[26] Yumbul, K., Erdem, S. S., and Savas, E. (2011) On protecting cryptographic applications against fault attacks using residue codes. *Proceedings of 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011*, Tokyo, Japan, 29 September, pp. 69–79. IEEE.

[27] Yumbul, K., Erdem, S. S., and Savas, E. (2012) On selection of modulus of quadratic codes for the protection of cryptographic operations against fault attacks. *IEEE Transactions on Computers*, **99**, 1.

[28] Joye, M. and Yen, S.-M. (2002) The Montgomery powering ladder. *Proceedings of CHES 2002*, Redwood Shores, CA, USA, 13-15 August, LNCS, **2523**, pp. 291–302. Springer, Berlin.

[29] Chen, T., Raghavan, R., Dale, J., and Iwata, E. (2005). Cell broadband engine architecture and its first implementation. Website (online). `http://www.ibm.com/developerworks/power/library/pa-cellperf/`.

[30] Halfhill, T. R. (2009). Tensilica tweaks xtensa: Xtensa lx3 and xtensa 8 cores boost

performance, cut power. Website (online). http://www.tensilica.com/uploads/pdf/Tensilica-LX3_Reprint.pdf.

[31] Walter, C. (1999) Montgomery exponentiation needs no final subtractions. *Electronics Letters*, **35**, 1831–1832.

[32] Hachez, G. and Quisquater, J.-J. (2000) Montgomery exponentiation with no final subtractions: Improved results. *Proceedings of CHES 2000*, Worcester, MA, USA, 17-18 August, LNCS, **1965**, pp. 293–301. Springer, Berlin.

[33] Ravi, S., Raghunathan, A., Potlapally, N. R., and Sankaradass, M. (2002) System design methodologies for a wireless security processing platform. *Proceedings of the 39th Design Automation Conference, DAC 2002*, New Orleans, LA, USA, 10-14 June, pp. 777–782. ACM, New York, NY, USA.

[34] Barreto, P. (2013). The AES Block Cipher in C++. Website (online). http://www.larc.usp.br/~pbarreto/.

[35] Grabher, P., Großschädl, J., and Page, D. (2008) Light-weight instruction set extensions for bit-sliced cryptography. *Proceedings of CHES 2008*, Washington, D.C., USA, 10-13 August, LNCS, **5154**, pp. 331–345. Springer, Berlin.

[36] Bertoni, G., Breveglieri, L., Fragneto, P., Macchetti, M., and Marchesin, S. (2002) Efficient software implementation of AES on 32-bit platforms. *Proceedings of CHES 2002*, Redwood Shores, CA, 13-15 August, LNCS, **2523**, pp. 159–171. Springer, Berlin.

[37] Könighofer, R. (2008) A fast and cache-timing resistant implementation of the AES. *Proceedings of CT-RSA Conference 2008*, San Francisco, CA, USA, 8-11 April, pp. 187–202. Springer, Berlin.

[38] Galbraith, S. D., Paterson, K. G., and Smart, N. P. (2008) Pairings for cryptographers. *Discrete Applied Mathematics*, **156**, 3113–3121.

[39] Chen, L., Morrissey, P., and Smart, N. P. (2008) Pairings in trusted computing. *Proceedings of the Second International Conference, Pairing 2008*, Egham, UK, 1-3 September, LNCS, **5209**, pp. 1–17. Springer, Berlin.

[40] Granger, R., Page, D., and Smart, N. P. (2006) High security Pairing-based cryptography revisited. *Proceedings of the 7th International Symposium, ANTS-VII*, Berlin, Germany, 23-28 July, LNCS, **4076**, pp. 480–494. Springer, Berlin.

[41] Boneh, D., Boyen, X., and Shacham, H. (2004) Short group signatures. *Proceedings of CRYPTO 2004*, Santa Barbara, California, USA, 15-19 August, LNCS, **3152**, pp. 41–55. Springer, Berlin.

[42] Ren, K. and Lou, W. (2008) A sophisticated privacy-enhanced yet accountable security framework for metropolitan wireless mesh networks. *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems, ICDCS 2008*, Beijing, China, 17-20 June, pp. 286–294. IEEE Computer Society.

[43] Joux, A. (2000) A one round protocol for tripartite Diffie-Hellman. *Proceedings of the 4th International Symposium, ANTS-IV 2000*, Leiden, The Netherlands, July 2-7, pp. 385–394. Springer, Berlin.

[44] Durahim, A. O. and Savas, E. (2011) A^2-MAKE: An efficient anonymous and accountable mutual authentication and key agreement protocol for wmns. *Ad Hoc Networks*, **9**, 1202–1220.

[45] Devegili, A. J., Scott, M., and Dahab, R. (2007) Implementing cryptographic Pairings over Barreto-Naehrig curves. *Proceedings of the First International Conference Pairing 2007*, Tokyo, Japan, 2-4 July, LNCS, **4575**, pp. 197–207. Springer, Berlin.

[46] Scott, M. (2005) Computing the Tate Pairing. *Proceedings of CT-RSA Conference 2005*, San Francisco, CA, USA, 14-18 February, LNCS, **3376**, pp. 293–304. Springer, Berlin.

[47] Scott, M. A note on twists for Pairing friendly curves. Website. `ftp://ftp.computing.dcu.ie/pub/crypto/twists.pdf`.

[48] Scott, M. and Barreto, P. S. L. M. (2004) Compressed Pairings. *Proceedings of CRYPTO 2004*, Santa Barbara, California, USA, 15-19 August, LNCS, **3152**, pp. 140–156. Springer, Berlin.

[49] Scott, M. (2013) *MIRACL—A Multiprecision Integer and Rational Arithmetic C/C++ Library*. Shamus Software Ltd, Dublin, Ireland. Available at `http://www.certivox.com/miracl/`.

[50] Turner, C. (2010). Cortex-R4 processor: High-performance and high-reliability for deeply-embedded real-time systems (white paper). Website (online). `http://www.arm.com/files/pdf/Cortex-R4-white-paper.pdf`.

[51] Zhang, X. and Li, S. (2007) A high performance ASIC based elliptic curve cryptographic processor over GF(p). *Proceedings of the 2nd International Design and Test Workshop, IDT 2007*, Cairo, Egypt, 16-18 Dec, pp. 182–186. IEEE.

[52] Öztürk, E., Sunar, B., and Savas, E. (2004) Low-power elliptic curve cryptography using scaled modular arithmetic. *Proceedings of CHES 2004*, Cambridge, MA, USA, 11-13 August, LNCS, **3156**, pp. 92–106. Springer, Berlin.

11 Appendix A: Addition Unit in TIE language

In this section, we explain how a new functional unit in the protected zone will be implemented and integrated to the base processor's pipelined data path. As an example, we consider the 128-bit adder unit used in big integer arithmetic for public key cryptography algorithms. The block diagram of the adder is given in Figure 4. Three 64-bit adder units are employed in the design, where two of them are used to implement the addition of the upper 64 bits of the integers. They, in fact, perform the same operation, one with carry and the other without carry. The carry out from the lower 64-bit addition is used to select the correct result for the upper 64-bit addition. The sizes of the adders are selected in such a way that the maximum applicable clock frequency is not affected.

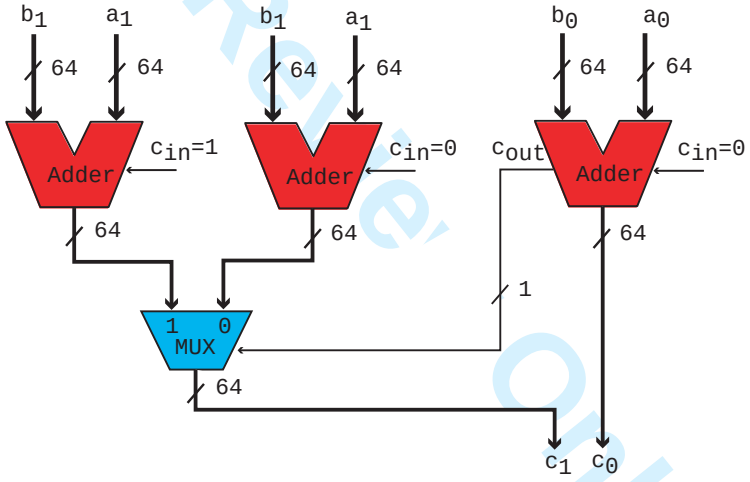

Figure 4: Adder unit for 128-bit integer addition operation [20]

The 128-bit adder unit is described in TIE language as follows:

```
function [128:0] add128 ( [127:0] oper1, [127:0] oper2, carry) shared
{
  wire [64:0] hi, hi_with_carry, hi_wo_carry, lo;
  wire c_in_1, c_in_0;

  assign c_in_1 = 1'b1;
  assign c_in_0 = 1'b0;
```

```
  assign lo    = TIEadd(oper1[63:0], oper2[63:0], carry);
  assign hi_wo_carry = TIEadd(oper1[127:64], oper2[127:64], c_in_0);
  assign hi_with_carry  = TIEadd(oper1[127:64], oper2[127:64], c_in_1);

  assign hi = TIEmux(lo[64], hi_wo_carry, hi_with_carry);

  assign add128 = {hi, lo[63:0]};
}
```

As can be seen from the TIE language description, the language is similar to other hardware description languages such as VHDL and Verilog. TIE compiler takes the description and generates an RTL block which is ready to be integrated into the processor core. A synthesizer tool takes both the new RTL blocks and the base processor as input and integrates the RTL blocks into the processor core. As an example, Figure 5 illustrates the integration of the 128-bit adder into the five-stage pipeline of the base processor. From the figure, we can easily see that it is a very tight integration and new instructions using the new functional units in the protected zone are treated as normal instructions of the base processor. The new instructions are scheduled in the same manner as the other instructions and go through all pipeline stages.

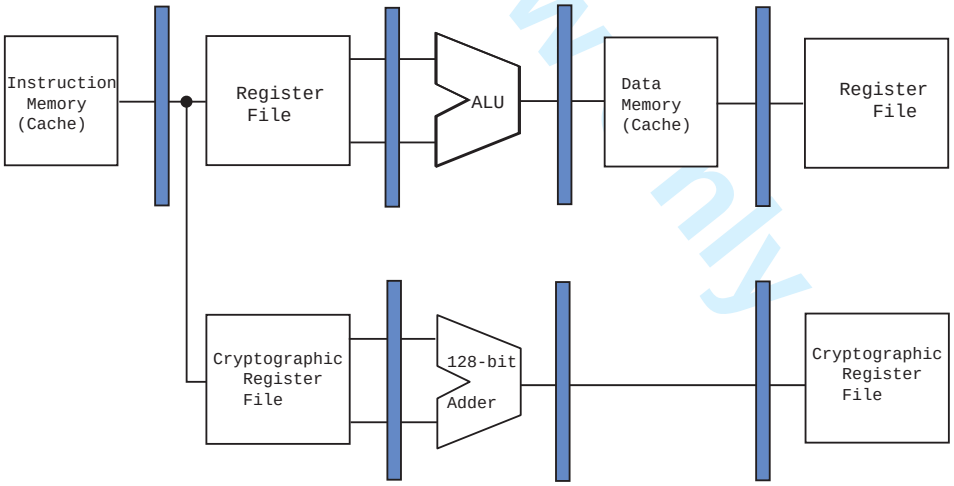

Figure 5: Tight integration of functions units in the protected zone into processor pipeline

12 Appendix B: Call Graph of Pairing Computation

In Figure 6, a detailed call graph of the pairing algorithm is given. Since temporary results during the computation can be sensitive, all local variables needed in different levels of the call graph are mapped to the local memory in the protected zone. The local memory is an extension to the processor core, and therefore we first compute the amount of memory needed for all operations in the graph. The dashed lines in the figure represents memory requests by the operations in each level. In our implementation, operations in the same level of the graph share the same memory locations to optimize the memory usage.

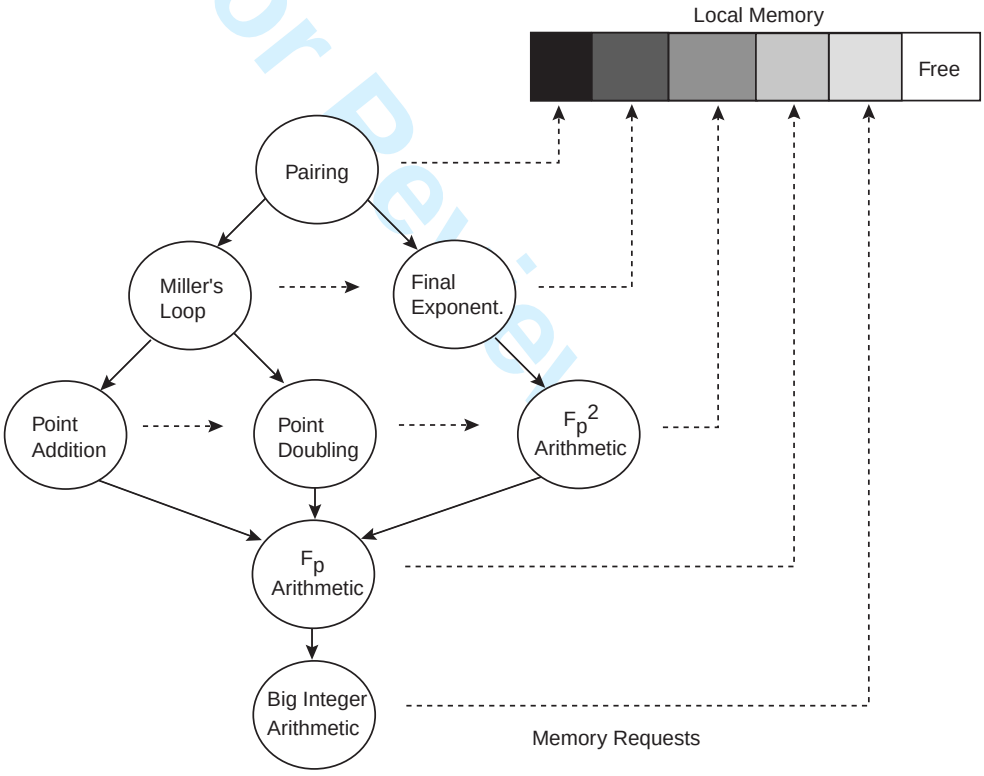

Figure 6: Call graph of pairing computation and local memory usage