Algorithm Exploration for Efficient Public-Key Security Processing on Wireless Handsets

Nachiketh Potlapally†, Srivaths Ravi†, Anand Raghunathan † and Ganesh Lakshminarayana‡ *
†C & C Research Labs, NEC USA, Princeton, NJ 08540
‡Alphion Corp., Eatontown, NJ 07724
{nachiketh,sravi,anand}@nec-lab.com, ganeshl@yahoo.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. In this work, we consider the task of efficient public-key security processing on wireless handsets. Our approach is based on extensive algorithmic exploration and tuning of the cryptographic algorithms that form the computational core of security protocols. In order to identify the optimum algorithm configuration, we have developed a novel performance estimation methodology based on automatic characterization and macro-modeling of software libraries, that enables us to replace target simulation with native execution during algorithm exploration. The proposed methodology results in two to three orders of magnitude speedup in the simulation time required. As a result, identifying the optimal algorithm configuration in the context of the popular SSL Handshake protocol takes less than a day, as opposed to several months using state-of-the-art processor models.

I. Introduction

The proliferation of the Internet has clearly demonstrated that a large fraction of the applications and services that are of interest to 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) [1]. This has led to the development of various mechanisms to provide security to users, by ensuring the privacy and integrity of communicated data, and the authenticity of the parties involved in a transaction [2, 3]. While security has already been established as a very serious concern in wired networks [4], the deployment of wireless communications ushers in even greater challenges. Wireless communications relies on the use of a public transmission medium, which makes the communicated signals easily accessible to malicious people or entities. This imposes new threats, in addition to security threats in the supporting wired infrastructure networks themselves. Surveys of current and potential users of mobile commerce (m-commerce) services have indicated security concerns as the single largest bottleneck to their adoption (52% of phone users and 47% of PDA users surveyed cited security as their primary concern)[5]. Several security mechanisms have been developed for the wired Internet, based on providing security enhancements to various layers of protocols (*e.g.*, IPSec at the network layer, SSL/TLS at the transport layer, SET at the application layer, *etc.*) [2, 3].

While the above mechanisms provide satisfactory security if utilized appropriately, there is a critical bottleneck that prevents their use to address security concerns in wireless networks. Wireless clients (*e.g.*, smart phones, PDAs) are much more resource (processing capability, battery) constrained than their wired counterparts. On the other hand, security protocols significantly increase computation requirements at the network clients and servers [6, 7], placing them beyond the capabilities of wireless handsets. For example, sample performance measurements [8] of the `pilotSSLeay` security libraries running on a Palm IIIx PDA [9] indicate that (i) 512-bit RSA key generation, digital signature generation, and signature verification require 3.4 minutes, 7.028 seconds, and 1.376 seconds, respectively, and (ii) DES encryption/decryption can be performed at a maximum data rate of around 13 kbps, assuming that the CPU is completely dedicated to security processing. Further, security operations are reported to quickly drain the Palm's batteries [8]. The poor performance of embedded processors in processing security protocols leads to very high network transaction latencies and low data rates. Hence, techniques to alleviate the computational burden of security processing are required.

A. Paper Overview and contributions

In this work, we present techniques to improve the performance and energy efficiency of security processing on wireless handsets. We focus on extensive algorithmic exploration and tuning of the underlying cryptographic algorithms as the mechanism to achieve these objectives. The proposed techniques are complementary to, and can be applied in conjunction with, improvements in security mechanisms, protocols, and hardware architectures [7, 10, 11, 12, 13, 14, 15, 16].

For most secure wireless transactions, the processing at the client is dominated by the public-key algorithm [7, 17]. Hence, we focus on the encryption/decryption operation used in most popular public-key algorithms [3, 18], namely modular exponentiation [1].

We present an extensive suite of algorithmic optimizations to the basic modular exponentiation algorithm, including known optimizations such as Chinese Remainder Theorem, Montgomery Multiplication, in addition to novel techniques such as input block size selection and software caching, *etc*. We formulate the various techniques as parametrizeable algorithmic optimizations, leading to a formal "algorithm design space" that is defined by the various possible algorithm configurations. We demonstrate that performance varies significantly (over an order-of-magnitude) across this space, which contains several hundred algorithmic configurations. Further, we show that the optimum algorithm configuration depends on input data characteristics, and the underlying hardware processor architecture, motivating the need for systematic algorithm exploration.

In conjunction with optimizing and tuning security algorithms, we are developing an optimized wireless security processor using

*Work done when the author was with NEC USA Inc.

[1]While our ideas are demonstrated using modular exponentiation and hence directly apply to many public-key algorithms (RSA, El-Gamal, Diffie-Helman *etc.*), the proposed methodology is equally applicable to other computation intensive security algorithms.

the extensible processor platform from Tensilica Inc. [19]. Concurrent development of the security algorithms and the underlying hardware architecture requires that the performance of algorithms be evaluated using processor models (*e. g.*, instruction set simulator (ISS) or hardware simulation 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 on hardware models. For example, simulating a single transaction of the SSL handshake protocol over a space of 495 RSA algorithm configurations would require 38 days of simulation time with instruction set simulation models of the Xtensa Processor on a 440MhZ Ultra 10 workstation with 0.5 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 modular exponentiation algorithm. The proposed methodology completes design space exploration in just 4 hours and 40 minutes, resulting in significant design time savings.

B. Organization of the paper

The rest of the paper is organized as follows. Section II presents some preliminary material on public-key algorithms. Section III then describes the parameters that affect their performance and define their design space. Section IV presents a novel methodology to completely traverse this design space for determining a performance-optimal algorithmic configuration. Section V describes the experimental results, while Section VI concludes.

II. Public-Key Algorithms

Public-key algorithms [2] perform two basic tasks: *key generation* and *encryption/decryption*. Key generation consists of generating the *private key* and the *public key*, which are used in the encryption and decryption of input data. The public key is disclosed to the world, whereas the private key is kept secret by the legitimate owner of the keys.

The key generation step is a one-time per session process, *i.e.*, keys are generated once and used for the entire length of the session [2]. Encryption/decryption constitutes bulk of the work done by a public-key cryptographic algorithm. Thus, any attempts to improve public-key algorithm performance should target this stage. In most public-key algorithms (*e.g.*, RSA, El Gamal, Diffie-Hellman, *etc.*), encryption/decryption is performed using modular exponentiation. Therefore, an optimization targeting modular exponentiation becomes applicable to most public-key algorithms.

Key generation consists of determining three quantities: the modulus (n), the public exponent (e) and the private exponent (d). The two tuples (e,n) and (d,n) constitute the public and the private key, respectively. To encrypt a message m (plaintext), we divide m into blocks $m_1, \ldots, m_p$. Then, encryption is the modular exponentiation defined by $c_i = m_i^e \bmod n$, $for\ i = 1\ to\ p$, where, c_i is the ciphertext corresponding to m_i. To decrypt a message, we take each encrypted block c_i, and compute $m_i = c_i^d \bmod n$.

III. Algorithm Design Space

Many commercial implementations of public-key algorithms exist [18], each with its choice of the data block size it operates on, the modular exponentiation core it uses, and so on. The different parameters controlling the implementation of an algorithm define its design space. The purpose of our study is to first identify the various algorithm parameters that control the implementation of modular exponentiation. With the algorithm design space defined, we not only want to identify the best values for each parameter (with respect to performance) for a particular hardware architecture, but also to examine if there is an interplay, among the various parameters, to improve the overall performance of the algorithm.

The factors that control the performance of a public-key algorithm are the size of the input block, the algorithms used for performing modular exponentiation and modular multiplication and the use of special-purpose enhancements like the Chinese Remainder Theorem. In addition, software engineering techniques can also speed up the implementation of an algorithm. We look at a specific optimization (software caches) relevant to this work. Each of the optimizations considered in this work is detailed below:

Input Block Size: Since the message is divided into blocks on which encryptpion (decryption) computations are performed, the block size affects performance. Typically, input block sizes are powers of 2 (*e.g.*, 32, 64, 128, *etc.*). To ensure that plaintext can be encrypted without data loss, the block size should be lesser than the size of the modulus.

Modular Exponentiation (ME) Algorithms: There are two ways of performing modular exponentiation [20], depending on how the bits in the exponent are scanned, namely: *left-to-right* (LR) and *right-to-left* (RL).

Chinese Remainder Theorem: Whenever the exponent size of ME is large (as in decryption), the Chinese Remainder Theorem (CRT) [21] can be used to break a single exponentiation into simpler steps. There are two ways of implementing CRT: single-radix conversion (SRC) and mixed-radix conversion (MRC) [20].

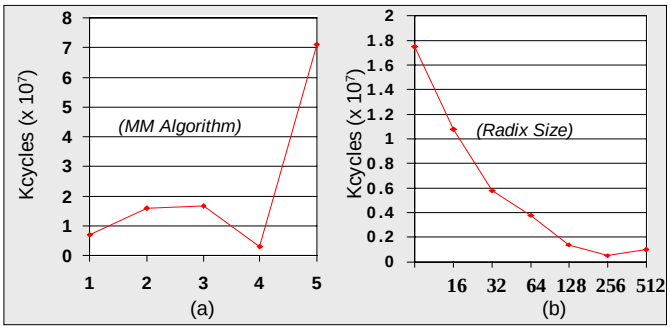

Figure 1: Performance of modular exponentiation with different (a) MM algorithms and (b)Radix sizes

Modular Multiplication (MM) Algorithms : Modular exponentiation (ME) is implemented as a sequence of modular multiplication (MM) operations. Thus, the performance of the MM operation can have a major influence on that of the ME operation (and thereby on the encryption/decryption operations). Figure 1(a) illustrates the performance of encryption/decryption using different MM algorithms in sample ME operations. ALGO 1 implements a basic Montgomery MM, with the mod operation implemented as divisions by a power of 2. ALGO 2 first computes the product and then breaks the subsequent mod operation into a series of atomic steps (each step operates on a part of the product as determined by a radix). ALGO 3 interleaves both product and mod computations. ALGO 4 uses the Karatsuba-Ofman method to first obtain the product [20] and then reduces the result using the optimized normalization method [22]. ALGO 5 is a specific instance of ALGO 3 with the radix set to 2. From Figure 1(a), we can see that ALGO 5 is the costliest. This can be explained by the large number of bitwise operations the algorithm has to perform for large inputs in order to compute the result.

Radix in MM Algorithms: The performance of the MM algorithms (Algos 2 and 3) is affected by the choice of the radix. Figure 1(b) shows the cumulative performance of encryption and decryption using Algo 3 (in ME), as the radix is varied from 8 to 512. The plot shows that minimum cost is obtained by using a radix of size 256 in this instance.

Caching: Modular exponentiation is a very costly operation and

[2] A session can be defined as a length of time for which the sender and receiver exchange information

appreciable time savings can be obtained, if the ME operation can be avoided for repeated input blocks (using the previously computed ciphertext instead). This observation prompted us to consider software caches before the ME operation (*pre-ME cache*). Software caches can also be used inside the MM units (*intra-MM cache*). Although, multiply and mod operations are not as costly as the ME operation, appreciable savings can still be obtained for a moderate hit-ratio. For example, Algo 1 has a step $M = T.N'(mod\,R)$, in which N' and R are fixed for the entire duration of encryption (or decryption). We use a cache in the following manner: **if** (T is present in the cache) **then** assign the corresponding computed value from the cache to M, **else** compute $M = T.N'(mod\,R)$ and store it in the cache.

Inter-dependencies and trade-offs: The different combinations of the parameters seen above result in a very large design space. Such a design space needs to be explored completely in order to determine the optimal choice of parameter values. This is necessary because the best-performing value for one parameter does not necessarily figure in the overall best configuration (with other parameters included) for the public-key algorithm.

For example, an input block size of 512 bits is typically considered a good choice for public-key encryption/decryption with a 1024-bit RSA modulus.With this configuration and using "algo 1", the cost of encrypting an example wireless data transaction is 64301.07 Kcycles on the target processor. On the other hand, the cost of encrypting the same transaction with a 32-bit input block size and a pre-ME cache is only 15714.5 Kcycles. This figure reflects a performance improvement of 75.5% (also includes the overhead introduced by the cache). Exploring the large design space to determine the optimal configuration of parameters, therefore, becomes inevitable.

IV. A methodology for efficient algorithmic design space exploration

In this section, we present an overview of the proposed methodology for evaluating algorithmic trade-offs in wireless security processing [3].

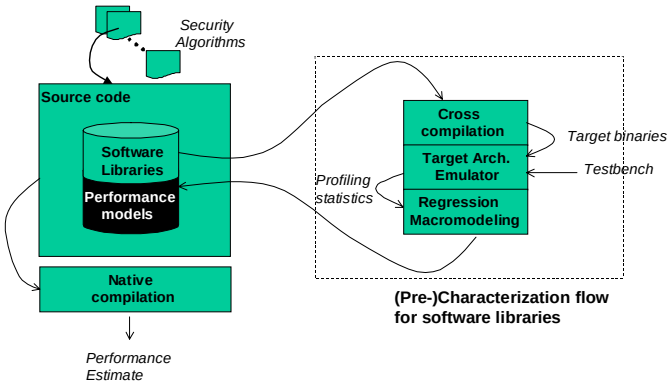

Figure 2: Enhanced architectural simulation with pre-characterized software libraries

Most algorithms, including security algorithms, are designed today as high-level entities that invoke functions from one or more pre-existing software libraries. Such an approach is used in design of our security processing platform, wherein the security algorithm sits atop a layer of software libraries , which in turn sit above the actual target architecture. As seen in Section III, there are many algorithmic choices or combinations of optimzations that must be examined so as to arrive at the *best* possible software implementation. The *best* choice is the one that requires the least number of CPU cycles, on an average.

Traditional methods of performing this evaluation would require running each candidate algorithm (serially, or, in parallel) on a target architecture ISS to derive performance metrics. Since each simulator run is expensive (see Section V)), we propose an alternative evaluation flow as shown in Figure 2. In this flow, we migrate the simulation runs to the native architecture and estimate the performance of an algorithm on the target architecture. Such a flow uses models of the software library routines that replicate (to a high degree of accuracy) their performance characteristics on the target architecture.

A performance model is a function that parameterizes the number of cycles incurred by the actual run of a library routine with some input data in terms of variables that characterize the input data. This characterization is performed by regression macromodeling (as shown in Figure 2) that takes as its input, (a) performance data of the library routine on the target for different input samples, and, (b) data values for the variables characterizing those input samples.

The performance data is collected from the profiling statistics generated by simulation runs on test programs containing the library routines for different input stimuli. This is a one-time cost, thereby accelerating the overall simulation process. Since the input space for a library routine can potentially be infinite, testbench generation is application-driven in the sense that the input samples are generated for the input space used by the application. For example, the GNU MP library provides a wide variety of C functions that can perform arbitrary precision arithmetic on integers, rationals and floats. However, a 1024-bit RSA algorithm requires only a few of those arithmetic functions with the operations restricted to (less than or equal to) 1024-bit arithmetic. Therefore, we characterize the library routines for this restricted domain only.

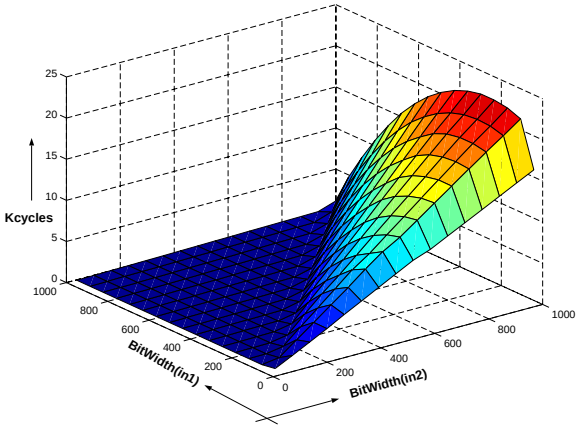

Figure 3: Performance profile of function $mod(in2, in1)$ over different input bit-widths

The performance profiles of arithmetic functions show a regular behavior (piecewise linear, quadratic, *etc.*) over input bit-width subspaces. For example, the average performance of function *mod* for different input bit-widths (the cartesian product of $BW1 : (32, 96, \cdots, 992) X BW2 : (32, 96, \cdots, 992)$) on a specific Xtensa processor configuration is shown in Figure 3. The plot indicates that a single function cannot fit the profile in an accurate manner. Therefore, the profile is partitioned along the lines $(bw1 < bw2)$, $((bw1 >= bw2)\&\&(bw2 > 32))$ and $(bw1 >= bw2)\&\&(bw2 <= 32)$. The corresponding fits obtained using S-

[3]Note that the proposed flow is general enough to be applied for exploring the algorithmic design space of other embedded software applications

PLUS [23] are indicated below.

$$
\begin{aligned}
cost &= 0.06990126 + 0.0005330226 * bw1 - 2.62605e-06 * bw2 \\
cost &= 0.3416738 + 3.998125e-05 * bw1 * bw2 - 1.450325e-06 \\
&\quad * bw1 * bw1 - 3.844676e-05 * bw2 * bw2 + 0.02121358 \\
&\quad * bw1 - 0.02028056 * bw2 \\
cost &= 0.5812022 + 0.000106492 * bw1 * bw2 + 0.01292429 * bw1 \\
&\quad - 0.02093991 * bw2
\end{aligned}
$$

The mean absolute errors of this model are very small (0.01853528, 0.01337336 and 0.128225 for the three fits). To understand the accuracy of this fit, we can compare the performance estimate for an input sample not used in the regression macormodeling process with the measured value. For example, the performance estimate for $(BW1 = 1024, BW2 = 1024)$ is 1.385 Kcycles, while an actual simulation run with 500 uniform random values averages to 1.35 Kcycles.

In this way, the performance model for a library routine can be derived 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 estimate the overall performance of the algorithm on the target architecture, while running solely through native execution.

V. Experimental Results

In this section, we tackle the task of determining an optimum configuration in the public-key algorithm design space for use in the SSL handshake protocol [3]. This section is divided into five sub-sections. Section A outlines the SSL handshake protocol, while Section B gives the implementation details of the public-key algorithms and the configuration details of the processing hardware. Section C then describes the optimized SSL handshake algorithms determined by the proposed methodology. Section D discusses the merits of the proposed design space exploration techniques.

A. Public-Key Computations in SSL handshake

The SSL handshake constitutes the initialization part of the SSL protocol. It is primarily used to authenticate the client and server, and securely exchange the key that is to be used subsequently for secure bulk data transfers. SSL handshake is dominated by public-key algorithm computations. The client is required to perform public-key operations at three stages of the SSL handshake protocol, which are:

- **Stage 1**: To verify the digital signature of the certificate authority (CA) who has signed the server certificate. This involves decryption using the public-key of the CA.
- **Stage 2**: To prepare its (client) digital signature. This is achieved by encrypting a piece of data using the private-key of the client.
- **Stage 3**: Encrypting the *pre-master secret* using the public-key of the server. The "pre-master secret" is used both by the client and the server to derive the session key.

The sizes of the data handled (encrypted or decrypted) in each stage and corresponding key sizes are given in the following Table. 1.

Parameter	Stage 1	Stage 2	Stage 3
Data Size	1024 bits	288 bits	384 bits
Key Size	16 bits	1024 bits	16 bits

Table 1: SSL handshake protocol: Characteristics of data and keys used for public-key encryption

Since the duration of the SSL handshake computations directly affects network transaction latencies (and hence battery drain due to security processing), we want to determine the best performing public-key algorithm configurations that can be used in the protocol.

B. Experimental set-up

The public-key algorithm candidates are highly modular, optimized C implementations and use library routines from two well-known software libraries: (i) the GNU MP library [22] provides a wide variety of C 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. Figure 4 plots the function call graph for a sample algorithmic configuration performing encryption. Over 450 algorithm candidates must be evaluated using the ISS model for the target handset processor, due to the permutations arising from five MM algorithms, five input block sizes, three CRT implementations (two distinct implementations, in addition to the absence of CRT), two radix sizes and three cache options (no cache, only pre-ME cache and only intra-MM cache). The target is an Xtensa configurable processor running at 214MHz, generated using Tensilica's T1030.1 processor generator [19]. The ISS model runs on top of the native development platform, which is a SUN Ultra 10 workstation with 0.5GB memory, running at 440 MHz. Simulating a single transaction of the SSL handshake protocol over the entire algorithm configuration space requires nearly 38 days of CPU time. This necessitates the efficient performance estimation methodology described in this paper.

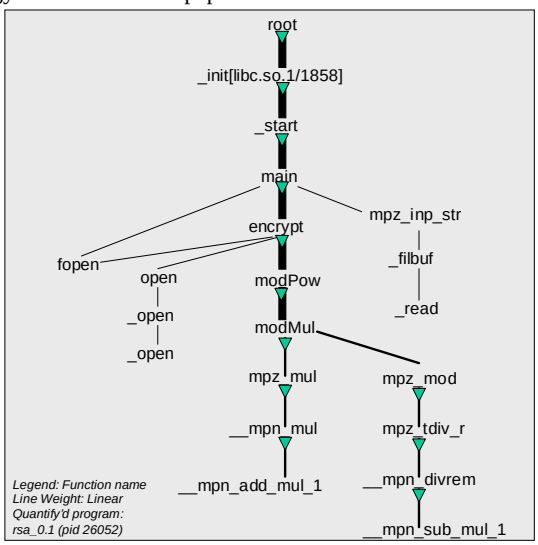

Figure 4: Function call graph for an algorithm performing public-key encryption

C. SSL Handshake Protocol: Optimal Algorithm Choice

Table 2 summarizes the results of design space exploration with the parameter values determined for optimal performance of the three public-key stages in the SSL Handshake protocol. The presence of CRT introduced a significant performance gain in Stage 2, and to a lesser degree in Stages 1 and 3. But, *single-radix conversion* (SRC) implementation of CRT results in better performance in Stages 1 and 3, while *mixed-radix conversion* method of implementing CRT performs better in Stage 2. The presence of *Pre-ME cache* did not contribute to a performance gain in any of the stages, while the *Intra-MM cache* resulted in modest gains only in Stages 1 and 3. The use of ALGO 4 for modular multiplication resulted in the best performing RSA encryption and decryption, in all the stages. Likewise, an input block size of 512 bits resulted in optimal performance across all the stages. The *radix* value applies to ALGO 2, which was observed to be the next

best performing MM algorithm. The radix value of 256 considerably improved the performance of ALGO 2 over the conventional Montgomery implementations (ALGO 1). The last row in the table indicates the overall performance gain of the optimal algorithmic configuration indicated for each stage over the conventional choice (that uses Montgomery MM algorithm, with 128-bit input block sizes [3], and radix size is fixed at 32 [24])

Parameter	Stage 1	Stage 2	Stage 3
Input Block Size	512	512	512
Radix	256	256	256
MM Algorithm	Algo 4	Algo 4	Algo 4
CRT	SRC	MRC	SRC
Pre-ME Cache	No	No	No
Intra-MM Cache	Yes	No	Yes
Speedup	74.6 %	82.9 %	66.37 %

Table 2: SSL handshake protocol: Optimal stage-wise parameter values

From Table 2, we also note that a particular set of values result in optimal performance in Stages 1 and 3, while a different set of values yield the best performance in Stage 2 (especially with respect to using the Intra-MM cache and the CRT algorithm). Table 3 gives the cost of a SSL handshake session on a wireless client using the conventional configuration, only the optimal configuration determined for *Stage 1* for all the three stages (*fixed* solution) and the optimal configuration for each stage (*adaptive*). SSL handshake incorporating optimal parameter assignment (fixed and adaptive) demonstrates nearly a 5X speedup over SSL handshake using the conventional public-key algorithm configuration. We can also see that while the difference in performances from using the *adaptive* and *fixed* solutions is not large, the *adaptive* solution comes at practically no extra cost. This observation justifies the use of an *adaptive* strategy (with the different stage parameters as defined in Figure 3) for effective execution of public-key operations in the SSL handshake protocol.

Parameter Assignment	Total Cost (Kilo Cycles)
Conventional	562115.54
Fixed	98968.86
Adaptive	98744.42

Table 3: Performance of conventional, fixed and adaptive public-key solutions to SSL Handhake Protocol

D. Efficiency and Accuracy of the Proposed Methodology

This section presents some results that demonstrate the accuracy and efficiency of performance macro-model based methodology for algorithmic design space exploration. Figure 5(a) plots the actual and estimated cycle counts per byte of input data, for six configurations in the design space of modular exponentiation. The plot shows that the performance profile determined by the proposed methodology accurately tracks the profile determined by actual target simulation.The mean absolute error in the macro-model-based estimates was only 11.8 %. Figure 5(b) indicates the corresponding speed-up in simulation time obtained by using the proposed methodology. Note that the Y-axis units are multiples of 1000 seconds. Macro-Model-based performance estimation completes for all the configurations (not just the six shown) in under 4 hours and 40 minutes. However, using target simulation, we could cover only six configurations in nearly 66 hours of CPU time. On an average, macro-model-based performance estimation was found to be 1407 times faster than target simulation.

VI. Conclusions

Existing efforts towards improving the efficiency of security processing have led to the development of many algorithmic optimizations and/or alternatives. Our work on exploration and tuning

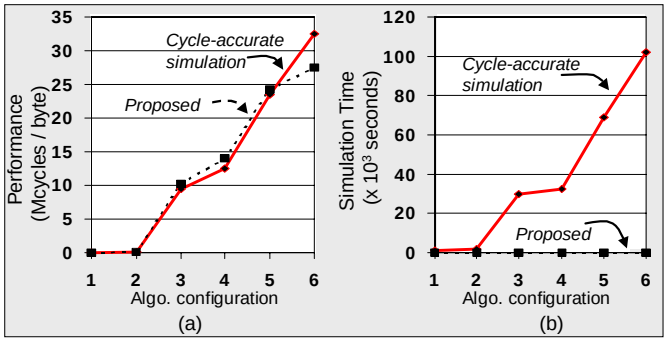

Figure 5: Accuracy (cycle count) and efficiency (simulation time) comparisons of the proposed methodology with cycle-accurate target simulation

of the algorithmic design space is complementary to these efforts since it can be applied to any such algorithm(s) running on any given hardware platform. Such techniques provide the capability needed for finding solutions that can alleviate the computational burden associated with secure wireless data communications.

References

[1] U. S. Department of Commerce, *The Emerging Digital Economy II*. http://www.ecommerce.gov/ede/report.html, 1999.

[2] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.

[3] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.

[4] W. W. W. Consortium, *The World Wide Web Security FAQ*. http://www.w3.org/Security/faq/www-security-faq.html, 1998.

[5] *ePaynews*. http://www.epaynews.com/statistics/ecappstats.html.

[6] S. K. Miller, "Facing the Challenges of Wireless Security," in *IEEE Trans. Comput.*, 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] *Palm Inc.* http://www.palm.com.

[10] 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.

[11] N. Koblitz, *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.

[12] *NTRU Communications and Content Security*. http://www.ntru.com.

[13] *ARM SecurCore*. http://www.arm.com.

[14] *SmartMIPS*. http://www.mips.com.

[15] 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.

[16] J. Burke, J. McDonald, and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography," in *Proc. Intl. Conf. ASPLOS*, pp. 178–189, Nov. 2000.

[17] Intel, *Enhancing Security Performance through IA-64 Architecture*. http://developer.intel.com/design/security/rsa2000/itanium.pdf, 2000.

[18] *RSA Security Inc.* http://www.rsa.com.

[19] Tensilica, *Xtensa application specific microprocessor solutions - Overview handbook*. http://www.tensilica.com, 2001.

[20] D. E. Knuth, *The Art of Computer Programming: Seminumerial Algorithms*. Addison Wesley, 1981.

[21] J. J. Quisquater and C. Couvreur, "Fast Decipherment algorithm for RSA public-key cryptosystems," in *Electronic Letters*, pp. 905–907, Oct. 1982.

[22] T. Granlund, *The GNU Multiple Precision Arithmetic Library*. http://www.gnu.org, 2000.

[23] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S-PLUS*. Springer-Verlag, 1998.

[24] S. R. Dusse and B. S. Kaliski, "A Cryptographic Library for the Motorola DSP 5600," in *Proc. EUROCRYPT*, pp. 230–244, 1991.