System design methodologies for a wireless security processing platform

Srivaths Ravi, Anand Raghunathan, Nachiketh Potlapally and Murugan Sankaradass
C & C Research Labs, NEC USA, Princeton, NJ 08540
{sravi,anand,nachiketh,murugs}@nec-lab.com

Abstract

Security protocols are critical to enabling the growth of a wide range of wireless data services and applications. However, they impose a high computational burden that is mismatched with the modest processing capabilities and battery resources available on wireless clients. Bridging the security processing gap, while retaining sufficient programmability in order to support a wide range of current and future security protocol standards, requires the use of novel system architectures and design methodologies.

We present the system-level design methodology used to design a programmable security processor platform for next-generation wireless handsets. The platform architecture is based on (i) a configurable and extensible processor that is customized for efficient domain-specific processing, and (ii) layered software libraries implementing cryptographic algorithms that are optimized to the hardware platform. Our system-level design methodology enables the efficient co-design of optimal cryptographic algorithms and an optimized system architecture. It includes novel techniques for algorithmic exploration and tuning, performance characterization and macro-modeling of software libraries, and architecture refinement based on selection of instruction extensions to accelerate performance-critical, computation-intensive operations. We have designed a programmable security processor platform to support both public-key and private-key operations using the proposed methodology, and have evaluated its performance through extensive system simulations as well as hardware prototyping. Our experiments demonstrate large performance improvements (e.g., 31.0X for DES, 33.9X for 3DES, 17.4X for AES, and upto 66.4X for RSA) compared to well-optimized software implementations on a state-of-the-art embedded processor.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General- *System architectures*; C.1.0 [**Computer Systems Organization**]: Processor architectures- *General*; C.2.0 [**Computer Systems Organization**]: Computer-Communication Networks- *General, Security and protection*; C.5.3 [**Computer Systems Organization**]: Computer System Implementation- *Microcomputers, Portable devices*; E.3 [**Data**]: Data encryption- *DES, Public key cryptosystems*

General Terms

Security, Performance, Design, Algorithms

Keywords

Security, Security processing, Encryption, Decryption, Wireless, Handset, Embedded system, Performance, DES, 3DES, AES, RSA, SSL, IPSec, Design methodology, Platform, System architecture

1. INTRODUCTION

A large fraction of the applications and services that are of interest to Internet users involve access to, and transmission of, sensitive information (*e.g.*, e-commerce, access to corporate data, virtual private networks, online banking and trading, multimedia conferencing, *etc.*), making security a serious concern [1, 2]. The deployment of high-speed wireless data and multimedia communications ushers in even greater security challenges. Wireless communication relies on the use of a public transmission medium, making the physical signal easily accessible to malicious entities. Surveys of current and potential users of mobile commerce (m-commerce) services have indicated security concerns as the single largest bottleneck to their adoption [3].

Several security mechanisms have been developed for wired and wireless networks, based on providing security enhancements to various layers of the protocol stack (*e.g.*, IPSec at the network layer, SSL/TLS and WTLS at the transport layer, SET at the application layer, *etc.*) [4, 5]. While the above mechanisms provide satisfactory security if utilized appropriately, there is a critical bottleneck that impedes their use to address security concerns in wireless networks. Wireless clients (*e.g.*, smart phones, PDAs) are, and will always be, much more resource (processing capability, battery) constrained than their wired counterparts. On the other hand, security protocols significantly increase computational requirements at the network clients and servers [6, 7, 8] to levels that exceed the capabilities of wireless handsets. For example, a PalmIIIx handset requires 3.4 minutes to perform 512-bit RSA key generation, 7 seconds to perform digital signature generation, and can perform (single) DES encryption at only 13kpbs, assuming that the CPU is completely dedicated to security processing [8]. Further, security processing has been reported to rapidly drain the Palm's batteries [8].

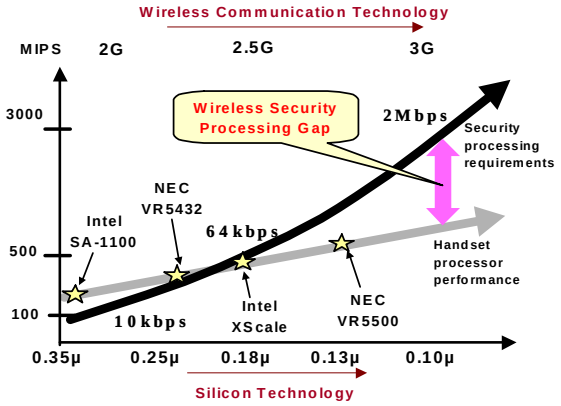


Figure 1: The security processing gap: Projected trends in security processing requirements and embedded processor performance

The increase in data rates (due to advances in wireless communication technologies), and the use of stronger cryptographic algorithms (to stay beyond the extending reach of malicious entities) threaten to further widen the gap between security processing requirements and embedded processor performance (the "*security processing gap*"). Figure 1 compares the projected trends in computational requirements (MIPS) for security processing, and the increase in embedded processor performance (enabled by improvements in fabrication technology and innovations in embedded processor architecture). The inadequate performance of embedded processors in processing security protocols leads to high network transaction latencies, and low effective data rates. Another critical bottleneck to security processing on wireless handsets is battery capacity, whose growth (5-8% per year) is far slower than the growth in processing requirements or processor performance [9]. In practice, various metrics such as performance, power, and cost, need to be considered together and it is their interaction that poses the toughest challenges to the system designer. For example, power and cost are the main reasons why embedded processors for wireless handsets are slower than their desktop counterparts. The proposed system design methodology and security processing platform architecture result in large improvements in performance as well as energy efficiency. However, space restrictions dictate that the discussions in this paper be limited to performance issues.

Algorithm-specific custom hardware implementations can always provide the highest levels of efficiency [10, 11, 12, 13]. However, in practice, the need for efficiency in security processing has to often be considered together with, and traded off against, the need for flexibility. Each security protocol standard typically specifies a wide range of cryptographic algorithms that the network servers and clients need to execute in order to facilitate inter-operability [4, 5]. Further, a security processor is often required to execute multiple distinct security protocol standards in order to support (i) security processing in different layers of the network protocol stack (*e.g.*, WEP, IPSec, and SSL), or (ii) inter-working among different networks (*e.g.*, an appliance that needs to work in both 3G cellular and wireless LAN envi-

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

ronments). Finally, programmability is desirable in order to allow easy adaptation to future security protocols and evolving standards. Hence, novel technologies to alleviate the computational burden of security processing while maintaining sufficient programmability are required.

1.1 Paper overview and contributions

We are developing a programmable security processor platform to enable secure data and multi-media communications in next-generation wireless handsets. The objective is to enable secure communications at data rates provided by 3G cellular (100 kbps - 2 Mbps) and wireless LAN (10 - 55 Mbps) technologies, while allowing for easy programmability in order to support a wide range of current and future security protocol standards. As explained above, the growth in computational requirements for security processing outstrips improvements in embedded processor performance, resulting in a significant performance gap. We believe that the use of novel system architectures and system-level design methodologies is critical to bridge this gap.

The system architecture of our security processing platform consists of

- A state-of-the-art commercial configurable and extensible processor (the Xtensa processor from Tensilica Inc. [14]) that is customized for efficient domain-specific processing, while retaining sufficient programmability, and
- Layered software libraries implementing cryptographic algorithms that are optimized and tuned to the underlying hardware platform.

Our system-level design methodology is based on the co-design of optimal cryptographic algorithms and an optimized system architecture. It allows the system designers to efficiently match the software to the characteristics of the hardware platform, and vice-versa. Our methodology includes novel techniques for algorithmic exploration and tuning as well as architecture refinement.

Concurrent development of the security algorithms and the underlying hardware architecture requires that the performance of algorithms be evaluated using either hardware models or instruction set simulation (ISS) models. In such a scenario, algorithmic exploration may be infeasible due to the size of the algorithm space, and the amount of time required to simulate realistic network transactions with hardware models. For example, simulating a single transaction of the SSL handshake protocol over a space of 495 RSA algorithm configurations would require over a month of simulation time with ISS models of the Xtensa processor, on a 440MhZ Sun Ultra 10 workstation with 1 GB memory. We propose a novel methodology to enable efficient and accurate exploration of the algorithm space, based on automatic performance characterization and macro-modeling of software functions that implement the various atomic steps in the cryptographic algorithm.

Architecture exploration is performed in our design flow through the generation and selection of custom instructions that accelerate performance-critical, computation-intensive operations. For programs where several distinct parts (*e.g.* functions) need to be accelerated through custom instructions, the large number of candidate sets of custom instructions make it difficult to evaluate all possibilities explicitly. The problem is further complicated by the fact that, it is often possible to have several different alternative custom instructions for accelerating a single sub-program, which present a tradeoff between the performance improvement and the overheads incurred by the hardware additions. We have developed techniques to automate the selection of custom instructions from a given candidate set, while considering the performance *vs.* hardware overhead tradeoffs.

We have designed a programmable security processor platform to support both private-key (*e.g.*, DES, 3DES, AES) and public-key (*e.g.*, RSA, ElGamal) operations, using the proposed methodology. We have evaluated the performance of the security processor through extensive system simulations, and through hardware implementation using a prototyping platform. Our experiments demonstrate large performance improvements (*e.g.* 31.0*X* for DES, 33.9*X* for 3DES, 17.4*X* for AES, and upto 66.4*X* for RSA) compared to well-optimized software implementations on a state-of-the-art embedded processor. We believe that system-level design methodologies, such as the one proposed here, are critical to overcoming the challenges encountered in security processing on wireless handsets.

2. OVERVIEW OF THE SECURITY PROCESSING PLATFORM

Figure 2 presents an overview of the target system architecture for our security processor platform. Efficient security processing is attained in this architecture through (i) the use of a configurable and extensible processor that is customized through the selective addition of custom instructions, co-processors, and peripherals, which implement performance-critical, computation-intensive operations, and (ii) optimized software libraries that are derived through extensive algorithmic exploration and tuning of the cryptographic algorithms they implement.

2.1 HW Platform architecture

The hardware platform is based on the Xtensa T1040 processor from Tensilica, Inc. [14]. The Xtensa features a 32-bit RISC-like base processor architecture. It offers a wide range of options to configure the base processor, including selection of generic instructions (*e.g.*, hardware multiplier, MAC, floating point unit, *etc.*), exceptions and interrupt mechanisms, endianness, register window customization, cache and memory interface configuration, debug and test hardware, *etc.* In addition to its configurability, the Xtensa also provides the designer with the ability to extend the instruction set through the addition of custom instructions that execute on designer-specified custom hardware units, which are tightly integrated into the processor execution pipeline. In our work, we exploit the customizability offered by the Xtensa processor platform in order to meet our performance objectives for security processing. HW/SW partitioning at the granularity of custom instructions often results in satisfactory performance improvements. However, in some cases, the characteristics of the application require that more coarse-grained functions be mapped to custom hardware. In such cases, one option is to use a HW co-processor that interfaces to the Xtensa's single-cycle cache interface. Alternatively, HW units that do not require high-performance communication with the processor core can be implemented as peripherals connected to the processor bus. In our work, we attempt to use custom instruction extensions to the maximum extent possible since they allow for easier integration, and facilitate higher levels of programmability and HW re-use, compared to the co-processor and peripheral options.

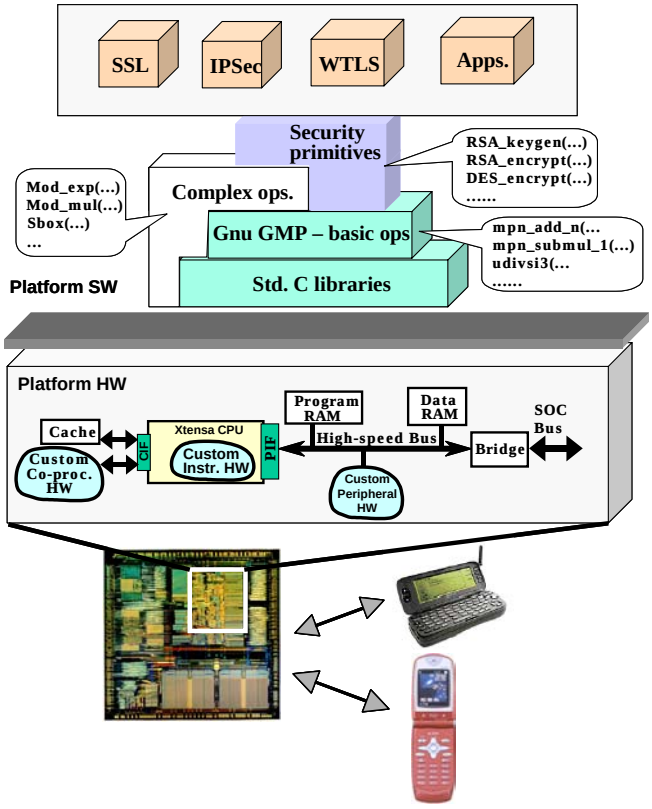


Figure 2: Overview of the target security processing system architecture

2.2 SW architecture

The choice of a suitable software architecture is critical to enable an efficient system design methodology. The software architecture for our security processor platform was designed using a layered philosophy, much like the layering used in the design of network protocols [15]. At the top level, the SW architecture provides a generic interface (API) using which security protocols and applications can be ported to our platform. This API consists of security primitives such as key generation, encryption, or decryption of a block of data using a specific public- or private-key cryptographic algorithm (*e.g.* RSA, ECC, DES, 3DES, AES, *etc.*). The security primitives are implemented on top of a layer of complex mathematical operations such as modular exponentiation, prime number generation, Miller-Rabin primality testing *etc.* [4]. These complex operations are in turn decomposed into basic mathematical operations, including bit-level operations (typically used in private-key algorithms) and multi-precision operations on large integers (typically used in public-key algorithms). The advantages of using the layered SW architecture approach include:

- The API interface at each software layer was fixed before implementation, allowing the design of each layer, and the porting of security protocols to our platform, to proceed concurrently. This reduced design time significantly, and enabled the use of more realistic application workloads to drive the design of each SW layer early in the

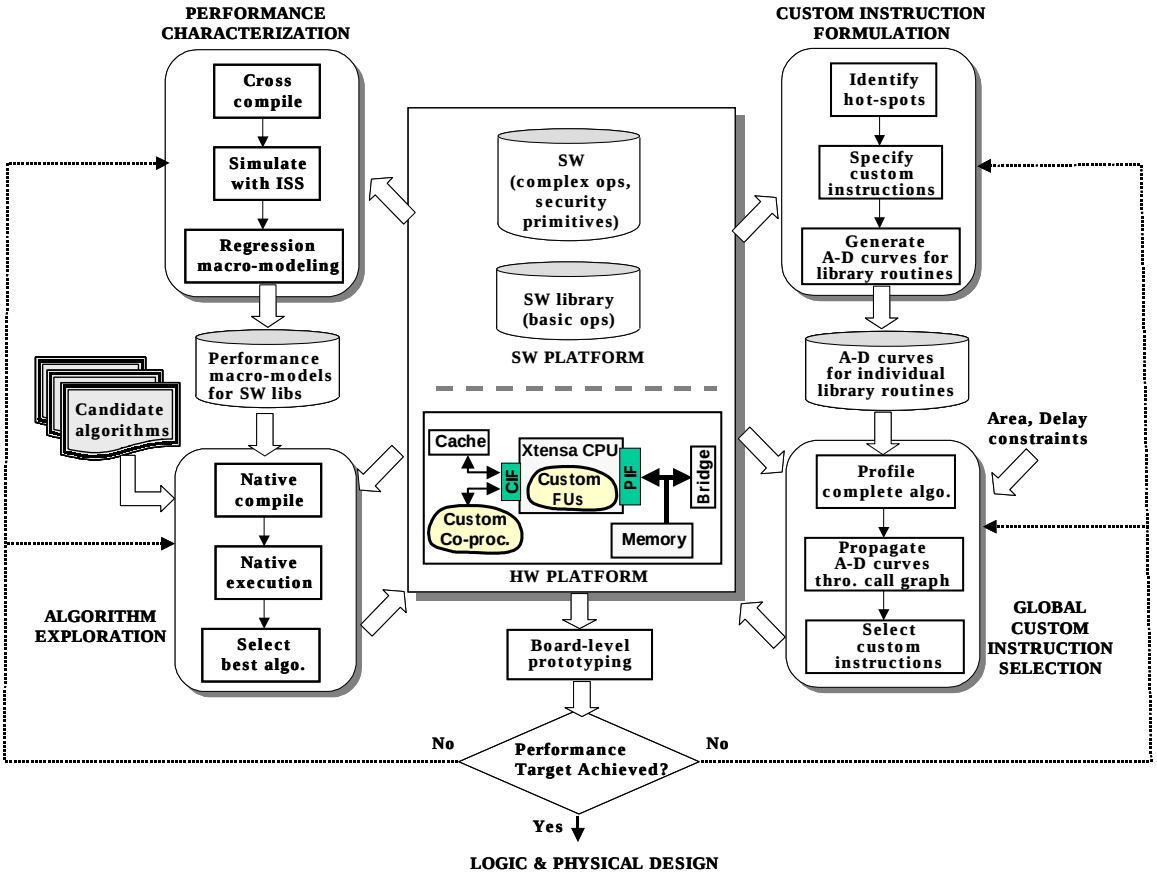


Figure 3: Overview of the security processing system design methodology

design process.

- The generation of candidate custom instructions could proceed once the software layer implementing basic operations was available (*i.e.*, without waiting for the entire SW implementation), since computations of the desired granularity are exposed in the basic operations.
- The separation of the top-level algorithms from the primitives or building blocks that are used to implement them enabled us to characterize the primitives and derive high-level performance macro-models, which were then used for efficient algorithmic exploration As illustrated in Section 3.2, this novel performance characterization methodology enabled the efficient exploration of a large number of candidate algorithms, which would have required several months of simulation time using ISS models.

3. DESIGN METHODOLOGIES

In this section, we present the methodology used to design the architecture of our wireless security processing platform. Section 3.1 presents an overview of the design methodology. Section 3.2 details the selection of the software constituents of the platform, while Sections 3.3 and 3.4 describe the steps involved in customizing the hardware platform.

3.1 Overview

Figure 3 outlines the system-level design flow used for our security processing platform. There are four major phases in the flow: (i) performance characterization of software libraries, (ii) algorithm exploration, (iii) formulation of candidate custom instructions to accelerate individual library routines, and (iv) global custom instruction selection to generate the required performance for each security algorithm. The methodology exploits the layered SW architecture in order to separate the above steps in a clean manner. Specifically, only implementations of the lower SW layers (standard libraries, basic operations) are required for performance characterization and formulation of custom instruction candidates, while algorithm exploration and global custom instruction selection are performed using the higher SW layers (complex operations, security primitives) while regarding the lower SW layers as a black box.

We now briefly describe the salient steps of our methodology, details of which are found in later subsections.

- The simulation time required for performance estimation is a significant bottleneck in algorithm design space exploration (in our context, several hours to few days per candidate algorithm). The performance macro-modeling phase effectively addresses this problem by enabling performance estimation through native compilation and execution, which can be orders of magnitude faster than Instruction Set Simulation. During the performance macro-modeling phase, we characterize the software library routines that constitute the basic steps of the algorithm, using a cycle-accurate ISS. We use statistical regression techniques to build macro-models that express the execution time of each routine as a function of parameters characterizing its input variables. The performance macro-modeling phase is explained in further detail in Section 3.2.
- The algorithm exploration phase attempts to identify optimal algorithmic implementations of security processing algorithms such as RSA, AES, 3DES *etc*. For each algorithm candidate, we instantiate the performance macro-models for library routines in the source code, and replace ISS runs with native compilation and direct execution on a host workstation, resulting in large speedups in simulation time. In our context, that allows exhaustive exploration of the algorithmic design space to be performed.
- In most scenarios, the optimized algorithm running on the base hardware platform does not achieve the target performance. Therefore, it becomes necessary to customize the underlying HW architecture, through custom instruction extensions in our case. During the custom instruction formulation phase, we focus on speeding up individual software library routines. That allows our designers to focus on small problem instances, where they best apply their creativity, leaving the global tradeoffs to the subsequent phase. The routine under consideration is profiled using traces derived from simulation of the entire algorithm. The computation-intensive parts of the routine are specified as a custom instruction. The hardware resources (functional units, register files, lookup tables, *etc*.) used in the custom instruction are varied to create a local area *vs.* delay tradeoff for the individual library routine. Having a rich set of alternatives is critical to achieving a high-quality solution in the global custom instruction selection phase. The custom instruction formulation phase is discussed further in Section 3.3.
- The global custom instruction selection phase determines a combination of (possibly several) custom instructions to result in maximum

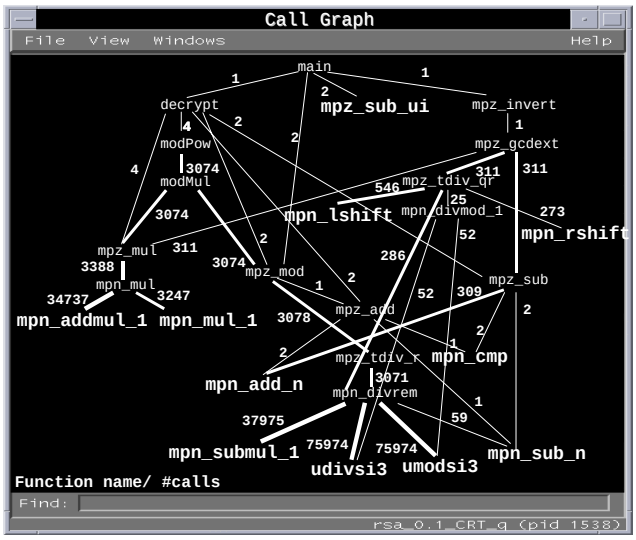


Figure 4: Call graph for an optimized modular exponentiation algorithm

speedup for the entire security algorithm subject to any applicable area constraints. This phase proceeds by propagating A-D curves for library routines through the function call graph of the entire algorithm. The potential explosion in the number of instruction combinations is contained using several techniques. The global custom instruction selection phase is described in detail in Section 3.4.

The optimized security processor platform is evaluated in the context of the target environment (*e.g.*, SSL and IPSec protocol processing, real-time video encryption/decryption, *etc.*) through board-level prototyping. Inadequacies in performance are addressed through further refinements to the HW or SW parts by iterating the steps described above with either relaxed area constraints, additional candidate algorithms, or additional custom instruction candidates.

3.2 Performance Macro-modeling for Algorithm-level Design Space Exploration

In this section, we introduce performance macro-models and describe their use in algorithmic design space exploration. A performance macro-model is a function that expresses the number of cycles incurred by the actual run of a library routine in terms of parameters that characterize the routine's input variables. For example, the performance of a routine *mpn_add_n* that adds two arbitrary length integers *in*1 and *in*2 can be expressed as a function of the bitwidths of the two inputs. The characterization process proceeds as follows. The routine under consideration is invoked in a test program that exercises it with a wide range of pseudo-randomly generated input stimuli. This test program is simulated using the cycle-accurate ISS for the target HW to generate performance data that consists of (i) the value of the parameters (*e.g.*, input variable bitwidths), and (ii) the number of execution cycles, for each invocation of the routine. A statistical regression is performed to fit the above data, resulting in the performance macro-model for the library routine. Note that, characterization is a one-time process, and results in acceleration of the overall performance estimation process.

Since the input space for a library routine can potentially be infinite, the input values used for characterization are generated to lie within a bounded super-space of the input space used by the application. For example, the GNU GMP library [16] provides a wide variety of functions that can perform arbitrary precision arithmetic on integers, rationals and floats. However, a 1024-bit RSA algorithm only requires operations restricted to (less than or equal to) 1024-bit arithmetic. Therefore, we characterize the library routines for this restricted domain only.

The performance profiles of arithmetic functions typically show a regular behavior (piecewise linear, quadratic, *etc.*) over input bit-width subspaces. Therefore, we can derive the performance model for a library routine fairly easily and accurately using regression-based approaches. All library routines instantiated in the source code of an algorithm can now be augmented with their respective performance models to allow performance estimation through native code execution on any host workstation. Further details of the performance macro-modeling technique can be found in [17].

3.3 Formulating custom instruction candidates and A-D curves

Figure 4 shows the profile statistics of an optimized modular exponentiation algorithm as a function call graph, with nodes representing function names, and edges weighted by the number of calls made to each function. For example, the function *decrypt* makes 4, 4, 2, 2 and 2 calls, to functions *mpz_mul*, *modPow*, *mpz_mod*, *mpz_add* and *mpz_sub*, respectively. Each node in the call graph may have more than one parent, since a function may be invoked by multiple higher-level functions. For example, *mpz_mul* is called by three functions - *decrypt*, *modMul*, and *mpz_gcdext*. For the sake of simplicity, the call graph in Figure 4 is truncated at functions that are highlighted with bold text, *i.e.*, calls to lower-level functions are not shown. The leaf nodes of the call graph in Figure 4 correspond to the library routines for which custom instructions are added in an interactive manner with the designer's involvement. It bears mentioning that, the granularity of the leaf nodes is a critical choice that determines the effectiveness of the custom instructions. Ideally, a function chosen to be a leaf node should contain sufficient amount of computation so as to provide scope for optimization, while being small enough that it is easy for a designer to understand and optimize. Our methodology contains heuristics for the choice of the leaf node based on the function's size and the fraction of the total program execution time it accounts for. However, we also provide the designer with an option to override automatic choices and manually specify the leaf nodes.

Since the added custom instructions can be provided with a variable number of hardware resources, we can associate an area-performance trade-off curve (also called A-D curve) with each custom instruction. The lower-most set of points in Figure 5(a) shows the A-D curve for a sample library routine *mpn_add_n* that performs the addition of two vectors. The original library routine is represented by the design point that has a zero area overhead and a performance of 202 cycles, as shown. All other design points are derived through custom instruction additions with varying number of adder resources, and hence, have non-zero area overheads. For example, the second design point is achieved by adding custom load/store instructions *load_UR*1, *load_UR*2 and *store_UR*3, and an addition instruction *add_2* that uses two 32-bit adder resources. When the number of adders is changed to 4 (*add_4*), performance improves at increased area costs, creating the next design point in the A-D curve. At some point, additional resources bring diminishing returns (*e.g.*, due to limits on parallelism or memory bottlenecks).

3.4 Global Custom Instruction Selection

In this section, we describe our methodology for selecting custom instructions using A-D curves of software library routines and the annotated call graph of the entire algorithm. Our procedure for selecting custom instructions involves combining and justifying A-D curves in a bottom-up fashion to derive a composite A-D curve for the root node of the call graph. The area and performance constraints for the platform can then be applied at the root node to pick the final custom instruction(s).

For any subgraph rooted at a node f, with children given by the set $children(f)$, the performance of f is governed by the following equation

$$cycles(f) = local\_cycles(f) + \sum_{g \in children(f)} cycles(g) \qquad (1)$$

In the above equation, $local\_cycles(f)$ refers to the number of cycles spent in computations local to f, which do not involve calls to any of its children. The above equation can be directly applied when all members of the set $children(f)$ have a single performance number associated with them (*i.e.*, no A-D curves). However, when A-D curves of one or more functions in $children(f)$ need to be combined, there are a few issues involved, as illustrated below. When the root node of a subgraph in the call graph has multiple children, the A-D curve computation simply degenerates to repeated application of the following cases.

Two child nodes - one child with an A-D curve and another with no A-D curve: Figure 5(a) illustrates this case for the graph rooted at node *root*, with one child *mpn_add_n* (which has an A-D curve), and a second child *other* (which requires 10 cycles per call). In this case, for every design point in the A-D curve of *mpn_add_n*, we have a corresponding design point in the A-D curve of *root*, with the performance computed using Equation (1).

Two child nodes with A-D curves: Figure 5(c) illustrates this case using a graph rooted at node *root* with two children, *mpn_add_n* and *mpn_addmul_*1, whose A-D curves are shown in Figures 5(a) and 5(b), respectively. As in the previous case, the performance of *root* is the sum of the performances of its children, each weighted by the number of calls made to them. In general, every combination of design points (Cartesian product) from the A-D curves of *mpn_add_n* and *mpn_addmul_*1 must be represented as a distinct point in the A-D curve of *root*. However, it turns out that whenever instructions are shared or dominated between design points, the number of design points in the composite A-D curve can be significantly reduced, as explained next.

Figure 6 shows the Cartesian product of the points on the A-D curves for *mpn_add_n* and *mpn_addmul_*1. Each entry corresponds to the union of the custom instructions that constitute the individual design points (we ignore load/store instructions, which are shared across both the children). For example, the shaded entry *add_2, mul_*1 is the union of custom instructions *add_2, mul_*1 for function *mpn_addmul_*1, and *add_*2 for function *mpn_add_n*. The symbol ϕ is used to denote the null set, *i.e.*, no custom instructions. Observe that the shaded entry *add_2, add_4, mul_*1 in Figure 6 is equivalent with many other design points. This is possible (i) when entries have the same custom instructions or (ii) when entries reduce to the same custom instructions. For example, the entry *add_2, add_4, mul_*1 has two add instructions *add_*2 and *add_*4, which differ only in the number of adder

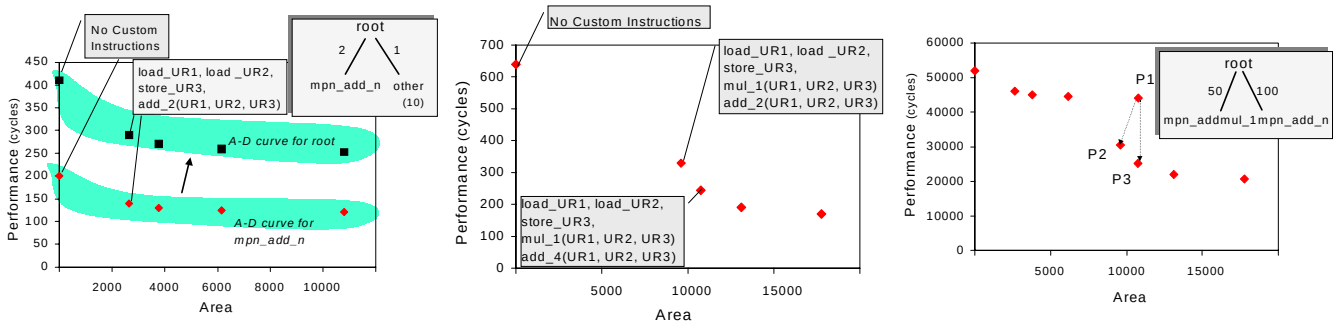


Figure 5: (a) A-D curve for library routine *mpn_add_n* and its propagation through an example call graph, (b) A-D curve for *mpn_addmul_1*, and (c) computing the A-D curve for a node with two children that have A-D curves

resources available while realizing the same functional capabilities. Given that *add_4* can be used to perform *add_2* with equal or better performance, we say that *add_4* *dominates* *add_2*, and reduce *add_2*, *add_4*, *mul_1* to *add_4*, *mul_1*. Figure 6 contains 25 candidate design points, which can be reduced to only 9 points corresponding to the shaded entries in Figure 6. The reduced set of 9 points are represented in the A-D curve for *root*, as shown in Figure 5(c).

Ø	Ø	add_2 mul_1	add_4 mul_1	add_8 mul_1	add_16 mul_1
add_2	add_2	add_2 mul_1	add_2 add_4 mul_1	add_2 add_8 mul_1	add_2 add_16 mul_1
add_4	add_4	add_2 add_4 mul_1	add_4 mul_1	add_4 add_8 mul_1	add_4 add_16 mul_1
add_8	add_8	add_2 add_8 mul_1	add_4 add_8 mul_1	add_8 mul_1	add_8 add_16 mul_1
add_16	add_16	add_2 add_16 mul_1	add_4 add_16 mul_1	add_8 add_16 mul_1	add_16 mul_1

Figure 6: Combining the design spaces of two area-delay (A-D) curves

Note that, at the root node of the entire call graph, the standard notion of Pareto-optimality can be applied to eliminate inferior points. In Figure 5(c), we can prune away design point *P*1 which has inferior performance while incurring more area with respect to design points *P*2 and *P*3.

4. EXPERIMENTAL RESULTS

We used the design methodology presented in this paper to build a security processing platform for wireless handsets that supports popular network-layer and transport-layer security protocols (*e.g.*, IPSec, SSL, WTLS, *etc.*). Section 4.1 describes the different software and hardware tools used to carry out the various steps of the methodology. Section 4.2 presents an overall evaluation of the security processing platform, including its performance in speeding up the secure socket layer (SSL) protocol. Section 4.3 discusses the results of the algorithmic design space exploration methodology, focusing on the efficiency and accuracy of the macro-modeling based performance estimation technique.

4.1 Experimental Methodology

For algorithmic design space exploration, each algorithm candidate was implemented as a highly modular, optimized C implementation using library routines from two well-known software libraries: (i) The GNU MP library [16] provides a wide variety of functions that can perform arbitrary precision arithmetic on integers, rationals and floats, and (ii) a hash library that provides a reliable means for creating hash tables. The GNU based cross-compiler, and the instruction set simulator for the target processor (an Xtensa processor core running at 188 MHz in 0.18 micron technology) were used to profile the different library routines. Performance macro-models were constructed using the statistical modeling tool S-Plus [18]. Native simulation was then performed on a SUN Ultra 10 440 MHz workstation with 1 GB of memory to select the best algorithm configuration for the given target hardware.

Custom instructions for the different library routines were implemented as Tensilica Instruction Extension (TIE) descriptions and parameterized for generating A-D curves. The TIE descriptions were compiled using the TIE compiler [14], which generates both C-stubs and synthesizable RTL Verilog descriptions. The C-stubs were then instantiated as intrinsics in test programs to derive the performance numbers in the A-D curves. The RTL descriptions were subject to logic synthesis using Synopsys Design Compiler [19] and technology mapped to the NEC CB-11 0.18 micron technology library [20] to determine the area numbers. The global instruction selection procedure described in Section 3.4 was then used to evaluate the different TIE candidates. The TIE solutions determined were combined with the base Xtensa processor core using the Xtensa processor generator [14] to build the enhanced target hardware.

Figure 7: Functional prototype of the security processing platform

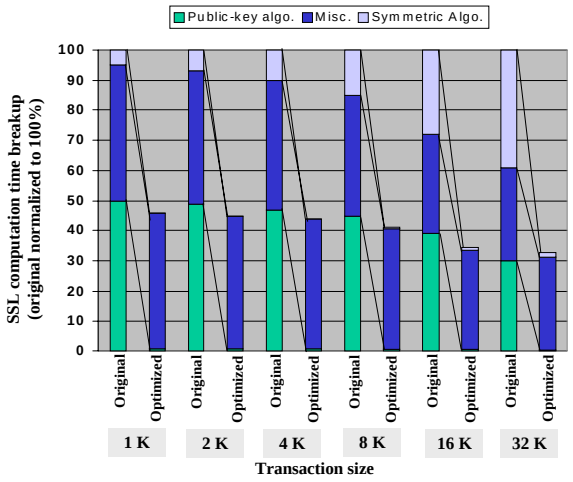
4.2 Evaluation of the security processing platform

We evaluated the performance of our security processor platform using standard implementations of private-key algorithms such as DES, 3DES, and AES, as well as the public-key algorithm RSA. The optimized HW platform and SW implementation resulting from our system design methodology were used to build a a board-level prototype implementation of the security processing platform, which is shown in Figure 7. The prototype was built using the Xtensa XT-2000 emulation board [21] with an EPSON graphics controller card [22] interfacing with an NEC LCD panel [23] (see Figure 7). The system prototype was used to demonstrate security processing performance improvements under various application scenarios, including real-time video decryption and SSL transaction acceleration.

Table 1: Performance speed-ups for popular security processing algorithms

Sec. Algo.	*Processing Rates*		
	Orig. (cycle/byte)	Final (cycle/byte)	Speedup
DES enc./dec.	476.8	15.4	31.0X
3DES enc./dec.	1426.4	42.1	33.9X
AES enc./dec.	1526.2	87.5	17.4X
RSA enc.	$34.29 * 10^3$	$3.16 * 10^3$	10.8X
RSA dec.	$12658 * 10^3$	$190.78 * 10^3$	66.4X

Table 1 illustrates the performance speed-ups for the individual security processing algorithms: 31.0X for DES, 33.9X for 3DES, 17.4X for AES,



Note: Due to large speedups in the optimized case, the public-key and private-key components are not always visible in the above graph

Figure 8: Estimated speedups for SSL transactions

and upto 66.4X for RSA. Note that, these improvements are obtained compared to already optimized software implementations. We next see how the enhancements made to these security algorithms help in speeding up the popularly used transport layer security protocol, SSL [5]. SSL uses a combination of private-key and public-key algorithms to secure the data transferred between a client and a server. The SSL handshake first allows the server and client to authenticate each other, using public-key techniques such as RSA. Then, it allows the server to create symmetric keys, which are exchanged and used for rapid encryption and decryption of bulk data transferred during the session. Figure 8 shows the estimated speedup of SSL transactions through the use of our security processing platform. The breakup of the computation workload for SSL processing between the private-key algorithm, public-key algorithm, and other miscellaneous computations, is also indicated in Figure 8. Note that, the breakup depends on the session size, hence we considered various session sizes ranging from 1KB to 32KB. For small data transactions (where public-key algorithm computations in the SSL handshake dominate), our platform contributes to an overall transaction speedup of around 2.18X. In the case of large transactions, (where the private-key algorithm starts to dominate the overall computation) our platform achieves an overall transaction speedup of 3.05X.

4.3 Algorithm design space exploration

We illustrate our algorithm design space exploration technique through the example of modular exponentiation, which is used for encryption and decryption in several public-key algorithms. Over 450 candidate algorithms were considered for evaluation due to the permutations arising from five modular multiplication algorithms, five input block sizes, three Chinese Remainder Theorem implementations, two radix sizes and three different software caching options [24]. Performance macro-model based evaluation of all the algorithm candidates completes in under 4 hours and 40 minutes. In comparison, only six algorithm candidates could be evaluated in nearly 66 hours of CPU time, using actual ISS runs. On an average, macro-model based performance estimation was found to be 1407 times faster than actual ISS runs. The performance estimated using the macro-models accurately tracked the performance profile determined by actual target simulation. The mean absolute error in the macro-model based estimates was only 11.8 %, and the relative accuracy was more found to be than adequate for the purpose of algorithm exploration.

5. RELATED WORK

Most of the efforts towards improving the efficiency of security processing have been targeted at addressing performance issues in e-commerce servers, network routers, firewalls, and VPN gateways [7, 25, 26, 27]. The fact that public key algorithms often dominate security processing requirements has driven the recent development of alternative public-key algorithms that offer reduced computational complexity [28, 29].

Various companies offer commercial security processor ICs to improve the performance of transaction servers and network routers [30, 31, 32, 33, 34, 35]. Architectural enhancements to high-end microprocessor systems to improve their performance in security processing have been investigated [25, 26]. Embedded processor designers have also developed security extensions to their products, typically based on the addition of application-specific co-processors and/or peripherals [36, 37]. Computer architects have researched domain specific instructions for security processing, with an aim to maximize efficiency without compromising programmability [38, 39]. Our target architecture and the system-level design methodologies presented here are complementary to most of the above efforts, and can enable high efficiency in security processing while maintaining programmability.

6. CONCLUSIONS

We presented the system-level design methodology used to design a programmable security processor platform for next-generation wireless handsets. The methodology was constructed using off-the-shelf commercial tools as well as novel in-house components where needed, in order to enable the efficient co-design of optimal cryptographic algorithms and an optimized HW platform architecture. Our experiments demonstrate large performance improvements (*e.g.* 31.0X, for DES, 33.9X for 3DES, 17.4X for AES, and upto 66.4X for RSA) compared to software implementations on a state-of-the-art embedded processor. We believe that system-level design methodologies, such as the one proposed here, are critical to meeting the challenging objectives and constraints encountered in security processing.

Acknowledgments: We acknowledge all brand or product names that are trademarks or registered trademarks of their respective owners. We would like to thank the members of the Tensilica support team for their invaluable assistance with the use of the Xtensa processor and tools.

7. REFERENCES

[1] U. S. Department of Commerce, *The Emerging Digital Economy II*. http://www.ecommerce.gov/ede/report.html, 1999.
[2] W. W. W. Consortium, *The World Wide Web Security FAQ*. http://www.w3.org/Security/faq/www-security-faq.html, 1998.
[3] *ePaynews*. http://www.epaynews.com/statistics/ecappstats.html.
[4] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.
[5] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.
[6] S. K. Miller, "Facing the Challenges of Wireless Security," in *IEEE Computer*, pp. 46–48, July 2001.
[7] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha, "Securing Electronic Commerce: Reducing SSL Overhead," in *IEEE Network*, pp. 8–16, July 2000.
[8] D. Boneh and N. Daswani, "Experimenting with Electronic Commerce on the PalmPilot," in *Proc. Financial Cryptography*, pp. 1–16, 1999.
[9] K. Lahiri, A. Raghunathan, and S. Dey, "Battery-driven system design: A new frontier in low power design," in *Proc. Joint Asia and South Pacific Design Automation Conf. / Int. Conf. VLSI Design*, pp. 261–267, Jan. 2002.
[10] A. G. Broscius and J. M. Smith, "Exploiting parallelism in hardware implementation of DES," in *Proc. CRYPTO'91*, pp. 367–376, 1991.
[11] A. Curiger, H. Bonnenberg, R. Zimmermann, N. Felber, H. Kaeslin, and W. Fichtner, "VINCI: VLSI implementation of the new secret-key block cipher IDEA," in *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 15.5.1–15.5.4, May 1993.
[12] C. K. Koc, "RSA hardware implementation," Tech. Rep. TR-801 (available online at http://security.ece.orst.edu/koc/ece575/rsalabs/tr-801.pdf), RSA Data Security Inc., Apr. 1996.
[13] T. Ichikawa, T. Kasuya, and M. Matsui, "Hardware evaluation of the AES finalists," in *Third Advanced Encryption Standard (AES) Conference*, Apr. 2000.
[14] *Xtensa application specific microprocessor solutions - Overview handbook*. Tensilica Inc. (http://www.tensilica.com), 2001.
[15] A. S. Tanenbaum, *Computer Networks*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
[16] T. Granlund, *The GNU Multiple Precision Arithmetic Library*. http://www.gnu.org, 2000.
[17] N. Potlapally, S. Ravi, A. Raghunathan, and G. Lakshminarayana, "Algorithm exploration for efficient public-key security processing on wireless handsets," in *Proc. DATE Designers Forum*, pp. 42–46, Mar. 2002.
[18] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S-PLUS*. Springer-Verlag, 1998.
[19] "Design Compiler, Synopsys Inc. (http://www.synopsys.com).".
[20] *CB-11 Family 0.18um CMOS Cell-based IC Design Manual*. NEC Electronics, Inc., December. 1999.
[21] *Xtensa Microprocessor Emulation Kit XT 2000 - User's Guide*. Tensilica Inc. (http://www.tensilica.com), 2001.
[22] *S1D13806 Embedded Memory Display Controller*. Epson Research & Development Inc. (http://www.erd.epson.com).
[23] *NL6448BC33-31 10.4 inch digital VGA LCD display*. NEC Electronics Inc. (http://www.necel.com).
[24] N. Potlapally, S. Ravi, A. Raghunathan, and G. Lakshminarayana, "Optimizing Public-Key Encryption for Wireless Clients," in *Proc. IEEE Int. Conf. Communications*, May 2002.
[25] Intel Corp., *Enhancing Security Performance through IA-64 Architecture*. http://developer.intel.com/design/security/rsa2000/itanium.pdf, 2000.
[26] K. Kant, R. Iyer, and P. Mohapatra, "Architectural Impact of Secure Sockets Layer on Internet Servers," in *Proc. Int. Conf. Computer Design*, pp. 7–14, 2000.
[27] A. Goldberg, R. Buff, and A. Schmitt, "Secure Server Performance Dramatically Improved by Caching SSL Session Keys," in *ACM Wksp. Internet Server Performance*, June 1998.
[28] M. Rosing, *Implementing Elliptic Curve Cryptography*. Manning Publications Co., 1998.
[29] *NTRU Communications and Content Security*. http://www.ntru.com.
[30] *Broadcom Corporation, BCM5840 Gigabit Security Processor*. http://www.broadcom.com.
[31] Corrent Inc. http://www.corrent.com.
[32] *HIFN Inc.* http://www.hifn.com.
[33] *Motorola Inc., MC190:Security Processor*. http://www.motorola.com.
[34] *NetOctave Inc.* http://www.netoctave.com.
[35] *Securealink USA Inc.* http://www.securealink.com.
[36] *ARM SecurCore*. http://www.arm.com.
[37] *SmartMIPS*. http://www.mips.com.
[38] Z. Shi and R. Lee, "Bit Permutation Instructions for Accelerating Software Cryptography," in *Proc. IEEE Intl. Conf. Application-specific Systems, Architectures and Processors*, pp. 138–148, 2000.
[39] J. Burke, J. McDonald, and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography," in *Proc. Intl. Conf. ASPLOS*, pp. 178–189, Nov. 2000.