Hardware/Software Co-design of Public-Key Cryptography for SSL Protocol Execution in Embedded Systems

Manuel Koschuch[1], Johann Großschädl[2], Dan Page[2], Philipp Grabher[2], Matthias Hudler[1], and Michael Krüger[1]

[1] FH Campus Wien – University of Applied Sciences,
Favoritenstraße 226, A–1100 Vienna, Austria
{manuel.koschuch,matthias.hudler,michael.krueger}@fh-campuswien.ac.at

[2] University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom
{johann,page,grabher}@cs.bris.ac.uk

Abstract. Modern mobile devices like cell phones or PDAs allow for a level of network connectivity similar to that of standard PCs, making access to the Internet possible from anywhere at anytime. Going along with this evolution is an increasing demand for cryptographically secure network connections with such resource-restricted devices. The Secure Sockets Layer (SSL) protocol is the current de-facto standard for secure communication over an insecure network like the Internet and provides protection against eavesdropping, message forgery and replay attacks. To achieve this, the SSL protocol employs a set of computation-intensive cryptographic algorithms, in particular public-key algorithms, which can result in unacceptably long delays on devices with modest processing capabilities. In this paper we introduce a hardware/software co-design approach for accelerating SSL protocol execution in resource-restricted devices. The software part of our co-design consists of MatrixSSL^{TM}, a lightweight SSL implementation into which we integrated elliptic curve cryptography (ECC) to speed up the public-key operations performed during the SSL handshake. The hardware part comprises a SPARC V8 compliant processor core with instruction set extensions to support the low-level arithmetic operations carried out in ECC. Our co-design executes a full SSL handshake using an elliptic curve over a 192-bit prime field in less than 300 msec when the SPARC processor is clocked at 20 MHz. A pure software implementation like OpenSSL is, depending on the field type and order, up to a factor of 10 slower than our co-design solution.

1 Introduction

The current de-facto standard for secure communication over an insecure, open medium like the Internet is the Secure Sockets Layer (SSL) protocol [9] and its successor, the Transport Layer Security (TLS) protocol [8,33]. Both use a combination of public-key and secret-key cryptographic techniques to ensure confidentiality, integrity and authenticity of communication between two parties (typically referred to as client and server). The SSL protocol is composed of two layers and includes several sub-protocols. At the lower level is the SSL Record Protocol, which specifies the format used to transmit data between client and server (including encryption and integrity checking) [9]. It encapsulates various higher-level protocols, one of which is the SSL Handshake Protocol. The main tasks of the handshake protocol are the negotiation of a set of cryptographic algorithms, the authentication of the server (and, optionally, of the client[1]), as well as the establishment of a *pre-master secret* via asymmetric (i.e. public-key) techniques [9]. Both the client and the server derive a master secret from this pre-master secret, which is then used by the record protocol to generate shared keys for symmetric encryption and message authentication [9]. The handshake protocol, on the other hand, relies on services provided by the record protocol to exchange messages between client and server.

The SSL/TLS protocol is algorithm-independent (or algorithm-agile) in the sense that it supports different algorithms for one and the same cryptographic operation, and allows the communicating parties to make a choice among them [9]. At the beginning of the handshake phase, the client and the server negotiate a *cipher suite*, which is a well-defined set of algorithms for authentication, key agreement, symmetric encryption, and integrity checking. Both SSL and TLS specify the use of RSA or DSA for authentication, and RSA or Diffie-Hellman for key exchange. In 2006, the TLS protocol was revised to include ECDSA as signature primitive and ECDH for key exchange [5]. The big benefit of Elliptic Curve Cryptography (ECC) [19] over traditional public-key schemes operating in $\mathbb{Z}_n$ or $\mathbb{Z}_p^*$ is its better security-per-bit ratio: A carefully chosen 160-bit ECC cryptosystem attains a security level comparable to that of 1024-bit RSA. As a consequence, public-key schemes based on elliptic curves over finite fields can use significantly shorter keys compared to their "classical" counterparts. These reduced key lengths translate directly into memory and bandwidth savings when SSL handshakes are performed with one of the ECC-based cipher suites from [5] instead of an RSA cipher suite. In addition, certain cryptographic operations (e.g. generation of signatures, key exchange) can be executed much faster in an elliptic curve group than in a multiplicative group like $\mathbb{Z}_p^*$.

The advent of the wireless Internet has created a strong demand for secure communication via mobile devices such as cell phones or PDAs. However, these devices are battery-operated, and hence severely constrained by computational resources (processing power, memory, network bandwidth, etc.). When implementing SSL for mobile devices, great care must be taken to utilize the scarce resources as efficiently as possible [2,3,14]. The delay a user experiences when establishing an SSL connection depends heavily on the execution time of the public-key operations carried out during the handshake (i.e. authentication and key agreement). If an RSA-based cipher suite is used, the client has to perform

[1] Most Internet applications use SSL only for server-side authentication, which means that the server is authenticated to the client, but not vice versa. Client authentication is typically done at the application layer (and not the SSL layer), e.g. by entering a password and sending it to the server over a secure SSL connection.

two modular exponentiations, one for the verification of the server's certificate and one for the encryption of the pre-master secret[2]. Even though these exponentiations involve public exponents (which are usually small), they constitute a significant overhead. For example, Gupta et al [14] analyzed the performance of an SSL client written in Java on a 20 MHz Palm Vx and found that the two RSA public-key operations account for almost 30% of the overall execution time of the SSL handshake[3]. On the other hand, when using an ECC cipher suite, the public-key operations (i.e. ECDSA verification, ECDH key exchange) make up more than 80% of the handshake time[4] [23]. Therefore, hardware acceleration of the public-key operations carried out during the handshake is desirable.

The straightforward approach to hardware acceleration of public-key cryptography is the integration of a dedicated co-processor to off-load the computationally expensive parts of an algorithm (e.g. modular exponentiation in the case of RSA, scalar multiplication in ECC) from the main processor [7,17]. In the embedded realm, however, fixed-function hardware accelerators in the form of cryptographic co-processors exhibit a number of disadvantages. Co-processors for RSA generally demand large silicon area, which poses a particular problem for low-cost embedded devices. On the other hand, co-processors for ECC often lack the flexibility to support the multitude of implementation options that are recommended by several standardization organizations around the world. One example of these options is the large number of "standardized" finite fields upon which elliptic curve cryptosystems can be built [34]. Supporting various fields of different characteristic and order is difficult with a fixed-function (i.e. hardwired) accelerator and may also consume a large amount of silicon area. Given the algorithm-agile nature of the SSL protocol, it seems questionable whether a cryptographic co-processor can meet the desired level of flexibility at moderate hardware cost. Modern security protocols, such as SSL or IPSec, are constantly evolving and hence changing their repertoire of crypto algorithms (e.g. to phase out compromised algorithms, to include new algorithms, or to adapt the minimal key size of algorithms), which again calls for a flexible and scalable approach to hardware acceleration.

In this paper we present a new methodology for hardware acceleration of the SSL handshake based on *hardware/software co-design* [35] of the involved cryptographic algorithms. The specific co-design approach we followed in our work is the integration of custom instructions into a general-purpose processor to speed up the processing of performance-critical arithmetic operations carried out in ECC (e.g. multiplication in finite fields). Hardware/software co-design at the

[2] Instead of sending a single certificate to the client, the server may also send a chain of two or more certificates linking the server's certificate to a trusted certification authority (CA). However, throughout this paper we assume that the certificate chain consists of just one certificate, and hence a single signature verification operation is sufficient to check the validity of the certificate.

[3] A 1024-bit modular exponentiation with a public exponent of 65537 executes in 1433 msec [14, Table I], and the full SSL handshake takes approximately 10 seconds.

[4] We will argue in Subsection 3.2 why ECC is advantageous over RSA for client-side SSL processing on resource-constrained devices.

granularity of instruction set extensions is particularly area-efficient and allows one to retain the full flexibility of a “pure” software solution, which makes this approach perfectly well suited for hardware acceleration of the SSL protocol in low-cost embedded systems. The hardware part of our co-design comprises an embedded SPARC V8 processor into which we integrated a set of six custom instructions to facilitate the efficient execution of arithmetic operations in prime and binary fields of large order [13]. The software part consists of MatrixSSL, a “lightweight” SSL implementation written in ANSI C [29]. MatrixSSL provides both client and server functionality, but lacks support for ECC. Therefore, we developed a simple crypto library including RSA, DSA, Diffie-Hellman, as well as ECC over prime and binary fields, and integrated it into MatrixSSL along with the ECC cipher suites from [5]. Our experimental results show that, due to the lightweight implementation of the SSL stack, the speed-up gained at the low-level field arithmetic propagates almost lossless up to the application layer. We also compare the results of our co-design with the performance figures of a pure software implementation of the SSL protocol, namely OpenSSL [28]. This comparison confirms that hardware/software co-design in the form of instruction set extensions for public-key cryptography, in particular ECC, is a good way to accelerate the SSL handshake.

2 Public-Key Cryptography

The SSL/TLS protocol makes heavy use of public-key cryptography during the handshake phase to accomplish such tasks as authentication and key establishment. In this section we briefly discuss implementation aspects of both classical public-key cryptosystems (RSA, DSA, Diffie-Hellman) as well as elliptic curve cryptosystems in the context of the SSL handshake.

2.1 RSA, DSA, Diffie-Hellman

The RSA cryptosystem operates in the residue class ring $\mathbb{Z}_n$, where n is the product of two large primes. DSA and Diffie-Hellman, on the other hand, use the multiplicative group $\mathbb{Z}_p^*$ (or a subgroup thereof) as underlying algebraic structure. The basic operation of all these cryptosystems is exponentiation, i.e. the repeated application of the ring or group operation, namely multiplication, to an element of the ring (resp. group). Of course, the multiplications are performed modulo n (or modulo p, respectively), which means that said exponentiation is actually a modular exponentiation of the form $c = m^e \bmod n$ [24]. In case of the RSA algorithm, the modulus n is a product of primes, the exponent e satisfies $\gcd(e, \phi(n)) = 1$, and the base m is in the interval $[0, n-1]$, i.e. $m \in \mathbb{Z}_n$. The security of the RSA cryptosystem is closely related to the Integer Factorization Problem (IFP), even though no mathematical proof exists that the factorization of n is needed to break RSA. Factoring an RSA modulus is widely believed to be computationally infeasible if its prime factors are large (e.g. ≥ 512 bits). On the other hand, the security of DSA and Diffie-Hellman relies on the Discrete

Logarithm Problem (DLP) in $\mathbb{Z}_p^*$, which is defined as follows: Given a generator g for $\mathbb{Z}_p^*$ (or a subgroup thereof) and an element a of said (sub)group, find the integer x such that $a = g^x \bmod p$. The DLP is considered intractable, provided that the group $\mathbb{Z}_p^*$ and the generator g are properly chosen.

The standard algorithm for computing a modular exponentiation $m^e \bmod n$ is the *square-and-multiply algorithm*, which is also referred to as binary exponentiation method [24] since it uses the binary expansion of the exponent e. Two variants of the binary method are described in [24]; one scans the bits of e from left to right (i.e. MSB first), the other from right to left (i.e. LSB first). Assuming an exponent e of length $l = 1 + \lfloor \log_2 e \rfloor$ bits, the square-and-multiply algorithm executes l modular squarings and roughly $l/2$ modular multiplications, with the exact number depending on the Hamming weight of e. The number of modular multiplications can be reduced if some extra memory for storing powers of the base m is available. For example, the *k-ary exponentiation method* (also called window method) processes k bits of the exponent e at a time, thereby reducing the number of modular multiplications to l/k in the worst case. However, the k-ary exponentiation requires pre-computation and storage of 2^k powers of the base m (see Algorithm 14.82 in [24]), which is why this method is rarely implemented on resource-constrained embedded devices like smart cards. If the base m is fixed and known a-priori (which is, for example, the case when generating a DSA signature), the number of both modular multiplications and squarings can be reduced through the *fixed-based comb method* as described in [24].

The execution time of a modular exponentiation depends heavily on the implementation of the two operations it consists of, namely modular multiplication and modular squaring. Both operations include a modular reduction, which can be efficiently performed using the well-known Montgomery technique [25]. Koç et al [22] describe several optimized software algorithms for Montgomery multiplication, among these is the so-called Coarsely Integrated Operand Scanning (CIOS) method. The CIOS method executes a total of $2s^2 + s$ single-precision (i.e. w-bit) multiply instructions, whereby n denotes the number of w-bit words that are needed to accommodate an n-bit operand, i.e. $s = \lceil n/w \rceil$ (see [22] for a detailed analysis).

The computational cost of a modular exponentiation can be reduced if the exponent e and/or the base m are suitably chosen, which is possible for RSA as well as DSA and Diffie-Hellman. For example, it is common practice to choose a small public exponent in RSA; a typical value is $2^{16} + 1$. In this case, operations involving the public exponent (e.g. RSA encryption) are significantly faster than operations involving the private exponent, even when the latter are supported by the Chinese Remainder Theorem [24]. On the other hand, Diffie-Hellman and DSA can use special primes to simplify the reduction operation. In addition, the generator g used in Diffie-Hellman key exchange can be small (e.g. $g = 2$), which reduces the cost of a modular exponentiation. DSA implementations generally take advantage of a generator g that generates a (large) subgroup of $\mathbb{Z}_p^*$, e.g. a 160-bit subgroup when p is a 1024-bit prime, which considerably alleviates the computational burden of a modular exponentiation with g as base.

2.2 Elliptic Curve Cryptography

Elliptic curve cryptosystems operate in an additive Abelian group, namely the group of points on an elliptic curve defined over a finite field. A discussion of the mathematical foundations of ECC is beyond the scope of this paper; we refer the interested reader to textbooks such as [19] and [4]. In short, ECDSA and ECDH can be seen as the elliptic-curve "equivalents" of the classical DSA and Diffie-Hellman cryptosystems, whereby the group $\mathbb{Z}_p^*$ is replaced by $E(\mathbb{F}_q)$, the group of $\mathbb{F}_q$-rational points on a curve E. The basic building block of all ECC schemes is *scalar multiplication*, i.e. an operation of the form $k \cdot P$ where P is a point on the curve (i.e. $P \in E(\mathbb{F}_q)$) and k is an integer [4]. Scalar multiplication in an additive group corresponds to exponentiation in a multiplicative group; both are performed through repeated application of the group operation to an element. However, the group operation in $E(\mathbb{F}_q)$ is the addition of points, which in turn is realized by a sequence of arithmetic operations in the underlying finite field $\mathbb{F}_q$. The security of both ECDH and ECDSA is based on the intractability of the elliptic curve discrete logarithm problem (ECDLP), which can be defined as follows: Given an elliptic curve group $E(\mathbb{F}_q)$, a base point $P \in E(\mathbb{F}_q)$, and a second point $Q \in E(\mathbb{F}_q)$, find the smallest integer k so that $Q = k \cdot P$, provided such an integer exists. Currently, the fastest algorithm known for solving the ECDLP requires fully exponential time when $E(\mathbb{F}_q)$ and the base point P were chosen with care. As a consequence, ECC schemes can use much shorter keys than their "classical'" counterparts based on the DLP or the IFP (e.g. 160 bits instead of 1024 bits) [19].

Ephemeral ECDH key exchange requires the server and the client to execute two scalar multiplications of the form $k \cdot P$; one to generate a key pair and the other to obtain the shared secret key. On the other hand, the generation of an ECDSA signature costs just one scalar multiplication $k \cdot P$, but the verifier has to execute a double scalar multiplication of the form $k \cdot P + l \cdot Q$ [4]. Similar to the square-and-multiply algorithm for exponentiation, a scalar multiplication can be carried out via *point additions* and *point doublings*, both of which, in turn, involve a sequence of arithmetic operations (i.e. addition, multiplication and inversion) in the underlying finite field $\mathbb{F}_q$ [19]. Inversion is by far the most expensive field operation. However, it is possible to add points on an elliptic curve without the need to perform costly inversions, e.g. by representing the points in *projective coordinates* [19]. When using projective coordinates, an entire scalar multiplication can be carried out solely with field additions (resp. subtractions) and field multiplications (resp. squarings); just a single inversion is necessary to convert the result from projective coordinates back to the conventional (affine) coordinate system.

Before an ECDH key exchange (or any other elliptic curve scheme) can be carried out, the involved entities have to agree upon a common set of so-called *domain parameters* [19], which specify the field $\mathbb{F}_q$, the elliptic curve E (i.e. the coefficients $a, b \in \mathbb{F}_q$ defining the curve), a base point $P \in E(\mathbb{F}_q)$ generating a cyclic subgroup of large order, the order n of this subgroup, and the co-factor

$h = \#E(\mathbb{F}_q)/n$. Consequently, elliptic curve domain parameters are simply a sextuple $D = (q, a, b, P, n, h)$.

The efficient implementation of the field arithmetic, in particular the multiplication, has a major impact on the performance of ECC cryptosystems. Prime fields and binary extensions fields are especially important since they have been recommended by numerous standards bodies around the world. The elements of a prime field $\mathbb{F}_p$ are simply the integers $0, 1, 2, \ldots, p-1$, and the arithmetic operations are addition and multiplication modulo p. Therefore, all algorithms for arithmetic in $\mathbb{Z}_p^*$ can be used for $\mathbb{F}_p$ as well, e.g. Montgomery reduction as described in [22]. However, it is possible (and common practice) to use special primes in ECC for which optimized modular reduction methods exist; a typical example are the generalized-Mersenne (PM) primes that are specified in some standards, e.g. in [34]. For example, the reduction of a 384-bit integer modulo the 192-bit GM prime $p = 2^{192} - 2^{64} - 1$ can be performed with three simple 192-bit additions. GM primes allow one to achieve better performance with the trade-off that each GM-prime requires a different reduction routine, resulting in large code size if all standardized GM-primes are to be supported.

The elements of a binary finite field $\mathbb{F}_{2^m}$ are binary polynomials of degree up to $m-1$; the arithmetic in $\mathbb{F}_{2^m}$ is polynomial arithmetic (i.e. addition and multiplication of binary polynomials) performed modulo an irreducible polynomial $p(t)$ of degree m. Addition in $\mathbb{F}_{2^m}$ is equivalent to exclusive-or and can be realized using the processor's XOR instruction on words of the operands. The major disadvantage of binary fields is that multiplication is relatively costly in software. Multiplication in $\mathbb{F}_{2^m}$ consists of a polynomial multiplication over $\mathbb{F}_2$, followed by a reduction of the product modulo the irreducible polynomial. The former is typically realized with the basic Shift-and-XOR method or one of its optimized variants such as the left-to-right comb method (see Algorithm 2.36 in [19]). Combining this method with Karatsuba's technique can be advantageous for large operands [21]. Also the reduction modulo $p(t)$ requires just shift and XOR instructions, and is relatively fast when $p(t)$ is sparse. Squaring in $\mathbb{F}_{2^m}$ is a linear operation and, hence, requires just a faction of the execution time of a multiplication.

3 Secure Sockets Layer (SSL) Protocol

The Secure Sockets Layer (SSL) protocol and its successor, the Transport Layer Security (TLS) protocol, are standardized protocol suites for enabling secure communication between a client and a server over an insecure network [8]. The main focus in the design of these protocols lay in modularity, extensibility, and transparency. Both SSL and TLS use a combination of asymmetric (i.e. public-key) and symmetric (i.e. secret-key) cryptographic techniques to authenticate the communicating parties and encrypt the data being transferred. The actual algorithms to be used for authentication and encryption are negotiated during the handshake phase of the protocol. SSL/TLS supports traditional public-key cryptosystems (i.e. RSA, DSA, Diffie-Hellman) as well as elliptic curve systems such as ECDSA and ECDH.

Table 1. SSL handshake, optional messages printed *italic*

Client	**Server**
ClientHello	
	ServerHello *Certificate* *ServerKeyExchange* *CertificateRequest* ServerHelloDone
Certificate ClientKeyExchange *CertificateVerify* ChangeCipherSpec Finished	
	ChangeCipherSpec Finished
Application Data	Application Data

3.1 SSL Handshake

The SSL protocol contains several sub-protocols, one of which is the handshake protocol. After agreeing upon a *cipher suite*[5] that defines the cryptographic primitives to be used and their domain parameters, the server (and possible the client too) is authenticated and a pre-master secret is established using public-key techniques. Table 1 shows an overview of this process (see [9] for a more detailed description). When using an RSA cipher suite, the pre-master secret is established through key transport: The client generates a random number and sends it in RSA-encrypted form to the server. On the other hand, when using an ECC-based cipher suite, the pre-master secret is established through a key exchange to which both the client and the server contribute randomness.

In the *ClientHello* message the client sends its supported cipher suites to the server, who confirms the selected suite in its own *ServerHello* message. Then, the server transmits its certificate and an optional request for authentication to the client. In most cases there is no mutual authentication and only the server presents its certificate to the client. The client is rarely authenticated during the handshake phase, but rather thereafter, e.g. by sending a password to the server. The client then verifies the server's certificate and answers with the *ClientKeyExchange* message, containing the material needed for the server to derive the shared pre-master secret. If the public key extracted from the server's certificate can not be used for encryption (e.g. because it is only authorized to signing), then the server sends a *ServerKeyExchange* message including a second public key. The *ChangeCipherSpec* is just a status message, telling both parties to use the negotiated suite from now on. The final *Finished* message is then the first one encrypted with the selected cipher and the symmetric key, derived from the pre-master secret.

[5] A cipher suite is a pre-defined combination of three cryptographic algorithms: A key exchange/authentication algorithm, an encryption algorithm, and a MAC algorithm.

The expensive steps in this process are the verification of the certificate's signature (using, for example, RSA, DSA, or its elliptic curve equivalent ECDSA) and the establishment of the shared pre-master secret, which is usually done in one of two ways, depending on the cipher suite chosen.

– If RSA is chosen, the client creates a random value, encrypts it with the server's public key (one modulo exponentiation) and sends the result back to the server, who can decrypt it (another modulo exponentiation).
– If ECDH is chosen, the client creates a random value k, calculates $R = k \cdot P$ (first scalar multiplication) and sends this value to the server. Then it calculates $k \cdot Q$ (second scalar multiplication), with Q being the server's public key, obtained from its certificate. The server finally performs a single scalar multiplication using R and its own private key. The final result for both server and client is the shared pre-master secret (see [5] for details).

3.2 Advantages of ECC Cipher Suites over RSA Cipher Suites

When an RSA cipher suite is used for the handshake, the client has to perform two modular exponentiations: one to verify the RSA signature contained in the server's certificate, and the other to encrypt the pre-master secret. Both of these exponentiations are carried out with public exponents, which are usually small [18,31]. Unfortunately, when using an ECC-based cipher suite, the situation is less favorable for the client. ECDH key exchange requires the client to execute two scalar multiplications, while the verification of an ECDSA signature involves a double-scalar multiplication of the form $k \cdot P + l \cdot Q$ [19]. ECDSA signature verification is the most costly of the cryptographic operations performed during an ECC-based handshake. In summary, RSA cipher suites impose high computational load on the server but low overhead on the client [1], while in general the opposite holds when an ECC cipher suite is used [16]. It is widely believed that RSA cipher suites are to prefer over their ECC-based counterparts when the SSL client runs on an embedded device with modest resources, while ECC cipher suites yield considerable better performance (i.e. throughput) figures on the server side [15,30]. However, there exist also numerous good arguments in favor of using ECC-based cipher suites on resource-restricted clients:

– Elliptic curve cryptosystems can use much shorter keys than RSA schemes to ensure a certain level of security (e.g. 160 vs. 1024 bits), which translates directly into memory and bandwidth savings. The former is important for low-cost devices with small memory, while the latter is relevant for mobile and battery-powered devices since wireless data transmission is very costly in terms of energy [32].
– Results from the literature confirm that a 1024-bit RSA signature can be verified significantly faster than a 160-bit ECDSA signature. However, the picture changes with higher security levels (i.e. longer keys). Brown et al [6] found that when using a Koblitz curve over $\mathbb{F}_{2^{233}}$, an ECDSA signature can be verified in 5,878 msec on a Palm V device, whereas the verification of an

RSA signature of roughly comparable strength (i.e. 2048-bit modulus and 17-bit public exponent) takes 7,973 msec.

- The key length of ECC scales linearly with that of symmetric ciphers such as the AES. For example, the NIST [26] recommends to use 128-bit AES in combination with 256-bit ECC or 3,072-bit RSA. However, 256-bit AES demands RSA keys with a length of 15,360 bits for equivalent security, while 512-bit keys suffice when using ECC. This linear scaling property makes a good case for ECC if AES-equivalent security levels are to be supported.
- Even though this paper focusses on client-side acceleration of SSL, it should be noted that ECC offers significant performance-advantages for SSL servers [15,16]. RSA cipher suites are highly computation-intensive on the server side [7,36], which may also impact the overall latency of the handshake, in particular if the server is under heavy load.
- When performing an SSL handshake with client authentication, ECC-based cipher suites are, in general, less costly than their RSA counterparts (this applies to both sides, the server *and* the client [15,30]).
- Using a cipher suite with ephemeral ECDH key exchange provides forward secrecy, whereas RSA-based key transport does not [4]. Another advantage of ECDH key exchange is that both the client and the server can contribute randomness to the generation of the pre-master secret, which is not the case with RSA-based key transport.
- ECDSA, ECDH, and ECMQV (an authenticated variant of ECDH) are the only public-key schemes included in NSA Suite B [27], i.e. RSA must not be used to secure sensitive or classified U.S. government communications.

For all these reasons we decided to use ECC cipher suites for the performance evaluation of our co-designed SSL stack. However, other cipher suites based on RSA, DSA, or Diffie-Hellman are also supported.

4 Implementation Details and Results

The hardware platform we used for our co-design is SPARC V8 softcore into which we integrated a small set of custom instructions to speed up public-key cryptography. Instruction set extension is a simple and efficient way to enhance a processor's capabilities to support special application domains. In contrast to a dedicated co-processor, the hardware overhead of custom instructions is, in general, relatively small. Moreover, since the instructions are directly integrated into the ordinary processing pipeline, there is no need for expensive operand transfers, which can heavily affect the performance of such solutions. Dedicated co-processors are also limited in terms of flexibility and usually not designed to support such a multitude of cryptographic algorithms as is needed in SSL.

Software implementations of cryptosystems often spend the majority of execution time in a few performance-critical code sections (e.g. inner loops), which makes it amenable to processor customization. It was shown in [12] that a small set of only five or six custom instructions suffices to accelerate the full domain

Table 2. Format and description of the CIS instructions for public-key cryptography

Format	Description	Operation
`umac rs1, rs2`	Unsigned Multiply & Accumulate	$accu \leftarrow accu + rs1 \times rs2$
`umac2 rs1, rs2`	Unsigned Mul. & Accumulate Twice	$accu \leftarrow accu + 2(rs1 \times rs2)$
`uaddac rs1, rs2`	Add to Accumulator Unsigned	$accu \leftarrow accu + rs1 + rs2$
`shacr rd`	Shift Accu Registers Right	$rd \leftarrow accu[31:0]; accu \leftarrow accu \gg 32$
`gf2mul rs1, rs2`	Bin. Polynomial Multiply	$accu \leftarrow rs1 \otimes rs2$
`gf2mac rs1, rs2`	Bin. Polynomial Mul. & Accumulate	$accu \leftarrow accu \oplus rs1 \otimes rs2$

of public-key primitives specified in the IEEE Standard 1363 [20]; these include traditional cryptosystems such as RSA, but also ECC systems over both prime fields and binary extension fields. Based on the ideas in [11,12], we devised the *Cryptography Instruction Set (CIS) extensions* to the SPARC V8 architecture [13] and integrated them into the LEON-2 softcore, an open-source SPARC V8 implementation developed by Gaisler Research. The LEON-2 VHDL model can be synthesized to FPGA and standard cell technologies [10].

The CIS extensions for PKC consist of a set of six custom instructions (see Table 2) and a functional unit (FU) on which the instructions are executed. This FU is basically a multiply-accumulate (MAC) unit composed of a (32×16)-bit unified multiplier and a 72-bit accumulator. A so-called unified multiplier is a multiplier that uses the same datapath for two different types of operands, namely integers and binary polynomials [12]. The MAC unit also contains three result accumulation registers (`%asr20`, `%y`, and `%asr18`), in the following called *accu registers*. Besides the six custom instructions shown in Table 2, the MAC unit is capable to execute the two "native" SPARC V8 multiply instructions `umul` and `smul`. Consequently, the CIS extensions can be easily integrated into a SPARC V8 core by replacing the original integer multiplier by a MAC unit for integers and binary polynomials and modifying the instruction decoder to support the custom instructions.

Most of the CIS instructions listed in Table 2 get two 32-bit words from the general-purpose register file as input and place the result in the accu registers. The `umac` instruction can be used to implement the inner loop of long integer multiplication according to the product scanning technique and is also useful for Montgomery multiplication [11]. Long integer squaring can be efficiently executed with help of the `umac2` instruction. The two instructions `uaddac` and `shacr` facilitate the modular reduction operation for the special primes used in EC cryptography. Finally, the instructions `gf2mul` and `gf2mac` interpret their operands as binary polynomials and perform polynomial multiply/MAC operations that allow to speed up EC systems based on binary fields. A detailed description of the custom instructions and their use in the diverse arithmetic algorithms can be found in [11,12].

Especially for binary extension fields, the presence of hardware support for polynomial multiplication offers a significant performance gain compared to a native software implementation. The additional instructions are easily accessible through a modified assembler and the use of inline assembly in ordinary

C programs. Due to their generic nature, they can be used for all sorts of cryptographic algorithms requiring fast integer or polynomial arithmetic.

We integrated the CIS extensions into the LEON-2 core and prototyped the extended processor in an FPGA. The CIS extensions have no impact one the cycle time, i.e. the extended LEON-2 can be clocked with the same frequency as the "original" LEON-2 processor (up to 50 MHz in our FPGA device). We also synthesized the extended LEON-2 using a 0.35μ standard cell library and found that the CIS extensions entail an increase in area by merely 5,550 gates compared to a baseline LEON-2 core with a (32×16)-bit multiplier.

4.1 Evaluation of Code Size and Performance

The software part of our co-designed SSL stack is based on the freely available MatrixSSL library [29]. MatrixSSL in its original form provides both client and server functionality, but does not feature ECC. Therefore, we developed a lightweight public-key cryptographic library and integrated it into MatrixSSL so as to support the ECC cipher suites specified in [5]. We used OpenSSL [28] as a reference implementation with respect to code size and performance. Similar to OpenSSL, our implementation is generic in the sense that it works for every curve over prime or binary extension fields and allows free combination of the cryptographic primitives (e.g. using ECDSA as signature primitive and RSA for key establishment). Table 3 shows a comparison between our implementation (i.e. MatrixSSL+ECC), the original MatrixSSL version (without ECC support) and the OpenSSL library in terms of source files and code size. The integration of ECC increased the size of MatrixSSL by just 15-20%. For comparison, the OpenSSL executable is almost 20 times larger.

The crypto library we integrated into MatrixSSL is realized in a very straightforward way. We used Algorithm 2.9 in [19] to implement the multiple-precision multiplication and Montgomery's well-known algorithm for modular reduction [25]. In order to keep the size of our library at a minimum, we did not include optimized reduction functions for special primes like the NIST primes. Also the curve arithmetic over $\mathbb{F}_p$ is based on well-known algorithms. We represent the elliptic curve points using the mixed Jacobian-affine coordinates described in [19, Section 3.2.2]. The scalar multiplication over $\mathbb{F}_p$ is carried out according to the double-and-add technique with non-adjacent-form (NAF) representation of the scalar to save some point additions. For ECDSA verification, Shamir's trick [19] in combination with a joint-sparse-from (JSF) representation of the scalars is used to interleave the two scalar multiplications [19]. We decided to

Table 3. Comparison of MatrixSSL, our SSL, and OpenSSL

Implementation	Number of source files	Lines of code	Size of executable
Original MatrixSSL	30	~ 9,500	114 kB
MatrixSSL with ECC	50	~ 10,900	130–140 kB
OpenSSL 0.9.8	1,100	~ 250,000	2,374 kB

not implement a window method for scalar multiplication because we aimed to keep the memory footprint at a minimum.

Also the algorithms for arithmetic in $\mathbb{F}_{2^m}$ are well documented and rather straightforward to implement. We used the so-called left-to-right comb method with windows of width 4 for the multiplication of binary polynomials [19]. Furthermore, we implemented a generic reduction function for irreducible trinomials and pentanomials. The term generic in this context means that the reduction function accepts arbitrary trinomials and pentanomials as input. In addition, we also included the Montgomery reduction for binary polynomials in our library to support irreducible polynomials which are not trinomials or pentanomials. The scalar multiplication on elliptic curves over $\mathbb{F}_{2^m}$ is performed according to the well-known algorithm of Lopez and Dahab [19].

We actually implemented two versions of the crypto library: one is written entirely in ANSI C, whereas the second contains assembly-language statements to access the custom instructions of our extended LEON-2 core. Reference [13] explains the implementation of the field arithmetic using the CIS instructions in detail. The CIS-optimized version uses Montgomery multiplication for both prime and binary fields. We refrained from the implementation of special reduction techniques for GM primes or sparse irreducible polynomials since we aimed at a "lightweight" implementation of the cryptographic primitives with small code size. The results from [13] indicate that the CIS extensions speed up the multiplication in prime fields by a factor of between two and three, whereas the multiplication in binary fields achieves a six to ten-fold performance gain. The exact speed-up factor depends on a number of implementation options (e.g. loop unrolling) and the length of the operands (e.g. when Karatsuba's technique [21] is used).

In the following, we evaluate and analyze the handshake performance of the co-designed SSL stack. As mentioned before, our implementation is generic in the sense that it supports arbitrary cipher suites and arbitrary ECC domain parameters. It is, of course, not feasible to evaluate every possible combination of cipher suites and domain parameters in this paper. Therefore, we focus on a representative example, namely an ECC cipher suite that uses ephemeral ECDH for key exchange and ECDSA as signature primitive [5]. We let our co-designed SSL stack operate as server, which means that it has to execute two scalar multiplications to establish a shared secret key. As usual, no client authentication is performed, i.e. the client does not send a certificate to the server.

Figure 1 shows the execution time (in clock cycles) of a scalar multiplication using four NIST prime fields as underlying algebraic structure. All cycle counts were measured on a LEON-2 core with CIS extensions [13], synthesized onto a Xilinx XCV-800 board, clocked at 20 MHz. When using a small field (e.g. a 160 or 192-bit field), the ANSI C version of our crypto library reaches roughly the same performance as OpenSSL, which is a remarkable result when considering that the latter features several performance enhancements such as specialized reduction methods for standardized primes, hand-written assembly code for all performance-critical operations, and code-size increasing optimizations like loop

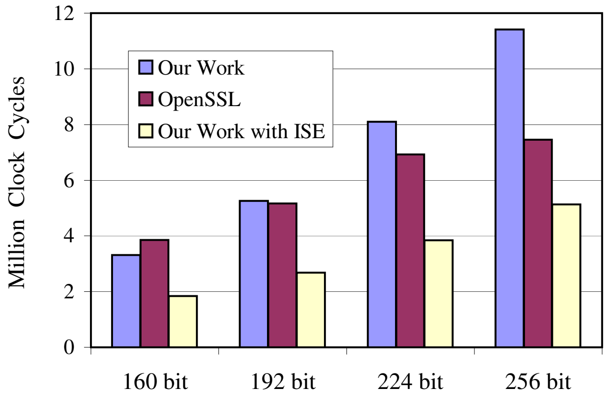

Fig. 1. Performance of scalar multiplication over prime fields

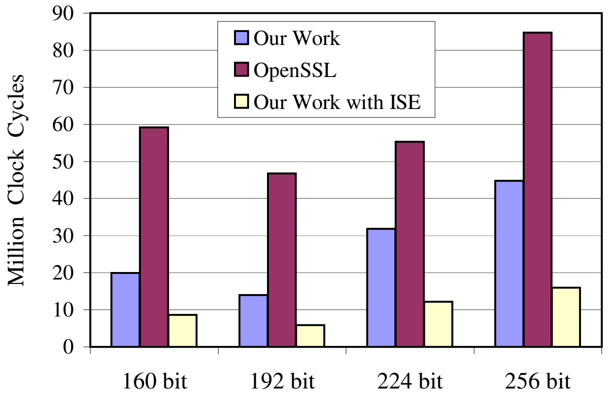

Fig. 2. Performance of entire handshake over prime fields

unrolling). On the other hand, both versions of our library perform the modular reduction according to Montgomery's algorithm [25], i.e. better timings would be possible when using one of the optimized reduction methods discussed in Section 2.2. The CIS extensions allow one to execute a full scalar multiplication over a 192-bit prime field in $2.6 \cdot 10^6$ cycles. Depending on the field size, the CIS extensions accelerate scalar multiplication by a factor of between 2.0 and 2.5.

Figure 2 illustrates that the performance gained at the field or group level propagates almost lossless all the way up to the application (i.e. the handshake) level. The CIS version of our SSL stack is again by a factor of between 2.0 and 2.5 faster than the ANSI C version that does not use custom instructions for field arithmetic. Our co-design is able to perform a full SSL handshake, from sending the first Hello message until receiving final Finished message, in less than 300 msec on a device running at 20 MHz when using a 192-bit field as underlying algebraic structure. Similar results can also be achieved for binary extension fields of roughly the same order. For comparison, OpenSSL is—depending on the field type and order—up to a factor of 10 slower than our implementation utilizing the CIS instructions. This big difference is partly due to the efficiency of our field arithmetic and partly due to the lightweight implementation of our protocol stack.

5 Conclusion

We presented a hardware/software co-design of the SSL handshake based on instruction set extensions for the low-level arithmetic operations carried out in public-key cryptography. Our solutions offers a significant gain in performance for field arithmetic as well as for an entire handshake when compared with a pure software implementation, thus allowing a handshake over a 192-bit prime field to complete in about 300 msec on a 20 MHz LEON-2 processor equipped with our CIS extensions. A single scalar multiplication over the same field takes approximately $2.6 \cdot 10^6$ cycles when using Montgomery's algorithm for the field arithmetic. Our solution requires very little additional hardware (about 5,500 gates), consumes a negligible amount of additional memory, and allows one to speed up a multitude of cryptographic algorithms, including RSA, DSA, Diffie-Hellman, as well as ECDSA and ECDH over both prime and binary fields. In addition, we have shown that the speed-up achieved in the low-level operations (i.e. the field arithmetic) propagates almost lossless up to the highest layers of the SSL protocol. So, by speeding up field multiplication and squaring using instruction set extensions, the entire high-level SSL handshake can be sped up by almost the same factor.

Acknowledgements. A preliminary version of this paper was presented at the 2nd Workshop on Embedded Systems Security (WESS 2007), which took place in Salzburg, Austria on October 4, 2007.

The research described in this paper has been supported by the EPSRC under grant EP/E001556/1 and, in part, by the European Commission through the ICT Programme under contract ICT-2007-216676 ECRYPT II. The information in this paper reflects only the authors' views, is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Manuel Koschuch, Matthias Hudler and Michael Krüger have been supported by the MA27 – EU-Strategie und Wirtschaftsentwicklung – in the course of the funding programme "Stiftungsprofessuren und Kompetenzteams für die Wiener Fachhochschul-Ausbildungen."

References

1. Apostolopoulos, G., Peris, V.G., Pradhan, P., Saha, D.: Securing electronic commerce: Reducing the SSL overhead. IEEE Network 14(4), 8–16 (2000)
2. Argyroudis, P.G., Verma, R., Tewari, H., O'Mahony, D.E.: Performance analysis of cryptographic protocols on handheld devices. In: Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA 2004), pp. 169–174. IEEE Computer Society Press, Los Alamitos (2004)

3. Berbecaru, D.G.: On measuring SSL-based secure data transfer with handheld devices. In: Proceedings of 2nd IEEE International Symposium on Wireless Communication Systems (ISWCS 2005), pp. 409–413. IEEE, Los Alamitos (2005)
4. Blake, I.F., Seroussi, G., Smart, N.P.: Advances in Elliptic Curve Cryptography. Cambridge University Press, Cambridge (2005)
5. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Möller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). Internet Engineering Task Force, Network Working Group, RFC 4492 (May 2006)
6. Brown, M.K., Cheung, D.C., Hankerson, D.R., López Hernández, J.C., Kirkup, M.G., Menezes, A.J.: PGP in constrained wireless devices. In: Proceedings of the 9th USENIX Security Symposium (SECURITY 2000), pp. 247–261. USENIX Association (2000)
7. Coarfa, C., Druschel, P., Wallach, D.S.: Performance analysis of TLS Web servers. ACM Transactions on Computer Systems 24(1), 39–69 (2006)
8. Dierks, T., Rescorla, E.K.: The Transport Layer Security (TLS) Protocol Version 1.1. Internet Engineering Task Force, Network Working Group, RFC 4346 (2006)
9. Freier, A.O., Karlton, P., Kocher, P.C.: The SSL Protocol Version 3.0. Internet Draft (November 1996), `http://wp.netscape.com/eng/ssl3/draft302.txt`
10. Gaisler, J.: The LEON-2 Processor User's Manual (Version 1.0.10) (January 2003), `http://www.gaisler.com/doc/leon2-1.0.10.pdf`
11. Großschädl, J., Kamendje, G.-A.: Architectural enhancements for Montgomery multiplication on embedded RISC processors. In: Zhou, J., Yung, M., Han, Y. (eds.) ACNS 2003. LNCS, vol. 2846, pp. 418–434. Springer, Heidelberg (2003)
12. Großschädl, J., Savaş, E.: Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 133–147. Springer, Heidelberg (2004)
13. Großschädl, J., Tillich, S., Szekely, A., Wurm, M.: Cryptography instruction set extensions to the SPARC V8 architecture (submitted for publication) (2007)
14. Gupta, V., Gupta, S.: Experiments in wireless internet security. In: Proceedings of the 3rd IEEE Conference on Wireless Communications and Networking (WCNC 2002), vol. 2, pp. 860–864. IEEE, Los Alamitos (2002)
15. Gupta, V., Gupta, S., Chang Shantz, S., Stebila, D.: Performance analysis of elliptic curve cryptography for SSL. In: Proceedings of the 3rd ACM Workshop on Wireless Security (WiSe 2002), pp. 87–94. ACM Press, New York (2002)
16. Gupta, V., Stebila, D., Fung, S., Chang Shantz, S., Gura, N., Eberle, H.: Speeding up secure Web transactions using elliptic curve cryptography. In: Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS 2004), pp. 231–239 (2004)
17. Gura, N., Chang Shantz, S., Eberle, H., Gupta, S., Gupta, V., Finchelstein, D., Goupy, E., Stebila, D.: An end-to-end systems approach to elliptic curve cryptography. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 349–365. Springer, Heidelberg (2003)
18. Gutmann, P.: Performance characteristics of application-level security protocols (2005), `http://www.cs.auckland.ac.nz/~pgut001/pubs/app_sec.pdf`
19. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer, Heidelberg (2004)
20. Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography (August 2000)
21. Karatsuba, A.A., Ofman, Y.P.: Multiplication of multidigit numbers on automata. Soviet Physics - Doklady 7(7), 595–596 (1963)

22. Koç, Ç.K., Acar, T., Kaliski, B.S.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro 16(3), 26–33 (1996)
23. Koschuch, M., Großschädl, J., Payer, U., Hudler, M., Krüger, M.: Workload characterization of a lightweight SSL implementation resistant to side-channel attacks. In: Franklin, M.K., Hui, L.C.K., Wong, D.S. (eds.) CANS 2008. LNCS, vol. 5339, pp. 349–365. Springer, Heidelberg (2008)
24. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
25. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
26. National Institute of Standards and Technology (NIST). Recommendation for Key Management – Part 1: General (Revised). Special Publication 800-57 (March 2007), `http://csrc.nist.gov/publications/PubsSPs.html`
27. National Security Agency (NSA). NSA Suite B Cryptography. Fact sheet (March 2008), `http://www.nsa.gov/ia/programs/suiteb_cryptography/`
28. OpenSSL Project. OpenSSL 0.9.7k. (September 2006), `http://www.openssl.org`
29. PeerSec Networks, Inc. MatrixSSL 1.7.1. (2005), `http://www.matrixssl.org`
30. Potlapally, N.R., Ravi, S., Raghunathan, A., Jha, N.K.: A study of the energy consumption characteristics of cryptographic algorithms and security protocols. IEEE Transactions on Mobile Computing 5(2), 128–143 (2006)
31. Potlapally, N.R., Ravi, S., Raghunathan, A., Lakshminarayana, G.: Optimizing public-key encryption for wireless clients. In: Proceedings of the 37th IEEE International Conference on Communications (ICC 2002), vol. 2, pp. 1050–1056. IEEE, Los Alamitos (2002)
32. Ravi, S., Raghunathan, A., Potlapally, N.R.: Securing wireless data: System architecture challenges. In: Proceedings of the 15th International Symposium on System Synthesis (ISSS 2002), pp. 195–200. ACM Press, New York (2002)
33. Rescorla, E.K.: SSL and TLS: Designing and Building Secure Systems. Addison-Wesley, Reading (2000)
34. Standards for Efficient Cryptography Group (SECG). SEC 1: Elliptic Curve Cryptography (2000), `http://www.secg.org/download/aid-385/sec1_final.pdf`
35. Wolf, W.H.: Hardware-software co-design of embedded systems. Proceedings of the IEEE 28(7), 967–989 (1994)
36. Zhao, L., Iyer, R., Makineni, S., Bhuyan, L.: Anatomy and performance of SSL processing. In: Proceedings of the 5th International Symposium on Performance Analysis of Systems and Software (ISPASS 2005), pp. 197–206. IEEE Computer Society Press, Los Alamitos (2005)