Hardware Acceleration of the SRP Authentication Protocol

Peter Groen[1,2], Panu Hämäläinen[2], Ben Juurlink[1], Timo Hämäläinen[2]

[1]Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628 CD Delft, the Netherlands
Phone: +31-15-2786196 Fax: +31-15-2784898
P.T.Groen@ewi.tudelft.nl

[2]Institute of Digital and Computer Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
Phone: +358-3-3653366 Fax: +358-3-3653095

Abstract— **This work focuses on the design of the hardware acceleration of a WLAN authentication proceduce. The acceleration is used in the Secure Remote Password protocol. The experimental platform is TUTWLAN, a WLAN platform being developed at Tampere University of Technology which runs on the Altera Excalibur development board that contains a chip with programmable hardware and a microprocessor. In this work this function is implemented in the programmable logic and used in the authentication protocol running on the microprocessor. In addition, proposals for further hardware speed-up of the exponentiation design are presented. The results show that a full modular exponentiation with inputs of 1023 bits can be performed in less than 40 ms using less than 10.000 logic elements consisting of 4-input lookup tables and registers. By using the implemented hardware accelerator in the authentication protocol, the execution time is reduced by a factor of 4. An additional speed-up factor of 5 (totaling a factor of 20) is possible by implementing the fastest design discussed in this work.**

Keywords— **Modular exponentiation, hardware acceleration, reconfigurable hardware, WLAN, authentication.**

I. Introduction

In recent years wireless Local Area Networks (WLANs) have gained popularity. They can be used to connect computers when building a wired network is impossible or inconvenient, e.g. when users are moving or when the network is set up only temporary. Because data is transferred through the ether, security is needed not only to prevent that data is intercepted by unauthorized users but also to authorize users and computers to the network. One way to obtain security is by using cryptographic operations. These operations, however, often require time consuming computations that slow down network speed and increase log-on time. Furthermore, support for several algorithms and possibilities to update the implementation may be required. Trading chip area and power-consumption for execution speed depending on the application requirements may also be profitable. High speed with flexibility can be obtained by using reconfigurable hardware for the time-consuming operations and a microprocessor for the remaining parts of the security implementation.

The objective of this work is to accelerate the authentication protocol of a WLAN, called TUTWLAN, developed at Tampere University of Technology. The protocol employed is the Secure Remote Password (SRP) authentication protocol [15]. It makes extensive use of hash and modular exponentiation functions. Both of them are often used in security protocols and require large amount of arithmetic operations. The aim is to find or design appropriate hardware accelerators that can be called from software routines in the SRP protocol.

This paper is structured as follows. Section II describes the SRP protocol. Section III explains the time-consuming functions in the protocol. The implemented hardware accelerator is described in Section IV. Section V presents the results and Section VI discusses how execution time can be further reduced. Conclusions can be found in VII and possibilities for future work in Section VIII.

II. The SRP Protocol

The Secure Remote Password (SRP) protocol [15] is an authentication and key-exchange protocol suitable for secure password verification and session key generation over an insecure communication channel such as in WLANs. In this section the protocol is described briefly.

The SRP protocol employs a method to exchange session keys called Asymmetric Key Exchange (AKE). In this method verifiers are stored instead of the passwords. The verifier is computed from the password using a one-way mathematical function [11]. The password and verifier correspond to private and public keys with the difference that the verifier is kept secret on the server instead of being publicly known. A stolen verifier is not sufficient to log-on because the password is still needed. To establish a secure connection it is sufficient when one side (server) stores the verifier(s) while the other (client) computes the verifier from a user given password.

The initial setup of SRP goes as follows. First, the user enters a password. Then a verifier is computed from the

TABLE I
USED SRP PARAMETER SIZES IN BITS.

Parameter	N = 511	N = 1023	N = 2047
user name, u	32	32	32
password (x)	64 (160)	64 (160)	64 (160)
salt s	80	80	80
generator g	8	8	8
verifier v	512	1024	2048
random a, b	256	256	256
public A, B	512	1024	2048
key K, M1, M2	320	320	320

password and a randomly generated password salt. Next the user name, salt and verifier are stored in the database on the server. Now the client is ready to authenticate to the server. The steps performed by the SRP protocol are depicted in Figure 1. All computations are performed modulo N. The modulus N is a large prime number with a length of hundreds of bits. SRP parameter sizes as used in this work are listed in Table I [6]. The authentication is successful if M_1 computed in Step 7 and M_2 in Step 8 are identical on the client as well as the server side.

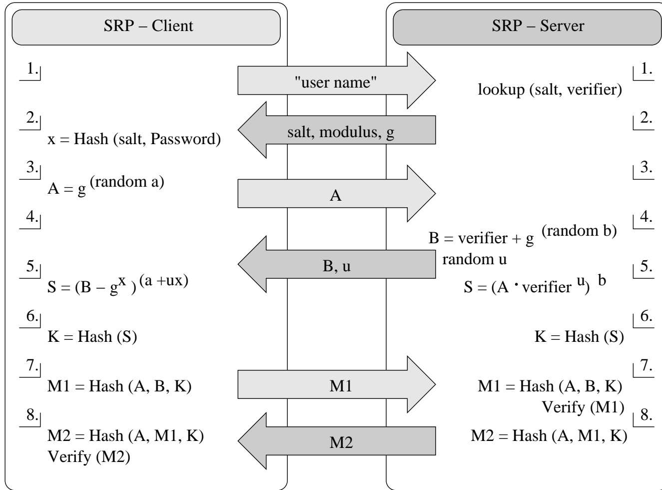

Fig. 1. Steps in the SRP protocol.

In some of the steps, a one-way hash function is employed. The Secure Hash Algorithm (SHA-1) [12] is used for this. It produces a 160-bit hash value from a string of variable length. In addition, the protocol uses an interleaved hash that computes 2 hash functions and yields 320 bits. The first hash function is applied to the even bytes and the second to the odd bytes of the input. The hash outputs are byte-wise interleaved to get the final result.

The SRP protocol can be implemented in C using a library that allows large variables to be processed and contains optimized C/C++/Assembly routines. The MIRACL library developed by Shamus Software Ltd [13] is very suitable since it supports the ARM9 processor that is used in the TUTWLAN platform. The software performance of the SRP protocol has been evaluated for the ARM9 in [6].

III. FAST MODULAR EXPONENTIATION

In the SRP protocol, modular exponentiations on large values are performed. This function requires many operations and consumes a large fraction of the total execution time of a software implementation of the SRP protocol [6]. This work, therefore, focuses on designing appropriate hardware accelerators for modular exponentiations.

An efficient way to compute exponentiations is by using the Square and Multiply algorithm [7]. Algorithm 1 describes the right-to-left Square and Multiply algorithm: bits are processed from least significant (rightmost) to most significant (leftmost). The right-to-left method allows for parallel squaring and multiplying.

Algorithm 1: Right-to-left Square and Multiply [7].
$P = X^E \bmod N, E = \sum_{i=0}^{n-1} e_i 2^i, e_i = \{0, 1\}$.

1. $Z_{[0]} = X,\ P_{[0]} = 1$
2. *FOR* $i = 0$ *to* $n - 1$ *DO*
3. $\quad Z_{[i+1]} = Z_{[i]}^2 \bmod N\ (square)$
4. $\quad$ *IF* $e_i = 1$
   THEN $P_{[i+1]} = Z_{[i]} \cdot P_{[i]} \bmod N\ (multiply)$
   ELSE $P_{[i+1]} = P_{[i]}$
5. *END FOR*

Modular exponentiations require a division after each multiplication. Montgomery [9] has invented a method to eliminate the expensive divisions. The operands are transformed into N-residue before starting to perform multiplications. The computations are performed on the residues and afterwards the residue is transformed back to normal representation. An N-residue of x is defined as $x \cdot R \bmod N$ where $R = 2^n$ and n is the number of bits in modulus N, such that $R > N$. The key idea is to perform multiplications modulo R instead of modulo N which replaces the expensive division after each multiplication by divisions by a power of 2 (i.e. a rightshift). Montgomery has defined Algorithm 2 to compute $T \cdot R^{-1} \bmod N$ from T for $0 \leq T < RN$.

Algorithm 2: Montgomery reduction $TR^{-1} \bmod N$ [9].
$0 \leq m < R, 0 \leq T < RN, N' = -N^{-1}$
function $REDC(T)$

1. $m = (T \bmod R)N' \bmod R$
2. $t = (T + mN)/R$
3. *IF* $t \geq N$ *THEN* $t - N$
   ELSE t

The radix R and modulus N need to be co-prime (no commom divider greater than one) to guarantee the existence of an inverse of N. Thus the algorithm works for any odd modulus, which is the case in SRP. Function $REDC(T)$ forms the basic framework to perform multiplication with N-residues. The output of this algorithm is the input times R^{-1}. A product of residues of x and y is computed as:

$$z = REDC((xR \bmod N)(yR \bmod N)) \quad (1)$$
$$= (xy)R^2R^{-1} \bmod N \quad (2)$$
$$= xyR \bmod N \quad (3)$$

Transformation to and from Montgomery-residues can be done by using *REDC*. The initial values x and y have to be multiplied with $R^2 \bmod N$. Transformation from residues to normal representation can be done by multiplication with 1. Now only $R^2 \bmod N$ has to be computed using slow non-Montgomery multiplications. This has to be performed only once for a given modulus.

A multiplication is composed out of the addition of products. Each product is the result of a part of the multiplier multiplied with the full multiplicand. If l is the number of bits of the multiplier, a full multiplication would require l times a multiplication of 1 bit of the multiplier with the full multiplicand. If k bits of the multiplier are processed (radix-2^k), a full multiplication would require l/k multiplications. Algorithm 2 can be transformed to Algorithm 3 to realize a radix-2^k multiplier that is called l/k times [5] [4].

Algorithm 3: Radix-2^k Montgomery modular multiplication [4]. $m = l/k$; $A, B < N$; $R = 2^{mk}$; $N' = -N^{-1} \bmod 2^k$; $gcd(2^k, N) = 1$,
Modulus $N = \sum_{i=0}^{l-1} m_i(2^k)^i$,
Multiplier $A = \sum_{i=0}^{l-1} a_i(2^k)^i$,
Multiplicand $B = \sum_{i=0}^{l-1} b_i(2^k)^i$,
Result $S = \sum_{i=0}^{l} s_i(2^k)^i$,
$m_i,\ s_i,\ b_i,\ a_i = \left\{0, 1, .., 2^{k-1}\right\}$.

1. $S_0 = 0$
2. *FOR* $i = 0$ *to* $m - 1$ *DO*
3. $\quad q_i = (((S_{[i]} + a_iB) \bmod 2^k)N') \bmod 2^k$
4. $\quad S_{[i+1]} = (S_{[i]} + q_iN + a_iB)/2^k$
5. *END FOR*
6. *IF* $S_{[m]} \geq N$ *THEN* $S_{[m]} - N$
   ELSE $S_{[m]}$

The output value of Algorithm 3 is $ABR^{-1} \bmod N$. The algorithm is suitable for implementation on reconfigurable hardware because large multipliers and divisions are avoided. The Montgomery algorithm allows a pipelined implementation because the next computation does not have to wait for the most significant bits in the determination of q_i. This makes Montgomery suitable for processing multiplications with very large moduli.

Algorithm 3 can be further improved by replacing the *IF* statement in Step 6 by two extra iterations. Two extra division will make the result always smaller than 2 times the modulus. The addition in Step 3 is avoided by multiplying B by 2^k. The product $a_iB \bmod 2^k$ will thus be 0 for all B. In Step 4, B can be taken out of the division. The modulus N must be odd to be co-prime to the Montgomery radix R, the inverse of N in Algorithm 3 for the radix-2 algorithm becomes 1 since $N' = -N^{-1} \bmod 2 = 1$ for every odd N. No inverse needs to be computed. Algorithm 4 shows the algorithm that is implemented in the hardware [8][14].

Algorithm 4: Radix-2 Montgomery modular multiplication [3]. $m = l$

1. $S_0 = 0$
2. *FOR* $i = 0$ *to* $m + 2$ *DO*
3. $\quad q_i = S_{[i]} \bmod 2$
4. $\quad S_{[i+1]} = (S_{[i]} + q_iN)/2 + a_iB$
5. *END FOR*

The fourth step contains the arithmetic operations that can be implemented in an array of processing units in reconfigurable hardware. Two multipliers and two adders are needed in each unit to implement the multiplications and additition.

IV. Implementation of Radix-2 Modular Exponentiation

The hardware acceleration for the SRP protocol has been implemented on the Altera Excalibur device. The chip consists of a 32 bit RISC ARM922 processor operating at up to 200 MHz combined with a programmable logic device (PLD). Processor and PLD are connected to each other by an embedded stripe. Communication between stripe and PLD goes through two dual ported RAM blocks (one for the smallest Excalibur device) or through AHB master/slave ports. The PLD can also interrupt the ARM directly through 6 interrupt lines.

The structure of the PLD in the Excalibur is equal to Altera's APEX20K devices at an internal voltage of 1.8. The APEX devices are based on a large number of Logic Elements (LEs). Each LE contains a 4-input look-up table (LUT) and one register which means that any boolean function of up to 4 bits can be implemented in one LE. It can also be configured to operate in a special arithmetic mode which has only two inputs but allows for fast propagation of carries between neighboring LEs. A register is available directly after each LUT. The output of the LUT can be routed through the register, around the register or

TABLE II

EXCALIBUR DEVICES.

Device	EPXA1	EPXA4	EPXA10
Single-port RAM	32kbytes	128kbytes	256kbytes
Dual-port RAM	16kbytes	64bytes	128bytes
Typical gates	100.000	400.000	1.000.000
Logic Elements	4.160	16.640	38.400
ESB/RAM blocks	26	104	160
Max system gates	263.200	1.052.000	1.772.000
Max user pins	246	488	711

both. In addition, the PLD has ESB/RAM blocks which can store up to 2048 bits each. Refer to Table II and [1] for specifications of the Excalibur Devices.

A. Exponentiation Unit

The computations are performed in an array of processing units as depicted in Figure 2. The multiplicand B is loaded into the units from a central point, all other signals are moved between neighboring units. Algorithm 4 defines that all intermediate values are at most two times as large as the modulus. This means that, internally everything should be 1025 bits to allow a 1024-bit modulus. To simplify the design, the maximum modulus is set on one bit less: $2^n - 1$. Moduli will have lengths of for example 511, 1023 or 2047 bits. This implies that the length of all intermediate values are a power of two which simplifies the structure of the design.

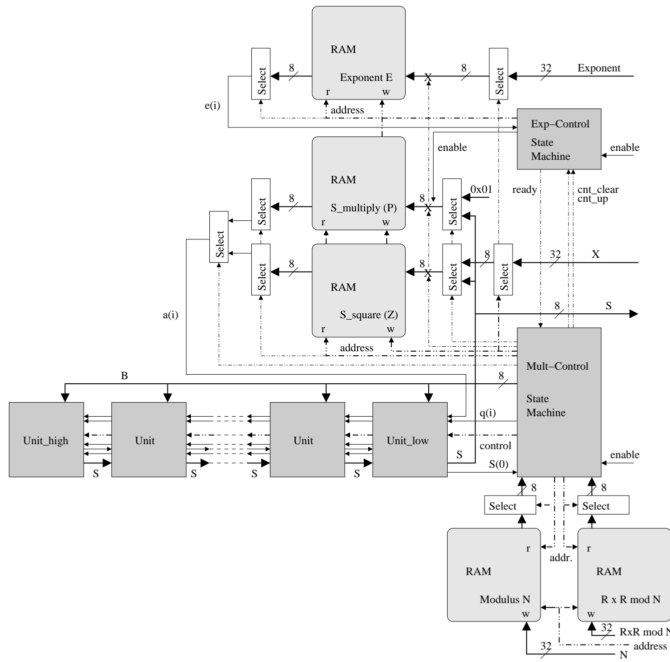

Fig. 2. Systolic radix-2 exponentiation in PLD.

Figure 2 shows five RAM blocks. The N and R block store the modulus N and the Montgomery radix $R^2 \bmod N$. The Montgomery radix is used to transform variables into residues. The upper block stores the exponent E and the two blocks in the middle of the figure store the results of each squaring and multiplication. Two control machines are included in the design. The upper machine (Exp-control) contains a counter to generate the address to select the appropriate bit of the Exponent E. The lower machine (Mult-control) controls the systolic array.

The procedure to compute $X^E \bmod N$ using Montgomery is as follows: X is transformed to Montgomery-residue. This takes one Montgomery multiplication. Then the Square and Multipy algorithm is applied. The final output P is transformed back to normal representation which takes another Montgomery multiplication. In the radix-2 algorithm, only one bit of the multiplier is processed per multiplication thus one iteration of the Multiply and Square algorithm takes $(l + 3) \times 2$ cycles. All iterations together take $n \times (l + 3) \times 2$ cycles. A full exponentiation (exponent and modulus have the same length), including transformations, takes $(n + 2) \times (n + 3) \times 2$ cycles, where n is the bit-length of the modulus.

B. Processing Units

The number of bits per unit is an important factor of the maximum frequency (fmax) since it directly affects the routing complexity and longest path. Eight registers are needed per unit to store signals that are transported between units. Processing more bits of the multiplicand per unit saves registers but increases the complexity and longest path. In this design, each unit processes eight bits of the multiplicand.

Figure 3 shows the structure of one processing unit. It computes $S_{i+1} = (S_i + q_i N)/2 + a_i B$. *Adder_1* computes $(S_i + q_i N)$ and *Adder_2* adds $a_i B$. The block labeled *Box* computes the 9th bit of the addition of S and $q_i N$. This bit is needed as extra input for the second adder because of the rightshift between the adders. For the highest unit, the carry-out bit of *Adder_2* comes back as $S(i)[0]$. *Adder_2* needs to be one bit larger and the box will contain a 2-bit adder instead of 1-bit. A unit has to perform the following steps to perform a Montgomery multiplication.

1. The modulus N (8 bits per unit) is loaded via B in Multiplexer+Reg_1. Multiplier_1 is set on multiplication by 1 to let the modulus pass through unchanged while the registers in Reg_1 are reading the data.
2. The multiplicand B (8 bit per unit) of Algorithm 4 is loaded via B in Multiplexer+Reg_1. This operand is the same for both squaring and multiplication step of the exponentiation Algorithm 1.

3. The multipliers a_i are moved through the units at one bit per clock (radix-2).
4. The result of the first multiplication (squaring) are fed back to Multiplexer+Reg_2 for the next iteration.
5. After the completion of the multiplications, both results are moved to Multiplexer+Reg_3. The result of the first multiplication (squaring) is sent to the previous unit and also to Multiplexer+Reg_1. The result of the second multiplication is moved to the previous unit only.
6. Step 2 to 5 are repeated to compute an exponentiation. In Step 2, S is loaded instead of B.

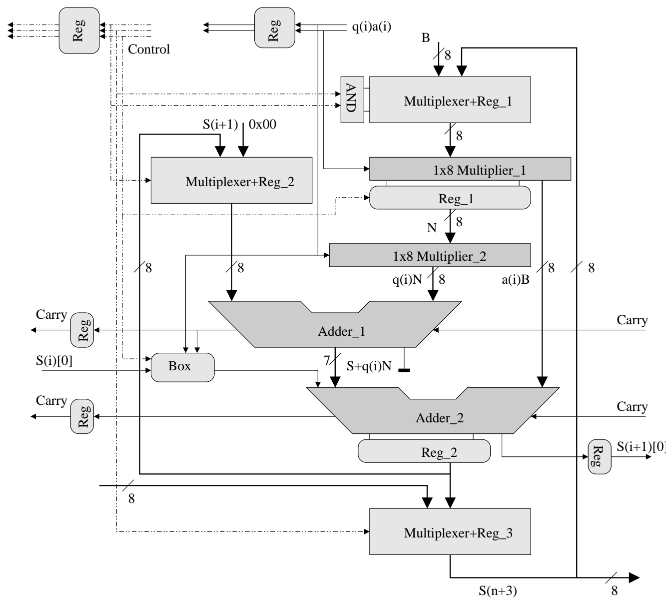

Fig. 3. 8-bit radix-2 processing unit.

All signals are registered before they leave a unit. Thus a signal needs one clock cycle to move to the higher neighbor. The higher neighbor uses these signals and shifts its lowest bit back to the lower unit afterwards. This allows for a two-cycle operating mode with interleaved squaring and multiplying. The unit could be transformed to a single-cycle version by not registering the bit of S that is transported to the lower unit. In a single-cycle version, the square and multiply steps in the exponentiation algorithm are performed in series. No long carry chains through the whole array are created as long as the adder is at least twice as large as the number of bits of S that are moved to the lower unit. For an 8-bit wide unit, up to 4 bits or radix-16 can be computed in one-cycle mode.

A unit needs 70 LEs when implemented in the Altera APEX PLD. The registers after the adder are combined in one LE. The registers that store the modulus N are placed after the $1{\times}8$ Multiplier so that they fit in the same LE.

C. Communication with the ARM Processor

Communication between the PLD and the ARM processor can be done through DPRAM or AHB slave/master ports. The DPRAM ports allow for quick access to large amount of data. Since the exponentiation function needs large numbers before computation can start and the result has to be written back afterwards, the DPRAM ports are most suited. Both DRPAM ports are configured for 32-bit data size so reading of an exponent and base of 1023 bits takes 32 clock cycles. About 400 LEs are needed for the communication with the ARM, mainly caused by registers and control that is needed to move the data to and from the exponentiation unit. The hardware modular exponentiation can be enabled/disabled by sending a signal to an AHB-slave in the PLD. This allows other parts of the TUTWLAN to use DPRAM when no exponentiations are computed.

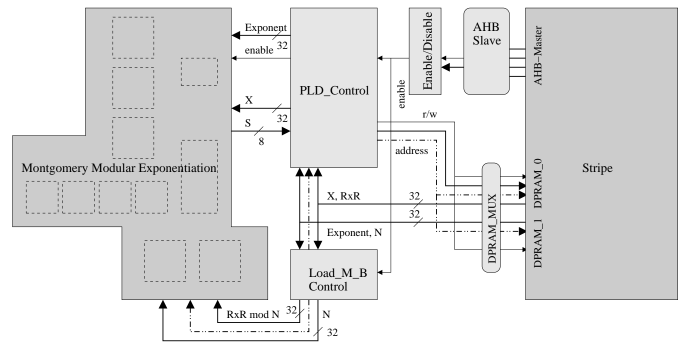

Fig. 4. Communication between ARM and Exponentiation.

Four parameters are needed to perform modular exponentiation. The modulus N, the Montgomery radix R^2, a base, and an exponent. The read and write process is managed by a state machine (PLD_control in Figure 4). Base and exponent are stored in ESB/RAM blocks, and the addresses are generated by the exponentiation control unit. Reading of N and R^2 is controlled by a separate control unit to generate the address. This is because they are stored in ESB/RAM blocks before the state machine in the modular exponentiation hardware is enabled. They have to be read only when a new modulus is used.

D. Software Routines

The hardware exponentiation is controlled by the software running on the ARM. The software consists of three routines:
1. Read N, compute $R^2 \bmod N$ and store them in DPRAM.
2. Read Base and Exponent, move them to DPRAM, start accelerator.
3. Read results from DPRAM.

The radius R needs to be transferred to Montgomery domain ($R^2 \mod N$) by traditional divisions. Although this could be performed in hardware it is performed in software, since it is likely that the modulus is not frequently changed in SRP. Changing the modulus requires computing new verifiers for each client and updating the whole server database (it is assumed that the whole database uses the same modulus).

The SRP implementation processes variables in a special large format supplied by the MIRACL library. Before moving variables to DPRAM, they are transformed to strings, then to a series of 32-bit integers and stored in DPRAM. The results are transformed from integers to string and then back to the MIRACL big format. The UART port is configured to output to a terminal on a regular PC. Data can be input and output through this terminal.

V. Experimental Results

During the experiments, the ARM processor was configured to run at a frequency of 50 MHz and the PLD at 33 MHz. The Quartus software was used together with the ARM Developer Suite [1][2] to compile and simulate the designs. The hardware part of the design is implemented in VHDL and Altera standard components from the Quartus II software. Leonardo Spectrum was used for synthesis. The Quartus software produces a .hex file that was downloaded into the flash memory of the Excalibur development platform. The design has been tested by comparing the results of the exponentiation in hardware with the results in software. The compiler returns values for the maximum frequency and the number of LEs needed in its compilation report. These values are depicted in Table III.

TABLE III
Max. frequency and LE usage radix-2.

Modulus	LEs	RAM-bits	fmax (MHz)
511 bit	5149	3584	65.37
1023 bit	9644	5120	65.11
2047 bit	18186	10240	58.01

The clock cycles in the PLD have been measured with a counter in the PLD that starts when the software enables the hardware exponentiation and stops immediately after the results have been written to the DPRAM. Table IV shows how many clock cycles have been measured by the counter and how many full exponentiations are possible per second.

The operations g^x, g^a, g^b and the exponentiation for the session-key S are computed in hardware, where the length of the exponent is 256 bits. The server computes v^u in

TABLE IV
Cycles for a full radix-2 exponentiation.

Modulus	clock cycles/exp.	ms./exp.	exp./sec.
511 bit	530,704	16	61.24
1023 bit	2,109,968	32	30.86
2047 bit	8,414,224	73	13.79

Step 5 of the SRP protocol, where u is 32 bits width, in software. The execution time of this step is significant but performing it in hardware would be even slower because the hardware always runs through the maximum length of the exponent. Also in g^x the exponent is smaller (160 bits). The efficiency of the hardware accelerator would be a little higher when the hardware is adapted to stop after all bits of the exponent have been processed.

The ARM9 can run at maximum at 200 MHz and the PLD up to the maximum frequency depicted in Table III. For comparison, the ARM9 is set on 200 MHz and the PLD on 50 MHz. The difference in execution time as a result of the hardware acceleration is shown in Figure 5. The speed-up increases with the size of the modulus. For a modulus of 1023 bits, the hardware implementation of the exponentiation is about four times faster than the software implementation. Since exponentiation takes most of the execution time of the SRP a speed-up of this function will result in a similar speed-up of the whole protocol.

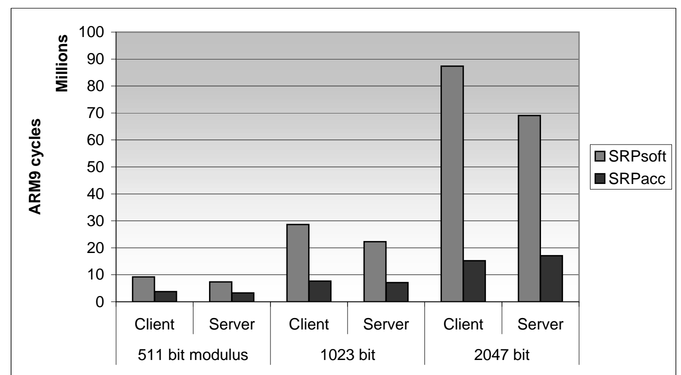

Fig. 5. SRP with (SRPacc) and without HW (SRPsoft).

VI. Improvements

The radix-2 array architecture performs multiplications at one bit of the multiplier per cycle. The design needs at minimum $2 \times n \times n$ clock cycles to complete a full modular exponentiation with Montgomery residues, where both X and E in X^E have a bit-length of n. The execution time can be further improved in the following ways:

1. Compute more than 1 bit per Montgomery multiplication (higher radix).

2. Compute more than 1 bit per iteration of the exponentiation (M-ary).
3. Compute multiplications and squaring in parallel.

A. Higher Radix

In higher radix, several bits of A are inserted into the units instead of only one. Contrary to the radix-2 design, the inverse modulus is needed as in Algorithm 3. The inverse modulus N' can be easily computed [10]. The time to compute the inverse is negligible since the inverse is small compared to the modulus. The higher radix Montgomery multiplication algorithm can be derived from Algorithm 3 in the same way as was done for the radix-2 design [4]. Radix 2^k takes $2 \times n \times n/k$ cycles.

Most of the implemented radix-2 design can also be used for a higher-radix version. To omit the need for multipliers, all possible outputs of the products $a_i 2^k S$ and $q_i 2^k N$ in Algorithm 3 have to be pre-computed and are stored in ESB/RAM. For a 1023 bit exponentiation, 2048 bits have to be read from RAM per clock cycle, which requires 128 ESB blocks. A block stores a 16 bit part of either $a_i 2^k S$ or $q_i 2^k N$. The outputs of $a_i 2^k S$ have to be pre-computed for each iteration of Algorithm 1. One 16-bit multiplexor and a 16-bit adder are needed to perform these pre-computations. Addressing for the RAM blocks takes 2 LEs per bit of a_i and q_i per processing unit. Together, approximately 50 extra LEs are needed per 16 bits of the modulus. For a modulus of 1023 bits and radix-16 this will be about 3200 extra LEs. The method becomes less efficient for radices larger than 16 because of the large amount of multiples that need to be pre-computed.

B. M-ary

The M-ary method allows processesing of several bits of the exponent per multiplication in the Square and Multiply Algorithm [7]. Processing 2 instead of 1 bit per multiply of the exponent reduces the number of clock cycles with almost 25%. Only one multiply operation is needed for every two square operations. The M-ary method requires parameter P in Algorithm 1 to be stored several times. If, for example, 2 bits of the exponent are processed per multiplication, three Ps have to be stored. The first P is modified if the bits are '01', the second if the bits are '10' and the third if the bits are '11'. In the end $P2 = P2 \cdot P3$ and $P1 = P1 \cdot P2$ and $P = P3 \cdot P2 \cdot P1$ are computed to get the final result. The M-ary method can be used on any number of bits but at a certain point so many post multiplications are needed that the benefit disappears. The clock cycles can be roughly computed as $n+n/(\mathit{number_of_bits})$ where n is the bit-length of the modulus.

Implementation of the M-ary method requires an extra ESB/RAM block for each P and LEs to build multiplexors for selection of the appropriate P. Since at most 8 bits are selected at each clock cycle, the cost of the multiplexor is small: a 2-ary with 3 multiples requires 16 LES, a 4-ary with 15 multiples requires 80 LEs to select the appropriate P. Furthermore, the control unit may increases in size, especially due to at least two extra states to cover the extra multiplications.

C. Parallel Arrays

Two parallel arrays for squaring and multiplying generally require twice as much area at a speed gain of a factor 2. Parallel arrays require the processing units to be single cycle. This is possible up to radix-16 when 4 out of 8 result bits are connected to the lower neighboring unit without being registered. Another modification is that instead of shifting between the adders in a processing unit, the entire inputs are shifted. An extra benefit of parallel squaring and multiplying is that squaring can be done in higher radix, while the M-ary method can reduce the number of multiplications simultaneously.

D. Execution Time Estimates

The number of clock cycles required by the methods sketched above can be computed by combining the formulas for M-ary, higher radix and parallel. Cycles have been estimated for radix-4, 16, 64, radix-64 and 256 with only the odd half of the possible products is stored in RAM and parallel arrays for radix-2, 4 and 16. The results are shown in Figure 6. Pre and post-computations are included in the execution times. Since the longest path within the units does not increase significantly for higher radix and not at all for M-ary, it is assumed that a clock frequency of 50 MHz is possible (25% smaller than the maximum frequency of the radix-2 design). The figure shows that the improvement for radices beyond 16 is small. Also the effect of the M-ary method decreases for higher radices due to larger number of cycles that is needed for pre/post-multiplications.

All designs fit on the EPXA10, the largest Excalibur device. Expansion to 2047-bit data-width is only possible for radix-2, 1-ary, 2-ary and 4-ary designs and will take approximately 4 times as many cycles as the radix-2. The fastest design is the radix-16, 4-ary parallel which is estimated to require less than 290.000 clock cycles at an area consumption of about 25,000 LEs and nearly all ESB/RAM blocks in the PLD.

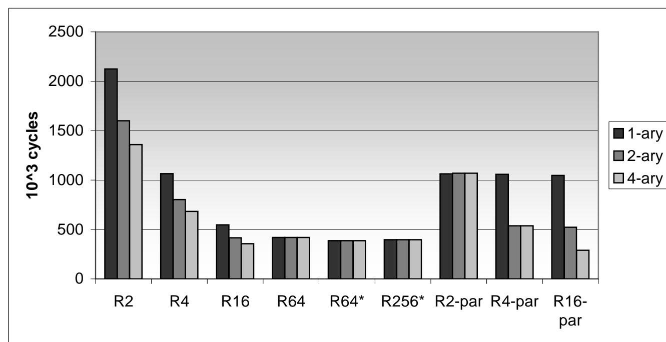

Fig. 6. Clock cycles for speed up methods (1023 bits modulus).

VII. Conclusions

The hardware accelerated SRP protocol is about 4 times faster for a 1023-bit modulus than a software implementation. For larger moduli the gain increases. A multiplier consisting of a radix-2 systolic array was designed in the PLD. It can be configured to perform modular exponentiations using moduli of 127, 255, 511, 1023, or 2047 bits. Where the modulus is 1023 bits uses less than 10,000 LEs or about 1/4th of the PLD of the largest Excalibur chip is needed. A full modular exponentiation with inputs of 1023 bits is performed in 2.1M clock cycles with a maximum frequency of about 65 MHz. The hardware exponentiation can be used continuously. The number of full exponentiations that can be computed per second is 30 for a 1023 bit modulus and 60 for a 511-bit modulus.

Faster hardware exponentiations are possible by using higher radix, M-ary and parallel multipliers. The fastest design would be one that uses radix-16, 4-ary and has 2 parallel arrays which would run in less than 300,000 cycles. The maximum frequency will be lower due to a higher complexity. Nevertheless, if it would drop to 40 MHz, it would still be 5 times faster than the radix-2 design. This design can make the hardware multiplier 20 times faster than the software in the ARM9 which runs at a maximum frequency of 200 MHz.

VIII. Future Work

The faster designs can be implemented for the Altera PLD in future work. An implementation of the radix-16, 4-ary, 2 parallel modular exponentiation might be one of the fastest designs for a modular exponentiation in reconfigurable hardware. Other possible improvements are:

- Modify the design such that it can handle modulus sizes other than powers of two minus one.
- The design can be optimized such that it can adapt to the size of the ground and exponent instead of always asuming the maximum size. This would make the hardware accelerator also useful in RSA public key encryption.
- Perform hashing in reconfigurable hardware. An SHA-1 hash function is available from Altera.
- The design presented here can be compiled for other, newer (Altera) PLDs. A PLD with a large number of multipliers allows for new, faster designs for modular exponentiations. Also more ESB/RAM blocks can be useful.
- The area consumption of the loading procedure can be reduced by about 75 % when only eight bits are loaded per clock instead of two times 32 bits.
- Taking a closer look to types other than systolic arrays with 8-bit units or modular multipliers.

References

[1] Altera Homepage. www.altera.com, 2003. Excalibur HW Reference Manual, APEX 20K Programmable Logic Device Family Data Sheet, Application Note 142 Using the Embedded Stripe Bridges.

[2] ARM Ltd. Homepage. www.arm.com, 2003.

[3] T. Blum and C. Paar. Montgomery Modular Exponentiation on Reconfi gurable Hardware. In *14th IEEE Symposium on Computer Arithmetic*, pages 70–77. IEEE, Apr 1999.

[4] T. Blum and C. Paar. High Radix Montgomery Modular Exponentiation on Reconfi gurable Hardware. In *IEEE Transactions on Computers*, volume 50(7). IEEE, Jul 2001.

[5] S.E. Eldridge and C.D. Walter. Hardware Implementation of Montgomery's Modular Multiplication Algorithm. In *IEEE Transactions on Computers*, pages 693–699. IEEE, Jul 1993. 42(6).

[6] P. Hämäläinen, M. Hännikäinen, M. Niemi, and T. Hämäläinen. Performance Evaluation of Secure Remote Password Protocol. In *IEEE International Symposium on Circuits and Systems*, volume 3, pages 29–32, Scottsdale, Arizona, USA, May 2002. IEEE.

[7] D.E. Knuth. *The art of computer programming: Seminumerical algorithms*, volume 2. Addison-Wesley, 2002.

[8] C.K. Koc, T. Acar, and B.S. Kaliski. Analyzing and Comparing Montgomery Multiplication Algorithms. In *IEEE Micro*, pages 29–33. IEEE, Jun 1996. (16)3.

[9] P.L. Montgomery. Modular Multiplication Without Trial Division. In *Mathematics of Computation*, pages 519–521, Apr 1985. 44(170).

[10] T. Ristimäki and J. Nurmi. Implementation of a Fast 1024-bit RSA Encryption on Platform FPGA. In *6th IEEE International Workshop on Design and Diagnostics of Electronics Circuits and Systems (DDECS'03)*, Poznan, Poland, Apr 2003.

[11] B. Schneier. *Applied Cryptography Second Edition*. John Wiley & Sons. Inc., 1996. ISBN 0471128457.

[12] The Secure Hash Standard. Federal information processing standards (fi ps) - publication 180-1, National Institute of Standards and Technology (NIST), USA, 1995.

[13] Shamus Software Ltd, Dublin, Ireland. *M.I.R.A.C.L. Users Manual*, Sep 2002.

[14] C.D. Walter. An Overview of Montgomery's Multiplication Technique: How to make it Smaller and Faster. In *Cryptographic Hardware and Embedded Systems (CHES 1999)*, Lecture Notes in Computer Science, Worcester, Massachusetts, USA, Aug 1999. Springer-Verlag Heidelberg.

[15] T. Wu. The Secure Remote Password protocol. In *Internet Society Network and Distributed Systems Security Symposium (NDSS)*, pages 97–111, San Diego, California, USA, Mar 1998.