Optimizing Public-Key Encryption for Wireless Clients

Nachiketh R. Potlapally[†], Srivaths Ravi[†], Anand Raghunathan[†]
[†]C & C Research Labs, NEC USA, Princeton, NJ

Ganesh Lakshminarayana[‡*]
[‡] Alphion Corporation, Eatontown, NJ

Abstract—**Providing acceptable levels of security imposes significant computational requirements on wireless clients, servers, and network elements. These requirements are often beyond the modest processing capabilities and energy (battery) resources available on wireless clients. The relatively small sizes of wireless data transactions imply that public-key encryption algorithms dominate the security processing requirements.**

In this work, we propose techniques to improve the computational efficiency of public-key encryption algorithms. We focus on modular exponentiation based encryption/decryption, which is employed in many popular public key algorithms (*e.g.,* RSA, El Gamal, Diffie-Hellman *etc.*). We study an extensive suite of algorithmic optimizations to the basic modular exponentiation algorithm, including known optimizations such as Chinese Remainder Theorem, Montgomery Multiplication, *etc.*, and new advanced techniques such as input block size selection, computation re-use through algorithm-level caching, *etc.* The proposed algorithmic optimizations lead to an "algorithm design space", across which performance varies significantly (over an order-of-magnitude).

We evaluated the proposed algorithmic optimization techniques by obtaining processing times for the SSL handshake protocol on a state-of-the-art embedded processor, when using the optimal algorithm configuration as well as a popular conventional algorithm configuration. The results demonstrate that the optimum algorithm configuration leads to a 5.7X improvement in SSL handshake protocol processing times. The proposed techniques are complementary to, and can be applied in conjunction with, improvements in security mechanisms and protocols, new hardware architectures, and improvements in silicon technologies.

I. Introduction

The ongoing and projected future explosion of the wireless Internet promises to empower users with "anytime, anywhere, anyform" information access, computation, and communication capabilities. Wireline Internet usage has clearly demonstrated that many applications and services of interest 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], [2]. This has led to the development of various mechanisms to ensure the privacy and integrity of communicated data, and the authenticity of the parties involved in a transaction [3]. The deployment of wireless communications ushers in even greater security 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 is 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) [4].

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.*) have been developed, which provide satisfactory security if utilized appropriately [3]. However, there is a critical bottleneck that prevents their adoption 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 computation requirements at the network clients and servers [5], [6], placing them beyond the capabilities of wireless handsets. For example, sample performance measurements of the `pilotSSLeay` security libraries running on a Palm IIIx [7] indicate that [8] (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].

New wireless security mechanisms and protocols, hardware architectures optimized for security processing, and improvements in silicon technologies, will enable higher performance and battery efficiency. However, increasing data rates due to advances in wireless communications technologies (*e.g.,* 3G cellular and advanced wireless LAN systems), the accompanying deployment of multimedia data services and real-time applications, and the need for increased security levels (as hardware improvements increase the reach of crackers and malicious entities), imply that the performance and energy requirements for secure wireless communications will outstrip availability, keeping the "performance gap" and "battery gap" alive for the foreseeable future.

A. Related Work

Various specialized wireless security mechanisms and protocols exist to address some of the concerns specific to wireless networks. Current cellular standards, such as GSM, focus on providing network access security *i.e.,* they protect data only on the wireless link [9]. The emerging 3G standards also consider network domain and user domain security [10]. Correspondingly, wireless LAN protocols define link-layer security protocols (*e.g.,* WEP in the case of IEEE 802.11b [11]). Approaches to transport and higher-layer security in wireless data handsets can be broadly classified into end-to-end and gateway assisted approaches. The wireless applications protocol (WAP) defines a transport layer security mechanism (WTLS) that is optimized for the modest data rate and computation resources of current

[*] Work done when the author was with NEC USA, C&C Research Labs

handsets [12]. This enables end-to-end transport layer security between WAP clients and servers located across the wired Internet. This approach suffers from a lack of inter-operability with existing wired (non-WAP aware) secure websites and e-commerce infrastructure. Other wireless data service providers, such as Palm.Net, employ a gateway-assisted approach where the wireless gateway server runs completely customized security protocols to communicate with the wireless client, while on the other hand it employs standard wired Internet security protocols to connect to secure servers [13]. While this approach overcomes the inter-operability issue, it suffers from the drawback that data exists un-encrypted for a short duration at the wireless gateway, potentially making it vulnerable to attacks [5].

The computational challenges of security processing in wired and wireless networks have led to several efforts to address these issues. However, 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. Performance analysis of SSL based security in web servers is presented in [6], [14], [15]. The use of session caching to alleviate SSL related processing overheads is explored in [6], [16]. 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 [17], [18].

Various companies offer commercial security processor ICs to improve the performance of transaction servers and network routers [19], [20], [21], [22], [23], [24]. Architectural enhancements to high-end microprocessor systems to improve their performance in security processing have been investigated [14], [15]. Embedded processor designers have also developed security extensions to their products, typically based on the addition of application-specific co-processors and/or peripherals [25], [26]. Computer architects have researched domain specific instructions for security processing, with an aim to maximize efficiency without compromising programmability [27], [28].

Our work on exploration and tuning of the underlying cryptographic algorithms is complementary to most of the above efforts, since it can be applied to the algorithms underlying any given security protocol, and running on any given programmable hardware platform. Hence, we believe such techniques will be useful in bridging the performance and battery gaps associated with secure wireless data communications.

B. Paper Overview and contributions

In this work, we present techniques to improve the efficiency of security processing on wireless handsets. We focus on extensive algorithmic exploration and tuning of public-key encryption algorithms as the mechanism to achieve these objectives.

For most secure wireless transactions, as illustrated in Section II, the processing at the client is dominated by the public-key algorithm [6], [14]. Hence, we focus on the encryption/decryption operation used in the most popular public key algorithms, namely modular exponentiation [3], [29].

We present an extensive suite of algorithmic optimizations to the basic modular exponentiation algorithm, including known optimizations (*e.g.*, Chinese Remainder Theorem, Montgomery Multiplication) and new techniques that we have developed (*e.g.*, input block size selection, and computation re-use through algorithm-level caching). 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, motivating the need for systematic algorithm exploration. Further, we show that the optimum algorithm configuration depends on input data characteristics, and the underlying hardware processor architecture. In order to identify the optimum algorithm configuration, we employ a novel performance estimation methodology that we recently developed [30], which is based on automatic performance characterization and macro-modeling of software functions that implement the various steps in the modular exponentiation algorithm.

The rest of the paper is organized as follows. Section II motivates the need for efficient security processing, and discusses the distribution of computational effort among the various components of a security protocol. Section III briefly presents some basic concepts in public-key cryptography. Section IV describes a suite of optimizations to public-key encryption/decryption algorithms, discusses their individual impact on performance, and the inter-dependencies between the various optimizations. Section V presents the experimental methodology used for algorithm exploration, and the results obtained, and analyzes the results to give recommendations for implementing security algorithms on wireless handsets.

II. Motivation

In this section, we illustrate the need for improving the performance of public-key algorithms with the help of statistics collected from a commercial web server running secure HTTP transactions.

A secure HTTP transaction is implemented using the Secure-Sockets Layer (SSL), which is the most widely used security protocol for online transactions [3]. SSL makes use of both private-key algorithms (or symmetric algorithms) and public-key algorithms (or asymmetric algorithms) to secure the data transferred between a client and a server. In private-key algorithms, both the sender and receiver are required to have knowledge of the secret key used for encrypting the data, since the same key is used for decryption. Thus, care has to be taken to exchange the key using a very secure and trusted channel. Public-key algorithms use different keys for encryption and decryption, thereby avoiding the necessity to exchange a secret key between the sender and the receiver. But, public-key algorithms have very high computational complexity compared to private-key algorithms, which makes them infeasible for bulk data encryption and decryption. SSL achieves high security with manageable computational complexity by first using public-key algorithms to mutually authenticate the client and server, and to safely exchange a session key between the client and the server. Based on the exchanged session key, private-key algorithms are then used for fast encryption and decryption of transferred data.

Figure 1 shows the percentages of time occupied by public-key (RSA [31]) and private-key algorithms for typical SSL transactions on data of varying sizes (from 1K to 32K), running on a web server based on the Intel iA32 platform [14]. The percentage of the time spent by the web server in general book-keeping operations is labeled as "miscellaneous". For typical wireless transactions (those in the 1-8K bytes range), public-key algorithm computations on an average occupy 50 %

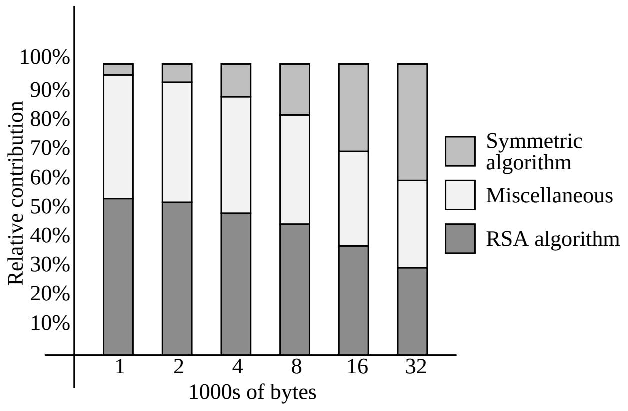


Fig. 1. Breakup of typical SSL transactions [15]

of the total processing time, whereas, computation associated with private-key algorithms consumes only about 9 % of the time. The dominance of public-key algorithm computations is even more marked on the client side when the client needs to generate a 48-byte digital signature.

Thus, improvements in the public-key algorithm performance will have a significant impact on improving the overall efficiency of e-commerce transactions. This performance improvement also translates to reduced computation energy requirements, and hence longer battery life in wireless clients. This observation provides us with a strong motivation to thoroughly explore the design space of public-key algorithms in order to discover the most efficient implementations.

III. Preliminaries

This section describes some background material on public-key algorithms for the sake of completeness. Further details are available in the literature [32].

Public-key algorithms 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 typically performed quite infrequently (*e.g.*, when generating a digital signature). 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 (using the private key or the public key). Therefore, an optimization targeting modular exponentiation becomes applicable to a wide range of 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 performed through modular exponentiation, defined by

$$c_i = m_i^e \, mod \, n, \; for \; i = 1 \; to \; p$$

where c_i is the ciphertext block corresponding to m_i. To decrypt a message, we take each encrypted block c_i, and compute

$$m_i = c_i^d \, mod \, n, , \; for \; i = 1 \; to \; p$$

IV. Public-Key Algorithm Optimizations

The most significant factors that control the performance of a public-key algorithm include 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 these optimizations can lead to several different alternative implementations of the public-key encryption algorithm. Many optimized implementations of public-key algorithms exist, however, to our knowledge, none of them consider all the algorithm optimizations in systematic manner. In order to provide a global view of the space of all possible algorithm configurations, we represent each of the optimizations as an *algorithmic parameter*. The different parameters controlling the implementation of an algorithm define the *algorithm 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 value for each parameter (for a particular underlying hardware platform), but also to examine if there is an interplay, among the various parameters, which can be exploited to improve the overall performance of the algorithm.

Each of the optimizations considered in this work is detailed in the following subsections, following which we comment on inter-dependencies between the various optimizations.

A. Input Block Size

As mentioned in Section III, a plaintext message is typically divided into several input blocks before encryption. A smaller input block size would reduce the size of the input value to each modular exponentiation step (simplifying its complexity), while increasing the number of calls to modular exponentiation. The effect of input block size on performance, was studied by performing encryption and decryption for varying input block sizes, *i.e.*, 32, 64, 128, 256 and 512 (on the same input). The number of Kilo cycles per byte of input data (Kcycles per byte) consumed for encryption and decryption on an embedded processor (see Section V for details of the target processor), was used to quantify performance. The results, plotted in Figure 2(a), were obtained by adding the kilo cycles consumed by RSA encryption and decryption, for various input block sizes. Figure 2(a) shows that the greater the block size, the better the performance. But, the performance obtained for block sizes greater than 512 were not significantly greater than that obtained by a block size of 512. Note that the block size cannot be increased beyond the "modulus" (1024-bits in this case) of the public-key algorithm in order to ensure lossless encryption.

B. Modular Exponentiation (ME) Algorithms

There are two ways of performing modular exponentiation [33], depending on how the bits in the exponent are scanned, namely: *left-to-right* (LR) and *right-to-left* (RL). Suppose that the exponent e can be represented in binary form as, $(e_{k-1}e_{k-2}\ldots e_0)_2$. In encryption, the ciphertext C corresponding to the input block M (or vice-versa for decryption) is obtained as follows:

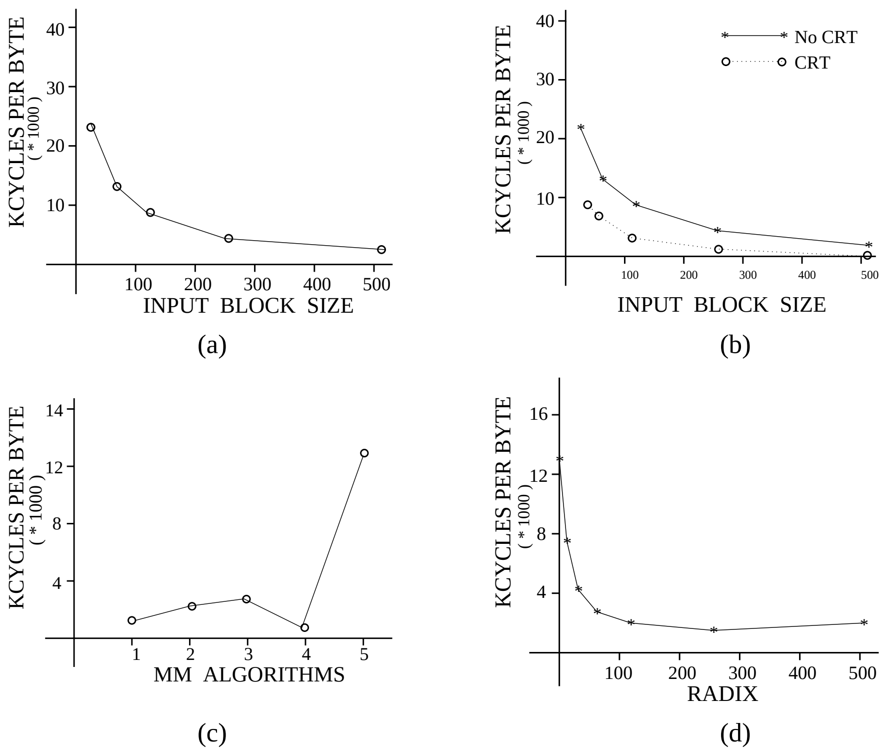


Fig. 2. Effect of (a) input block size, (b) CRT, (c) MM algorithm, and (d) radix size

- **Left-to-Right (LR) Algorithm**: Initially set $C = 1$. For i from $k-1$ down to 0, set C to $C^2 \, (mod\, n)$. In addition, if $(e_i == 1)$, set C to $C.M \, (mod\, N)$.
- **Right-to-Left (RL) Algorithm**: Initially set $C = 1$. For i from 0 up to $k-1$, set C to $C.M \, (mod\, N)$. In addition, if $(e_i == 1)$, set M to $M^2 \, (mod\, N)$.

Unlike in the LR algorithm, the operations in every iteration of the RL algorithm are independent of each other. Thus, the RL algorithm can potentially result in a speedup over the LR algorithm. However, the speedup obtained in practice depends on whether sufficient parallelism (*e.g.*, parallel MM units) is available in the target processor.

C. Chinese Remainder Theorem

The exponent size (of ME) in decryption (usually, 1024 bits) is much larger than in encryption (normally, 16 bits or less). Therefore, decryption is much more computationally intensive and time consuming than encryption. The Chinese remainder theorem (CRT) [34] is employed for reducing decryption times. Using CRT, intermediate values are obtained by performing ME using a reduced exponent size, and these values are combined to obtain the final decrypted result. This is made possible by the knowledge of the secret primes p and q (used to obtain the modulus n). There are two ways of implementing CRT, namely: single-radix conversion (SRC) and mixed-radix conversion (MRC) [33]. We describe the MRC method here. The decryption operation, $M = C^d \, (mod\, N)$ (M, C and d are the plaintext, ciphertext and private key respectively) is broken down to $M = M_1 + M_{22}.p$, where,

$$M_1 = C^{d1} \, (mod\, p) \qquad (1)$$

$$M_2 = C^{d2} \, (mod\, q) \qquad (2)$$

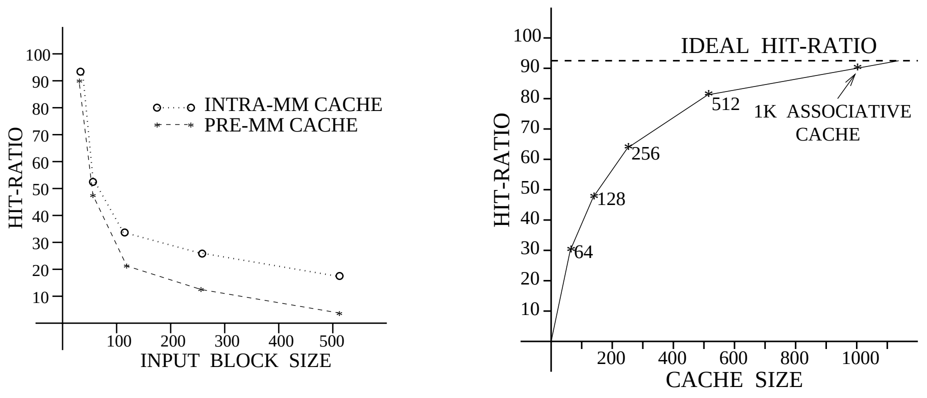
$$M_{22} = (M_2 - M_1)(p^{-1} mod\, q) \, (mod\, q) \qquad (3)$$

The values, $d_1 = d\,(mod\,(p-1))$ and $d_2 = d\,(mod\,(q-1))$, are pre-computed for a given private key d. Note that d_1 and d_2 are half the size of the private key, d, which explains the improvement obtained by CRT. Figure 2(b) illustrates the superiority of decryption using CRT (lower curve) over decryption without CRT (upper curve).

D. Modular Multiplication (MM) Algorithms

Each modular exponentiation (ME) operation is implemented as a sequence of modular multiplication (MM) operations. Each ME operation involves roughly $1.5k$ MM operations, where k is the bit-size of the exponent [35]. For example, when the exponent in ME is 1024 bits, the MM operation is invoked 1500 times, on an average, by each ME operation. Thus, the performance of the MM operation can have a major influence on that of the ME operation (and thereby on the encryption/decryption performance). There are as many ways of performing MM, as there are of performing multiplication and mod operations. Depending on the constituent operations, each MM technique has a varying impact on the performance of the encryption/decryption operations. The main trade-off among the various MM algorithms is between the speed and storage required (to hold intermediate values). In our study, five different MM algorithms were analyzed, whose details are as follows:

- **Montgomery MM (MM-Algo 1)**: This algorithm [36] implements the mod operation (reduction of the product) as divisions by a power of 2. However, there is an overhead incurred in the form of mapping the given inputs to Montgomery residue space before starting the MM computation(pre-processing), and then mapping the result back to the normal space (post-processing).
- **Radix-r, Separate Montgomery MM (MM-Algo 2)**: In this variation of Montgomery MM, the reduction of the



(a) (b)

Fig. 3. Effect of caching (pre-ME and intra-MM)

product is broken into a series of atomic steps, where each atomic step operates on a part (determined by radix r) of the product [37], *i.e.*, instead of reducing the whole product at once (as in MM-Algo 1), it is broken into chunks (determined by radix r), each of which is successively reduced. The complexity of individual operations in the algorithm is reduced, but the number of operations required increases (compared to MM-Algo 1).

- **Radix-r, Interleaved Montgomery MM (MM-Algo 3)**: In this Montgomery MM implementation, the product is accumulated in discrete steps (compared to MM-Algo 2) and successively reduced, and this process proceeds until the entire product is computed (and reduced) [37]. This implementation reduces the storage requirements (because of the partial product accumulation and reduction). The storage and computational complexity of the algorithm are reduced, but the number of steps increases (compared to MM-Algo 1).
- **Normalization based MM (MM-Algo 4)**: This algorithm involves obtaining the product using Karatsuba-Ofman method [33], and then reducing the result using the optimized normalization method [38]. Due to the absence of pre- and post-processing operations, this technique has fewer number of operations than the previous implementations (Algo's 1,2 and 3).
- **Binary Montgomery MM (MM-Algo 5)**: This is a special case of **MM-Algo 3**, where the radix is 2, *i.e.*,, $r = 2$. This particular value of the radix drastically simplifies the operations in Montgomery MM algorithm through the use of very simple and fast bitwise operations. However, the number of bitwise operations required is large.

Figure 2(c) shows the performance of encryption/decryption using the the above mentioned MM algorithms in sample ME operations. MM-Algo 5 turns out to be very costly. This can be explained by the large number of bitwise operations the algorithm has to perform, together with the poor efficiency of general purpose processors in executing bit-level operations. MM-Algo 4 performs the best.

E. Radix in MM Algorithms

The performance of MM algorithms (MM-Algos 2 and 3) is affected by the choice of the radix. Figure 2(d) shows the cumulative performance of encryption and decryption using MM-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 MM algorithms. MM-Algo 2 exhibits similar behavior.

F. Caching

Modular exponentiation is a very costly operation and 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 examine the usage of software caches before the ME operation. The encryption process in the presence of caches can be described as: **if** (M_i present in cache) **then** use C_i from the cache, **else** $C_i = M_i^e\,(mod\,N)$. Decryption can be implemented in the same way. This kind of cache is referred to as the *pre-ME cache*.

As mentioned earlier, a typical 1024-bit exponent ME operation results in 1500 MM operations on average. This increases the chances of inputs, to the costly multiplication and mod operations in the MM operation, being repeated. This motivates the use of software caches inside the MM units. 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, MM-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 $M = T.N'\,(mod\,R)$. This type of cache is called *intra-MM cache*.

Figure 3(a) shows the variation in the hit ratios of pre-ME (lower curve) and intra-MM (upper curve) caches as a function of the input block size. Intra-MM caches exhibit better performance scaling compared to pre-ME caches, as the input block size is increased. For this experiment, we assumed unlimited cache sizes, *i.e.*, the modular exponentiation result computed on each unique input block is added to the cache. Due to the overheads associated with maintaining a software cache, in practice, it is necessary to limit the cache size and consequently use a replacement policy.

In order to evaluate the cache size necessary for a good hit ratio, we performed experiments with associative cache sizes of varying sizes. The results indicate that a 1K cache results in a hit-ratio almost equal to the "ideal" hit-ratio (Figure 3(a)) for pre-ME caches (Figure 3(b)). The same behavior is observed for intra-MM caches also. Thus, 1K associative caches were used for pre-ME and intra-MM caches.

G. Inter-dependences 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, Figure 2(a) indicates that the input block size of 512 bits is potentially a good choice for public-key encryption/decryption. With this block-size (along with 1024-bit RSA modulus and "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 reduces to 15714.5 Kcycles. This figure reflects a performance improvement of 75.5% (also includes the overhead introduced by the cache). The above experiment demonstrates that performing each algorithmic optimization separately (independently) can lead to significantly sub-optimal performance. Exploring the large design space to determine the optimal configuration of parameters, therefore, becomes inevitable.

V. Public-Key Algorithms: Design Space Exploration, Results and Analyses

In this section, we tackle the task of determining an optimum configuration in the public-key algorithm design space for use in a popular handshake protocol (SSL). Section V-A describes the SSL handshake protocol and its public-key components. Section V-B outlines the implementation details of the public-key algorithms, the configuration details of the processing hardware and the performance evaluation strategy. Section V-C describes the results of our experiments, including the optimal algorithm identified therein.

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

TABLE I
SSL handshake protocol: Characteristics of public-key functions used

A. Public-Key Computations in SSL handshake

The SSL handshake constitutes the initialization part of the SSL protocol. It is primarily used to securely exchange the key (used subsequently for secure bulk data transfers) between the client and the server, and 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 Table I.

B. Experimental Methodology

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 [38] provides a wide variety of C functions that can perform arbitrary precision arithmetic on integers, rationals and floats, and (ii) a hash library which provides a reliable means for creating hash tables for use as software caches. Over 450 algorithm candidates must be evaluated due to the permutations arising from two ME algorithms, five MM algorithms, five input block sizes, three CRT implementations (two distinct implementations, in addition to the absence of CRT), 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 [39]. The processor configuration includes 4 KB direct-mapped instruction and data caches and does not include any special-purpose instructions or functional units. The instruction set simulator for the target processor 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 a space of over 450 RSA algorithm configurations requires nearly 38 days of CPU time. In order to identify the optimum algorithm configuration, we have developed a software performance estimation methodology based on automatic characterization and macro-modeling of the software library routines [30]. In this framework, performance evaluation to compute the number of cycles taken by each algorithm configuration completes in just 66 hours. Details of performance macromodeling based design exploration are available in [30].

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 II
Optimal stage-wise parameter values and speedups for the SSL handshake protocol

C. SSL Handshake Protocol: Optimal Algorithm Choice

Table II summarizes the results of design space exploration with the algorithm 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. *MM-Algo 4* 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 *MM-Algo 2,* which was observed to be the next best performing MM algorithm. The radix value of 256 considerably improved the performance of *MM-Algo 2* over the conventional Montgomery implementations (*MM-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 of 32 [37])

Table III illustrates the performance impact of replacing a single design parameter in a conventional public-key algorithmic configuration with its corresponding optimal value (Table II). We can see that by making only the input block size optimal (*i.e.*, 512 bits), performance improves by 70.5 %, 63.1 % and 62.08 % in Stages 1,2 and 3, respectively. The presence of CRT improves the performance of Stage 2 by 63 % (using MRC method), and by 32 % and 30.2 % in Stages 1 and 3 (by using SRC method). The presence of the Intra-MM cache enhances the performance of Stages 1 and 3 only.

Parameter	Stage 1	Stage 2	Stage 3
Input Block Size	70.5 %	63.1 %	62.1 %
Radix	10.6 %	11.8 %	10.5 %
MM Algorithm	43.7 %	43.2 %	45.2 %
CRT	32.0%	63.0%	30.2 %
Pre-ME Cache	-	-	-
Intra-MM Cache	5.1 %	-	4.6 %

TABLE III
EFFECT OF OPTIMAL PARAMETER VALUES ON PERFORMANCE

From Table II, 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 IV 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 parameters. 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 the *adaptive* solution 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 IV
PERFORMANCE OF CONVENTIONAL, FIXED AND ADAPTIVE PUBLIC-KEY SOLUTIONS TO SSL HANDHAKE PROTOCOL

REFERENCES

[1] U. S. Dept. 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] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.
[4] *ePaynews*. http://www.epaynews.com/statistics/ecappstats.html.
[5] S. K. Miller, “Facing the Challenges of Wireless Security,” in *IEEE Computer*, pp. 46–48, July 2001.
[6] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha, “Securing Electronic Commerce: Reducing SSL Overhead,” in *IEEE Network*, pp. 8–16, July 2000.
[7] *Palm Inc.* http://www.palm.com.
[8] D. Boneh and N. Daswani, “Experimenting with Electronic Commerce on the PalmPilot,” in *Proc. Financial Cryptography*, pp. 1–16, 1999.
[9] *European Telecommunication Standard GSM 02.09*, http://www.etsi.org /getastandard/home.htm.
[10] *3GPP Draft Tech. Spec. 33.102*, http://www.3gpp.org/specs/specs.htm.
[11] *LAN MAN Standards Committee of the IEEE Computer Society*. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification: IEEE standard 802.11, http://grouper.ieee.org/groups/802/11/main.html, 1990.
[12] WAP Forum Ltd, *Official Wireless Application Protocol*. John Wiley and Sons, 2000.
[13] *Palm Net*. http://www.palm.com/pr/palmvii/7whitepaper.pdf, 2000.
[14] Intel, *Enhancing Security Performance through IA-64 Architecture*. http://developer.intel.com/design/security/rsa2000/itanium.pdf, 2000.
[15] 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, Oct. 2000.
[16] 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.
[17] N. Koblitz, *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.
[18] *NTRU Communications and Content Security*. http://www.ntru.com.
[19] *Broadcom Corporation, BCM5840 Gigabit Security Processor*. http://www.broadcom.com.
[20] Corrent Inc. http://www.corrent.com.
[21] *HIFN Inc.* http://www.hifn.com.
[22] *Motorola Inc., MC190:Security Processor*. http://www.motorola.com.
[23] *NetOctave Inc.* http://www.netoctave.com.
[24] *Securealink USA Inc.* http://www.securealink.com.
[25] *ARM SecurCore*. http://www.arm.com.
[26] *SmartMIPS*. http://www.mips.com.
[27] 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.
[28] J. Burke, J. McDonald, and T. Austin, “Architectural Support for Fast Symmetric-Key Cryptography,” in *Proc. ASPLOS*, pp. 178–189, Nov. 2000.
[29] *RSA Security Inc.* http://www.rsa.com.
[30] N. Potlapally, S. Ravi, A. Raghunathan, and G. Lakshminarayana, “Algorithm exploration for efficient public-key security processing on wireless handsets,” in *Proc. Design and Test in Europe*, Mar. 2002.
[31] R. L. Rivest, A. Shamir, and L. M. Adleman, “A method for Obtaining Digital Signatures and Public-key Cryptosystems,” in *Comm. of the ACM*, pp. 120–126, Feb. 1978.
[32] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.
[33] D. E. Knuth, *The Art of Computer Programming: Seminumerial Algorithms*. Addison Wesley, 1981.
[34] J. J. Quisquater and C. Couvreur, “Fast Decipherment algorithm for RSA public-key cryptosystems,” in *Electronic Letters*, pp. 905–907, Oct. 1982.
[35] R. L. Rivest, “Rsa chips (past/present/future),” in *Proc. EUROCRYPT*, 1984.
[36] P. L. Montgomery, “Modular multiplication without trial division,” in *Mathematics of Computation*, pp. 519–521, 1985.
[37] S. R. Dusse and B. S. Kaliski, “A Cryptographic Library for the Motorola DSP 5600,” in *Proc. EUROCRYPT*, pp. 230–244, 1991.
[38] T. Granlund, *The GNU Multiple Precision Arithmetic Library*. http://www.gnu.org, 2000.
[39] Tensilica, *Xtensa application specific microprocessor solutions - Overview handbook*. http://www.tensilica.com, 2001.