PAX: A Datapath-Scalable Minimalist Cryptographic Processor For Mobile Environments

A. Murat Fiskiran Ruby B. Lee

Princeton Architecture Laboratory for Multimedia and Security (PALMS)

Department of Electrical Engineering

Princeton University

Abstract

We describe a datapath-scalable, minimalist cryptographic processor, called PAX, for mobile environments where the communication with the outside world is done on wireless connections. PAX is designed to fully utilize the high data rates of the newest and developing wireless technologies. Today, these rates exceed 2 Mbps for cellular/PCS connections, and 50 Mbps for WLAN connections. Future rates are expected to be about 20 Mbps and 100 Mbps for cellular/PCS and WLAN respectively.

In designing PAX, we first select a cipher suite that is suitable for mobile environments. This provides all basic security functions such as confidentiality, data integrity, user authentication, and digital signatures. We then define the PAX instruction set, which contains few new instructions that provide huge speedups for key sections of the algorithms in our cipher suite. We compute the processor speeds required for secure communications at data rates that can be supported by the newest and developing wireless technologies. For bulk encryption and hashing, a 7 MHz 32-bit single-issue PAX processor is sufficient to match the 2.4 Mbps data rate of future 3G cellular networks. To match the 54 Mbps data rate of the IEEE 802.11a/g WLAN connections, the clock rate needs to be 150 MHz. Both figures are significantly under the 400 MHz rate used by the processors in today's mobile information appliances such as PDAs.

Datapath scalability refers to the feature that the same instruction set can be implemented in processors with different word sizes. This feature, first introduced in the PLX multimedia instruction set, provides extra flexibility in balancing the performance and cost of a system. We test the usefulness of datapath scalability by varying the word size from 32 bits to 64 bits to 128 bits. For public-key cryptography and bulk encryption, datapath scalability provides 10× to 20× *additional* speedup.

1. Introduction

Security requirements of a mobile device that communicates wirelessly differ from a wired device in two important ways. First, a wireless channel is always public, which makes it inherently vulnerable to passive attacks such as eavesdropping. Second, the mobile device is likely to have far fewer computational resources when compared to a wired device (such as a desktop computer) to perform the compute-intensive cryptographic operations to attain a desired level of security. This imposes certain restrictions on the designers in their choice of hardware and cryptographic algorithms that can be used in the device to achieve sufficient security at a low enough cost.

In this paper, we describe a datapath-scalable, minimalist cryptographic processor, called PAX, that can support all basic security functions at high-enough throughputs so that it can meet the link speeds, or available bandwidths, offered by the newest and developing wireless technologies. *Datapath scalability* refers to the property that the same instruction set can be implemented in processors with different word sizes. This feature was first introduced in PLX, which is a minimalist, high-performance multimedia instruction set [1-3]. Datapath scalability provides extra flexibility to a designer in balancing the cost and performance of a system.

In designing PAX, we first select a cipher suite that supports the four basic security functions that must be implemented by any cryptographic processor: user authentication, confidentiality, integrity, and digital signature. Second, we define a concise and powerful instruction set for this processor so that it can perform these security functions at very high data rates. PAX is designed to be a concise and efficient processor for cryptography, so that it can be used either as an embedded processor, a cryptographic co-processor alongside a general-purpose processor, or a security module in a system-on-chip.

The rest of the paper is organized as follows. In Section 2, we describe the major wireless technologies and compare their data rates. In Section 3, we describe the cryptographic algorithms in our cipher suite. In Section 4, we describe the PAX architecture. In Section 5, we present our performance results. Section 6 is the conclusion.

2. Major wireless technologies

Wireless technologies are broadly classified into two major groups: cellular/PCS (Personal Communication Service) and WLAN (Wireless Local Area Network) [4].

Cellular/PCS uses the FCC-regulated 800 MHz and 1900 MHz frequency bands [4, 5]. Signals are transmitted at high power, which provides a long range to devices that use these technologies. Cellular/PCS technologies are classified into *generations* depending on their capabilities (Table 1). Most systems currently in use are second generation (2G), and have low data rates (for example 14.4 kbps for IS-95.) The 3G systems are designed to provide much higher data rates, for example 2.4 Mbps for stationary users using the IS-856 technology [5]. A hybrid generation, denoted 2.5G, is an interim solution for the current 2G networks to have 3G-like capabilities. 2.5G data rates fall between 2G and 3G data rates, such as the 64 kbps for IS-95B. 4G systems are currently in the design stage, and they have minimum target data rates of 10-20 Mbps.

Table 1 Major cellular/PCS technologies

Generation	Examples	Data Rate
2G	IS-136, IS-95 (cdmaOne), GSM, PDC	14.4 kbps for IS-95
2.5G	IS-95B, EDGE, HSCSD, GPRS	64 kbps for IS-95B
3G	W-CDMA, CDMA2000 (IS-2000), CDMA 2000 1xEV-DO (IS-856)	2.4 Mbps for IS-856
4G	In development	10-20 Mbps for stationary users, 2 Mbps for vehicular users

The second major group of wireless technology, the WLAN, uses the unregulated 2.4 GHz ISM (Industrial, Scientific, and Medical) frequency bands for transmission [4]. Compared to cellular/PCS, WLANs have shorter range but higher data rate. Major WLAN technologies and their maximum data rates are: Bluetooth [6], 723 kbps; IEEE 802.11b (Wi-Fi) [7], 11 Mbps; and IEEE 802.11a/g [7], 54 Mbps (Table 2). UWB (Ultra Wide Band) is a developing technology that promises very high data rates at short ranges while consuming very little power [4]. Currently however, the usage of UWB for practical applications is uncommon.

Table 2 Major WLAN technologies

Technology or Standard	Range	Data Rate
Bluetooth	6 m in low power mode, 10 m in medium power mode, 100 m in high power mode	Max. 723 kbps
IEEE 802.11b (Wi-Fi)	50 m indoors typical	Max. 11 Mbps, decreases with distance
IEEE 802.11a/g	25 and 50 m indoors typical respectively for 802.11a and 802.11g	Max. 54 Mbps, decreases with distance
UWB (in development)	Tested at ~10 m	Tested at 50-100 Mbps

3. Cipher suite

A minimalist cipher suite must include at least one algorithm for each of the following security functions: user authentication, confidentiality, integrity, and digital signature. In this section, we describe the algorithms that we selected for each of these functions (Table 3) and our rationale for this selection.

3.1. ECC for user authentication and digital signature

Elliptic curve cryptography (ECC) [8, 9] is a faster and simpler alternative for providing user authentication and digital signatures as compared to the integer public-key algorithms such as RSA or DH (Diffie-Hellman) [10]. Compared to these integer algorithms, ECC offers a much higher security per key bit, so that a desired level of security can be attained using smaller keys.

For example, eDH (elliptic-curve DH) with 163-bit keys provides more security than DH with 1024-bit keys (Table 4). More importantly, smaller keys are more preferable in mobile environments since they require less storage space and fewer computations, as well as consume less bandwidth during transmission. For our cipher suite, we have chosen the NIST-recommended 163-bit, 233-bit, and 283-bit key sizes [11], which provide sufficient security for all practical purposes. We use eDH [12] and eEl-Gamal (elliptic-curve El-Gamal) for user authentication, and eDSA (elliptic-curve Digital Signature Algorithm) [11] for digital signatures.

Table 3 Cipher suite

Function	Algorithm
User authentication	eDH, eEl-Gamal
Confidentiality	AES with 128-bit key
Integrity	SHA-1, SHA-256, AES-hash
Digital signature	eDSA

Table 4 Key size for equivalent security

Integer algorithm key size in bits (e.g. DH)	**Elliptic-curve algorithm key size in bits** (e.g. eDH)
512	106
768	132
1024	**160**
2048	210
21000	600

3.2. AES for confidentiality

AES is the US federal standard for block encryption [13]. It operates on 128-bit blocks of data, using 128-bit, 192-bit, or 256-bit keys. When 128-bit key is used, the cipher involves 10 rounds. AES has been designed to have very fast software and hardware implementations in both desktop and constrained environments. We use the optimized implementation described in [14], which relies on frequent table lookups.

An overview of the AES round structure is shown in Figure 1. The input to the round is the 128-bit block made up of four 32-bit words. AES uses four 1 kB tables, labeled TA, TB, TC, and TD. Each table has 256 entries, and each entry contains 4 bytes. During the round, the least-significant byte of each word is used as an index into TA; the next byte is used as an index into TB; and so on, until all tables are accessed four times. Finally, the four table lookup results (for each input word) are XORed among themselves and with the corresponding round key.

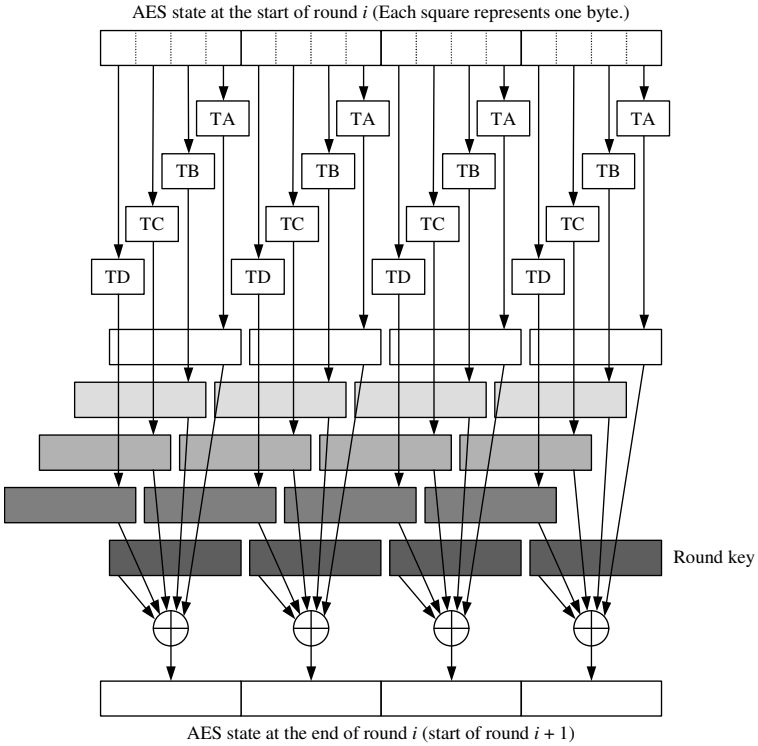

Figure 1 Overview of one AES round

In [15], we describe how the table lookups in this AES implementation can be accelerated using efficient addressing in load instructions. In this paper, we further optimize the AES execution by using small on-chip tables separate from the main memory, and new instructions to access these tables in parallel (Section 4).

3.3. SHA-1, SHA-256, and AES-hash for integrity

For integrity, we use two of the hash algorithms recommended in the Secure Hash Standard (SHS) [16]: SHA-1 and SHA-256. Both algorithms are designed for 32-bit datapaths and operate on 512-bit blocks. They produce 160-bit and 256-bit hashes respectively.

We also include a third hash algorithm, called AES-hash [13], to demonstrate that our AES optimizations can also be used to provide integrity. AES-hash generates a 256-bit hash, and it is derived from AES with 256-bit keys.

4. PAX instruction set

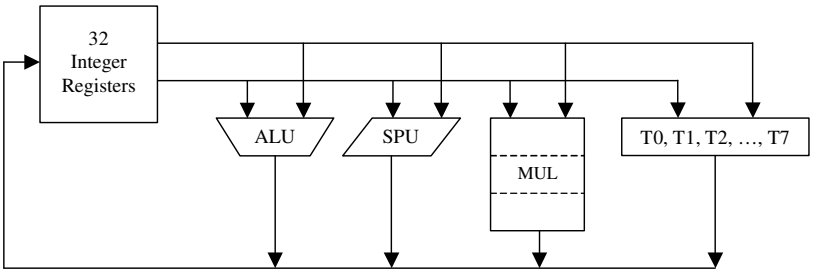

Figure 2 Single-issue PAX processor with 3 functional units

The datapath of a single-issue PAX processor is shown in Figure 2. The instructions are executed by three functional units: the arithmetic-logic unit (ALU), the shift-permute unit (SPU), and the polynomial multiplier (MUL) with 3 pipeline stages. The inputs and the output of the multiplier are word-sized, and only the lower or the higher half of the product is computed. There are eight on-chip lookup tables, labeled T0-T7, that are used for fast symmetric-key cryptography. The size, structure, and usage of these tables will be explained in Section 4.3. There are 32 integer registers, numbered R0 through R31. R0 is hardwired to 0.

4.1. ALU and shift instructions

In Table 5, we list the PAX instructions organized into major groups. The default word size of a PAX processor is 32 bits (PAX32), and it can be scaled up to 64 bits (PAX64) or 128 bits (PAX128). The default size of 32 bits is motivated by the fact that AES, SHA-1, and SHA-256 all have special optimized implementations for this word size. The only exception to datapath scalability is that the 32-bit versions of four instructions must be included in the instruction set even when the word size is 64 bits or 128 bits. This is necessary in order not to incur performance degradation in SHA-1 and SHA-256 at these larger word sizes. These four instructions are: **add**, **roti**, **load**, **store**.

Table 5 PAX instructions[1]

Arithmetic and Load Immediate Instructions	
Mnemonic	**Description**
add[2]	$c \leftarrow a + b$
addi	$c \leftarrow a + imm$
sub	$c \leftarrow a - b$
subi	$c \leftarrow a - imm$
loadi.z.*s*	Load *imm* into the 16-bit subword *s* of *c*; clear other subwords.
loadi.k.*s*	Load *imm* into the 16-bit subword *s* of *c*; keep other subwords unchanged.

Logical Instructions	
and	$c \leftarrow a \,\&\, b$
andi	$c \leftarrow a \,\&\, imm$
or	$c \leftarrow a \mid b$
xor	$c \leftarrow a \oplus b$
xori	$c \leftarrow a \oplus imm$
not	$c \leftarrow \overline{a}$

Shift Instructions	
sll	$c \leftarrow a << b$
slli	$c \leftarrow a << imm$
srl	$c \leftarrow a >> b$, with zero extension
sra	$c \leftarrow a >> b$, with sign extension
srai	$c \leftarrow a >> imm$, with sign extension
roti[2]	$c \leftarrow a <<< imm$
shrp	$c \leftarrow [(a \,\|\| b) >> imm]_{\mathrm{L}}$
hibit	See text

Permute Instructions	
shuffle.low	See text
shuffle.high	See text

Branch Instructions	
beqz	Branch if $a = 0$
bnez	Branch if $a \neq 0$

Branch Instructions (continued)	
bg	Branch if $a > b$
bge	Branch if $a \geq b$
jmp	Branch unconditionally

Other Program Flow Control Instructions	
call	Call subroutine
return	Return from subroutine
trap	Trap

Load/Store and Table Lookup Instructions	
load[2]	$c \leftarrow \mathrm{MEM}[a + imm]$
store[2]	$c \rightarrow \mathrm{MEM}[a + imm]$
load.update	$c \leftarrow \mathrm{MEM}[a + imm]$, $a \leftarrow a + imm$
store.update	$a \leftarrow a + imm$, $c \rightarrow \mathrm{MEM}[a]$
ptlu.*subword.table .offset.step*	See text
ptlw.*table.offset*	See text

Multiply Instructions	
polmul.low	$c \leftarrow (a \otimes b)_{\mathrm{L}}$
polymul.high	$c \leftarrow (a \otimes b)_{\mathrm{H}}$

[1] *c*, *a,* and *b* correspond to the values in the destination and source registers respectively. *imm* represents an immediate value given in the instruction word. Subscripts **L** and **H** indicate the lower and higher halves of a quantity respectively. MEM is the memory array. $\oplus$ denotes xor. <<< denotes a rotate. $\|$ denotes concatenation. $\otimes$ denotes polynomial multiplication. Multiply instructions are pipelined and have 3-cycle execution latency; all other instructions are single-cycle.

[2] The 32-bit version of this instruction must be included in the instruction set at all word sizes.

PAX has the basic ALU and logical instructions: **add**, **subtract**, **and**, **xor, or**, **not**, with some of these having both register and immediate versions. Loading of a register by an immediate is done with **loadi.z.*s*** and **loadi.k.*s*** as in PLX [3]. In **loadi.z.*s***, one 16-bit subword of the destination register (selected via the ***s*** field) is written with the 16-bit immediate given in the instruction

word. The remaining subwords are cleared to zero. In **loadi.k.*s*** the remaining subwords are kept unchanged.

Of the shift instructions, the 32-bit rotate immediate (**roti**) is very frequently used in SHA-1 and SHA-256. Therefore, it is necessary that the 32-bit version of this instruction is kept even in 64-bit and 128-bit datapaths.

In the **shrp (**shift right pair) instruction, two source registers are concatenated and shifted right by a number of bits given in the immediate field of the instruction. The lower half of the shifted result is then written to the destination register (Figure 3). This is instruction is very useful to shift data objects that span multiple words, such as the 163-bit binary polynomials that are used frequently in ECC.

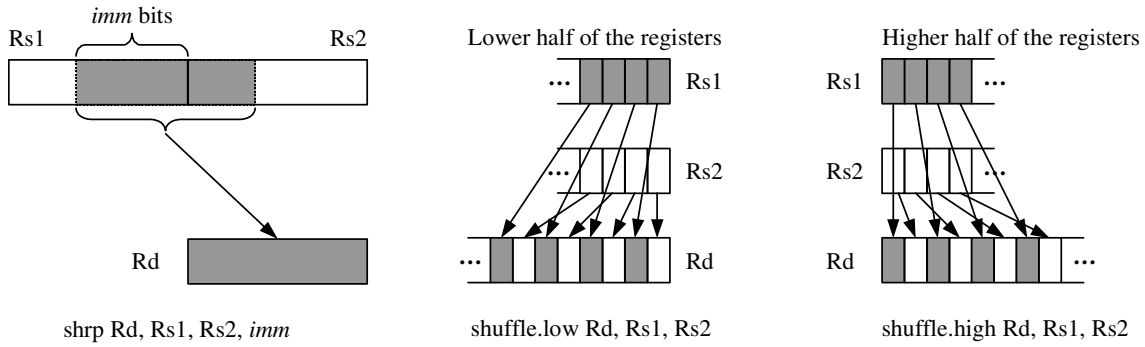

Figure 3 Shuffle.low, shuffle.high, shrp instructions

hibit is a new instruction that is very useful in polynomial arithmetic (used in ECC), especially in computing the multiplicative inverse of a polynomial [12, 17], which is the most costly polynomial operation. **hibit** writes the index of the most significant bit of the source register plus 1 to the destination register. If the source register is all 0s, then 0 is written to the destination register. In particular, **hibit** provides a huge speedup in computing the degree of a polynomial, which is a frequent operation in polynomial inversion [12].

4.2. Permute instructions

PAX has two bit permutation instructions: **shuffle.low** and **shuffle.high**. Both instructions read individual bits alternating between the two source registers, and write these bits to the destination register (Figure 3). In **shuffle.low**, the lower halves of the source registers read, in **shuffle.high** the higher halves are read.

Shuffle instructions are very useful in computing the square of a binary polynomial. (A *binary polynomial* is a polynomial whose coefficients are either 0 or 1. Because our ECC implementation uses binary fields where each field element is represented as a binary polynomial, such polynomial arithmetic constitutes the bulk of the ECC workload.) For example, let $a(x) = x^3 + x^2 + 1$. In binary, $a(x)$ will be represented as $(1101)_2$. The square of $a(x)$ can be computed by multiplying it with itself:

$$a^2(x) = a(x) \times a(x) = (x^3 + x^2 + 1) \times (x^3 + x^2 + 1) = x^6 + x^4 + 1 + \cancel{2x^5} + \cancel{2x^3} + \cancel{2x^2} = x^6 + x^4 + 1$$

Note that the x^5, x^3, and x^2 terms have vanished because all operations on the coefficients are modulo 2. The binary representation of the above is:

$$a^2(x) = a(x) \times a(x) = (\mathbf{1101})_2 \times (\mathbf{1101})_2 = (\mathbf{1}0\mathbf{1}0\mathbf{0}0\mathbf{1})_2$$

Notice that the squaring of $a(x)$ corresponds to interleaving the original bits in the binary representation of $a(x)$ with 0s. This is valid for all binary polynomial. The interleaving of the consecutive bits of a register with 0s corresponds to a shuffle with R0 (Figure 3). Using **shuffle.low** and **shuffle.high**, the squaring of a 163-bit polynomial can be completed in only 11 instructions if the word size is 32 bits, and in only 3 instructions if the word size is 128 bits.

4.3. Load, store, and ptlu instructions

The basic **load** and **store** instructions use base+displacement addressing. 32-bit variants of these instructions must also be preserved at all word sizes because many cryptographic algorithms are

optimized for a 32-bit memory pipe. In the **update** versions, post-modify is used for loads; pre-modify is used for stores.

The **ptlu** instruction is used to access the eight on-chip tables and perform parallel table lookups. This instruction has the following format:

ptlu.*subword*.*table*.*offset*.*step* Rd, Rs

The *subword*, *table*, *offset*, and *step* are given as immediates in the instruction. The 3-bit *table* field is used to select one of the eight tables for lookup (T0 through T7). These tables have 256 entries each, and each entry is word-sized. The total size of the tables is 8 kB, 16 kB, and 32 kB for PAX32, PAX64, and PAX128 respectively. The byte-sized indices used to access the tables are read from Rs. Because there are no effective address computations, the table accesses can be performed in the execution stage of the instruction, and the read values can be immediately forwarded and used by another instruction in the next cycle. (In a regular **load** instruction, the load-use interlock will normally prohibit such immediate using of the loaded data.) Another benefit of using an on-chip table is the invariability of the access time for any single table lookup. Unlike the data memory, where a single lookup can take either a single cycle (if it is a cache hit) or many cycles (if it is a cache miss), a **ptlu** access always takes a single cycle. This allows us to report more accurate performance results for our symmetric-key ciphers since we know the exact table access latency.

The 4-bit *offset* field is used to select the first index in Rs that will be used to access the table. The 4-bit *step* field gives the distance (in bytes) between the subsequent bytes in Rs that are used as indices when multiple lookups are performed. The *subword* field selects the size of the data that is read from the table. Because certain combinations of the four subop fields will not be meaningful, the programmer or the compiler is responsible for avoiding these.

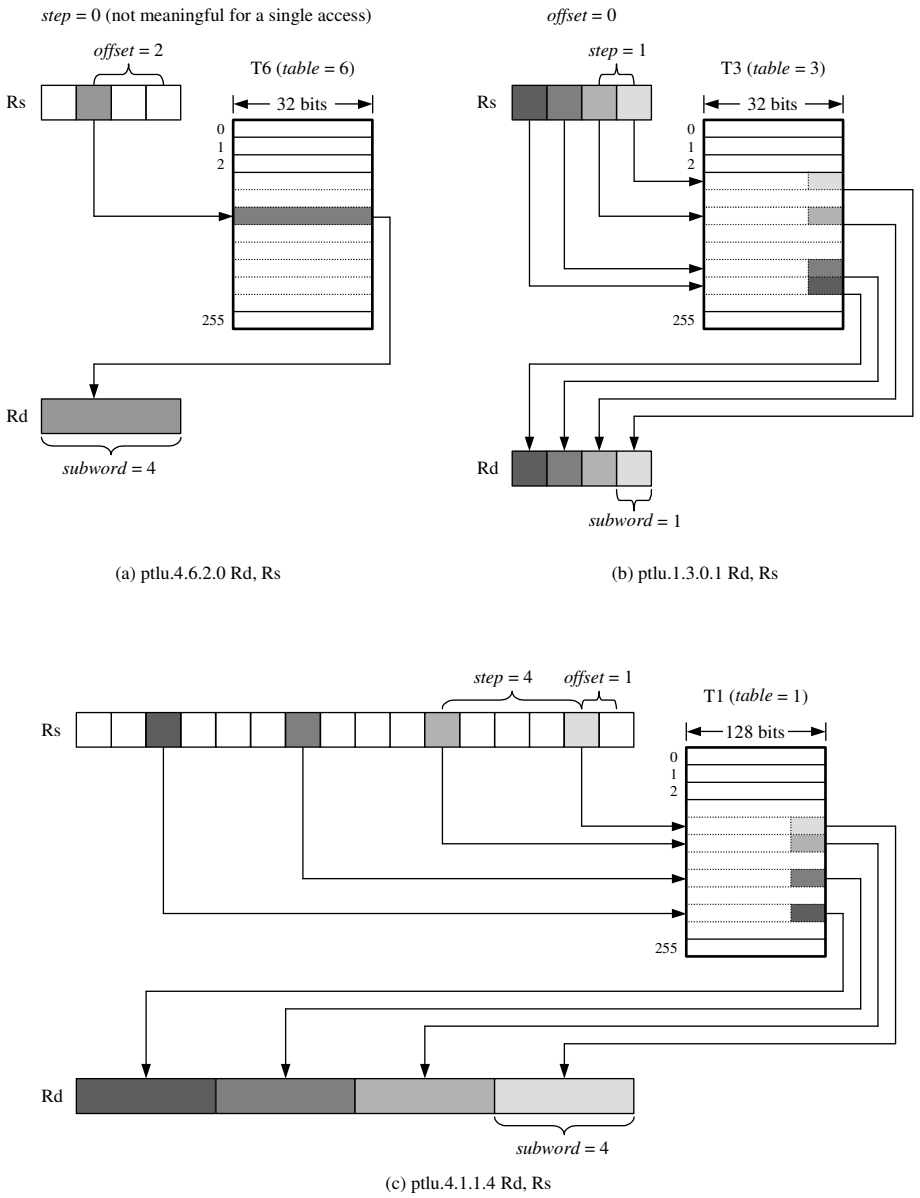

Figure 4 Examples of the **ptlu** instruction in PAX32 (a), PAX64 (c), and PAX128 (c)

Figure 4 contains four examples of the **ptlu** instruction. Figure 4a shows the **ptlu** used in PAX32 to perform a single lookup from T6, using the third byte of Rs as index. Such lookups are very common in symmetric-key ciphers, where they are called S-box lookups and are used to achieve cryptographic confusion. Figure 4b shows four parallel lookups from T3, where the index for each lookup is a different byte of Rs. This type of lookup is used in AES key expansion.

Figure 4c is for PAX128, and it depicts how we use **ptlu** for fast AES implementation. Consider the four TB lookups in the AES round shown in Figure 1, and assume that T0-T3 have been initialized to TA-TD respectively. In PAX128, four lookups of the same table (T1) can be performed with the single instruction **ptlu.4.1.1.4 Rd, Rs** (Figure 4c). Two separate **ptlu** instructions would be needed in PAX64 to perform the same four lookups, and four **ptlu** instructions are needed in PAX32. All of the 16 table lookups in an AES round can be completed using just four **ptlu** instructions in PLX128, with each instruction performing four parallel lookups and the *table* field changing from 0 to 1 to 2 to 3. Because the number of simultaneous lookups quadruples as the word size is increased from 32 bits to 128 bits, the performance of AES also quadruples. Such linear performance scaling is unattainable with a simple RISC instruction set, or with the optimizations in [15].

The writing of the tables T0-T7 is done with the **ptlw** instruction, which has the following format:

ptlw.*table* Rs1, Rs2

The 3-bit *table* field selects one of the eight tables, and the least significant byte of Rs2 selects one of the 256 entries in this table. The data to be written to the selected entry of the table is read from Rs1.

4.4. Multiply instructions

Only the multiplication of two binary polynomials is required by the algorithms in our cipher suite. Similar to the polynomial squaring (Section 4.2), the coefficients of the product polynomial are computed component-wise. For example, let $a(x) = x^3 + 1$ and $b(x) = x^2 + x$. The binary representations of $a(x)$ and $b(x)$ are $(1001)_2$ and $(0110)_2$ respectively. The product of $a(x)$ and $b(x)$ is computed as:

$$a(x) \times b(x) = (x^3 + 1) \times (x^2 + x) = x^5 + x^4 + x^2 + x$$

Or, in binary representation as:

$$a(x) \times b(x) = (1001)_2 \times (0110)_2 = (110110)_2$$

A standard integer multiplier cannot be used to perform such polynomial multiplication. Software methods, such as the shift-and-xor algorithm [17, 18], are very slow. Hardware solutions recommend the usage of either a dedicated polynomial multiplier (also known as a binary field multiplier), or dual-field multiplier, which can perform both integer multiplication and polynomial multiplication. Because no integer multiplication is required for the algorithms in our cipher suite, we use a polynomial multiplier, which can execute two instructions: **polymul.low** and **polymul.high**. Both instructions have 3-cycle execution latency. Because the multiplier is pipelined, the full latency of these may be hidden through instruction scheduling. In **polymul.low**, the multiplier generates only the lower half of the product; in **polymul.high**, it generates only the higher half.

PAX also has one unconditional (**jmp**) and four conditional (**beqz**, **bnez**, **bg**, **bge**) branch instructions. The **call** and **return** instructions are used to branch to a subroutine and return from a subroutine respectively.

5. Performance

We now compare the performance of PAX with the results reported in [17] for a Pentium II workstation (for public-key cryptography), and with the results from a basic RISC instruction set (for symmetric-key cryptography). In all cases, we also show the additional speedup that can be obtained with the datapath scalability feature by varying the word size from 32 bits to 64 bits to 128 bits.

5.1. ECC

Table 6 summarizes the results of the arithmetic operations for the 163-bit, 233-bit, and 283-bit binary polynomials used in ECC. The first column gives the execution times for the Pentium II (PII) workstation [17]; the second column is for a single-issue 32-bit PAX processor, denoted PAX32/1. The last three columns of Table 6 show the speedups that can be obtained if the word size in PAX were doubled and quadrupled. For polynomial addition, squaring, and multiplication, an *additional* 10× to 20× speedup over PAX32/1 is provided by datapath scalability.

Table 7 shows the execution times for elliptic-curve point multiplication (the key operation in ECC) for 163-bit, 233-bit, and 283-bit keys, and Table 8 shows the execution times for eDH, eEl-Gamal encryption, and eEl-Gamal decryption for the 163-bit key. The speedup of the PAX32/1 processor over the PII is more than 2× for each algorithm, and this increases as the word size increases.

Table 6 Execution times for binary field arithmetic operations

Field size (bits)	**Operation**	**Execution time** (cycles)		**Speedup over PII**		
		PII	**PAX32/1**	**PAX32/1**	**PAX64/1**	**PAX128/1**
163	Addition	40	18	2.2×	13.3×	20.0×
	Squaring	160	28	5.7	20.0	53.3
	Multiplication	1200	142	8.5	33.3	92.3
	Reduction	72	149	0.5	0.7	2.5
	Inversion	16104	11873	1.4	2.1	2.7
233	Addition	48	37	1.8	5.3	24.0
	Squaring	220	42	5.8	27.5	55.0
	Multiplication	2028	315	6.9	32.7	144.9
	Reduction	88	256	0.5	0.9	3.4
	Inversion	29220	29960	1.0	2.2	3.4
283	Addition	52	41	1.7	4.3	17.3
	Squaring	300	49	6.7	23.1	60.0
	Multiplication	2492	387	7.7	25.4	75.5
	Reduction	140	343	0.6	1.1	3.2
	Inversion	38596	37196	1.0	2.2	3.0

Table 7 Execution times for elliptic curve point multiplication

	Execution time (cycles in thousands)		Speedup over PII		
Field size (bits)	**PII**	**PAX32/1**	**PAX32/1**	**PAX64/1**	**PAX128/1**
163	1296	576	2.3×	4.5×	13.9×
233	3079	1225	2.5	6.6	24.3
283	4641	1882	2.5	6.0	16.9

Table 8 Execution times for 163-bit eDH, eEl-Gamal encryption, and eEl-Gamal decryption

	Execution time (cycles in thousands)		Speedup over PII		
Operation	**PII**	**PAX32/1**	**PAX32/1**	**PAX64/1**	**PAX128/1**
eDH	2592	1152	2.3×	4.5×	13.9×
eEl-Gamal encryption	2593	1153	2.3	4.5	13.9
eEl-Gamal decryption	1313	588	2.2	4.4	13.2

Table 9 shows the execution time for 163-bit eDSA signature generation and verification for 1 kB and 1 MB messages. The results for 1 kB signature verification are especially important because this corresponds to the time required for the verification of an X.509 digital certificate. For short messages, the additional speedup obtained over PAX32/1 through datapath scalability is more than 5× for PAX128/1. For long messages, no additional speedup is obtained because the execution time is dominated by SHA-1 instead of the elliptic-curve operations.

Table 9 Execution times for 163-bit eDSA on 1 kB and 1 MB messages

		Execution time (cycles in thousands)	Speedup over PAX32/1	
Operation	**Message size**	**PAX32/1**	**PAX64/1**	**PAX128/1**
Signature generation (including SHA-1 hash)	1 kB	609	1.9×	5.1×
	1 MB	19956	1.0	1.0
Signature verification (including SHA-1 hash)	1 kB	1186	2.0	5.6
	1 MB	20533	1.0	1.0

5.2. AES

Table 10 shows the execution times for 128-bit AES key expansion and encryption. We report five results for AES. The first is for a basic RISC instruction set that performs table lookups using standard load instructions with base+displacement addressing. The second case is for an

enhanced RISC as described in [15]. The last three results are for the PAX32/1, PAX64/21, and PAX128/1, all of which use the **ptlu** instruction. The speedups over the basic RISC for these three cases are 2.3×, 4.6×, and 9.2× respectively.

Table 10 Execution times for 128-bit AES key expansion and encryption

Operation	**Execution time** (cycles)			**Speedup over basic RISC**		
	Basic RISC	**Basic RISC with fast addressing**	**PAX32/1**	**PAX32/1**	**PAX64/1**	**PAX128/1**
Key expansion	330	210	210	1.6×	1.6×	1.6×
Encryption	840	360	360	2.3	4.6	9.2

Table 11 Execution times on PAX32/1 for SHA-1 and SHA-256

Algorithm	**Hash size** (bits)	**Cryptographic strength** (bits)	**Block size** (bits)	**Execution time per block** (cycles)
SHA-1	160	80	512	1182
SHA-256	256	128	512	2512

Table 12 Execution time for AES-hash

				Execution time per block (cycles)		
	Hash size (bits)	**Cryptographic strength** (bits)	**Block size** (bits)	**PAX32/1**	**PAX64/1**	**PAX128/1**
AES-hash	256	128	256	1140	780	600
Speedup over SHA-256 (normalized to equal block size) →				1.1×	3.2×	4.2×

5.3. SHA-1, SHA-256, and AES-hash

Table 11 shows the execution times for SHA-1 and SHA-256. Only PAX32/1 results are reported because these hashing algorithms have extremely serial structures, and therefore do not benefit from datapath scalability. For AES-hash, where datapath scalability is useful, we report results for the three word sizes, and also show the speedups over SHA-256 (Table 12).

5.4. WTLS

Security protocols on the Internet usually use encryption and hashing together in order to ensure both the confidentiality and the integrity of the transmitted data. One well-known example is the

WTLS protocol [19], which stands for Wireless Transport Layer Security, and which is very similar to the TLS (Transport Layer Security) protocol used in wired networks.

Each WTLS session between a host and a client starts with the client's authenticating the host. This step, which is called a WTLS handshake, involves a series of cryptographic computations such as verification of the digital certificate of the host, exchanging of a session key, etc. The execution time for the aggregate of these steps for PAX32 is shown in Table 13, as well as the additional speedups that can be obtained through datapath scalability.

Table 13 Execution time for WTLS handshaking

Execution time (cycles in thousands)	**Speedup over PAX32/1**	
PAX32/1	**PAX64/1**	**PAX128/1**
2929	2.0×	5.9×

Table 14 Throughput of AES, SHA-1, and WTLS-record at various clock rates

Algorithm	**Processor**	**Throughput in Mbps at these clock rates**			
		100 MHz	**200 MHz**	**400 MHz**	**800 MHz**
AES encryption	PAX32/1	35.6	71.1	142.2	284.4
	PAX64/1	71.1	142.2	284.4	568.9
	PAX128/1	142.2	284.4	568.8	1137.8
SHA-1	PAX32/1	43.3	86.7	173.3	346.5
WTLS-record (includes SHA-1 based HMAC and AES encryption)	PAX32/1	14.7	29.4	58.7	117.5
	PAX64/1	20.2	40.5	80.1	161.9
	PAX128/1	25.0	49.9	99.9	199.7

After the handshake is complete, the WTLS protocol enters the *record* mode, where each message that is sent is first appended with a SHA-1 based HMAC (keyed hash), and then encrypted with a symmetric-key algorithm, which, in our case, is AES. We show the throughput of the WTLS-record phase for PAX in Table 14.

5.5. Performance summary

We observe that a PAX32/1 processor can perform exceptionally fast public-key cryptography using ECC. This is achieved by using four key instructions: **shuffle.low** and **shuffle.high** (used

in polynomial squaring); **polymul.low** and **polymul.high** (used in polynomial multiplication); and **hibit** (used in polynomial inversion). Compared to a PII workstation, these instructions provide speedups of 5.7×, 8.5×, and 1.4× respectively for squaring, multiplication, and inversion. Moreover, when the word size is increased to 128 bits through datapath scalability, these speedups increase to 53.3×, 92.3×, and 2.7× respectively. If 233-bit keys are used instead of 163-bit keys, the speedups further increase to 55.0×, 144.9×, and 3.4×.

In Table 14, we showed the throughputs for encryption and hashing on PAX processors at various clock speeds. The results for the 400 MHz clock are especially significant because this corresponds to the speed commonly used by today's high-end mobile information appliances such as the HP iPAQ and Jornada series Personal Digital Assistants (PDAs).

For AES and SHA-1, PAX32/1 performance is sufficient to match all data rates offered by existing and developing wireless technologies. By using **ptlu** instructions, we have achieved speedups of 2.3×, 4.6×, and 9.2× in AES compared to a basic RISC instruction set. As a result, a 60 MHz PAX32/1 processor can match the 20 Mbps target data rate of future 4G networks. This is significantly lower than our 400 MHz benchmark clock rate.

Table 15 Percent processor utilization required for AES at various wireless links speeds

Clock rate (MHz)	**Processor**	**Cellular/PCS Technology**				**WLAN Technology**			
		IS-95 (2G)	**IS-95B** (2.5G)	**IS-856** (3G)	**4G**	**Bluetooth**	**802.11b**	**802.11a/g**	**UWB**
400	PAX32/1	•	•	2	14	1	8	38	70
	PAX64/1	•	•	1	7	•	4	19	35
	PAX128/1	•	•	•	4	•	2	10	18

• means less than 1%. ♦ means more than 100%.

Table 16 Percent processor utilization required for SHA-1 at various wireless links speeds

Clock rate (MHz)	**Processor**	**Cellular/PCS Technology**				**WLAN Technology**			
		IS-95 (2G)	**IS-95B** (2.5G)	**IS-856** (3G)	**4G**	**Bluetooth**	**802.11b**	**802.11a/g**	**UWB**
400	PAX32/1	•	•	1	12	•	6	31	58

Table 17 Percent processor utilization required for WTLS-record at various wireless links speeds

Clock rate (MHz)	Processor	Cellular/PCS Technology				WLAN Technology			
		IS-95 (2G)	IS-95B (2.5G)	IS-856 (3G)	4G	Bluetooth	802.11b	802.11a/g	UWB
400	PAX32/1	•	•	4	34	1	19	92	♦
	PAX64/1	•	•	3	25	1	14	67	♦
	PAX128/1	•	•	2	20	1	11	54	100

In Tables 15-17, we show what percentage of a 400 MHz PAX processor's clock cycles are consumed by AES, SHA-1, and WTLS-record at varying word sizes and link speeds. For AES encryption at 4G cellular/PCS data rates, only 4% of the clock cycles of a PAX128/1 processor are used. For 2G, 2.5G, and 3G networks, less than 1% of the cycles are used. Even for the 100 Mbps data rate of the experimental UWB technology, only 18% of the cycles are utilized.

6. Conclusions

In this paper, we described PAX, a datapath-scalable, minimalist cryptographic processor for mobile environments, where the communication with the outside world is done wirelessly.

First, we gave a brief comparative review of the existing and developing wireless technologies. The data rates for these vary between 64 kbps (2.5.G) to 2.4 Mbps (3G) for cellular/PCS, and between 723 kbps and 54 Mbps for WLAN.

Second, we selected a set of cryptographic algorithms suitable for mobile environments. For key exchanges, user authentication, and digital signatures, we have chosen 163-bit, 233-bit, and 283-bit ECC, which offers savings in execution time, storage, bandwidth usage, and computation time. For confidentiality and data integrity, we use AES, SHA-1, and SHA-256. In addition to being US federal standards, these algorithms have fast optimized implementations.

Third, we described the PAX instruction set. PAX is similar to PLX [1-3] in design philosophy in that it is a minimalist RISC-like instruction set with few new instructions that

provide huge speedups in key algorithms. In PAX, these instructions are: **shuffle.low** and **shuffle.high** (used in polynomial squaring); **polymul.low** and **polymul.high** (used in polynomial multiplication), **hibit** (used in polynomial inversion); and **ptlu** and **ptlw** (used for the table lookups in AES and AES-hash). These provide significant overall speedups for both ECC and AES. The datapath scalability provides an *additional* 10× to 20× speedup when the word size is increased from 32 bits to 128 bits.

Finally, we showed the host processor speeds needed to match the data rates offered by the wireless technologies we considered in the first step. For AES encryption and SHA-1 hashing, 7 MHz PAX32/1 is sufficient to match the 2.4 Mbps data rate of future 3G cellular networks. To match the 54 Mbps data rate of the 802.11a/g WLAN technology, the clock rate needs to be 150 MHz. Both figures are well under the 400 MHz clock rate used in today's high-end mobile information appliances such as the HP iPAQ and Jornada series PDAs.

In summary, we have shown that it is possible to achieve high-performance cryptography by using a suitable cipher suite and a simple RISC-like instruction set with few additional instructions. With the convergence of cellular and WLAN technologies, and the emergence of uniform Internet access on wired and wireless networks, it is necessary for mobile devices to support common Internet security protocols at a low-enough cost. We have shown that PAX is an excellent candidate to fulfill this requirement, either as an embedded processor, a cryptographic co-processor alongside a general-purpose processor, or a security module in a system-on-chip.

References

1. Ruby B. Lee and A. Murat Fiskiran, "PLX: A Fully Subword-Parallel Instruction Set Architecture for Fast Scalable Multimedia Processing," *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo (ICME 2002), pp. 117-120*, August 2002.

2. Ruby B. Lee, A. Murat Fiskiran, Zhijie Shi, and Xiao Yang, "Refining Instruction Set Architecture for High-Performance Multimedia Processing in Constrained Environments," *Proceedings of the 13th International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2002)*, *pp. 253-264*, July 2002.
3. Ruby B. Lee, et al., "PLX Project at Princeton University", http://palms.ee.princeton.edu/plx.
4. Theodore S. Rappaport, A. Annamalai, R. M. Buehrer, and William T. Tranter, "Wireless Communications: Past Events and A Future Perspective," *IEEE Communications Magazine, vol. 40, no. 5, pp. 148-161*, May 2002.
5. Richard Parry, "Overlooking 3G," *IEEE Potentials*, *vol. 21, no. 4, pp. 6-9*, October 2002.
6. Charles D. Knutson, David K. Vawdrey, and Eric S. Hall, "Bluetooth," *IEEE Potentials*, *vol. 21, no. 4, pp. 28-31*, October 2002.
7. IEEE P802.11, The Working Group for Wireless LANs, http://grouper.ieee.org/groups/802/11/.
8. Victor S. Miller, "Use of Elliptic Curves in Cryptography," *Lecture Notes in Computer Science, no. 218, pp. 417-426*, Springer-Verlag, 1986.
9. Neal Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation, vol. 48, no. 177, pp. 203-209*, 1987.
10. Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone, "Handbook of Applied Cryptography," CRC Press, ISBN 0-8493-8523-7, October 1996.
11. NIST (National Institute of Standards and Technology), "Digital Signature Standard (DSS) – FIPS Pub. 186-2," February 2000.
12. Richard Schroeppel, Hillarie Orman, Sean O'Malley, and Oliver Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," *Advances in Cryptology - CRYPTO'95, Lecture Notes in Computer Science*, *pp. 43-56*, Springer-Verlag, 1995.
13. NIST, "Advanced Encryption Standard (AES) - FIPS Pub. 197," November 2001.
14. Joan Daemen and Vincent Rijmen, "AES Proposal: Rijndael," *AES submission to NIST*, 1998.
15. A. Murat Fiskiran and Ruby B. Lee, "Performance Impact of Addressing Modes on Encryption Algorithms," Proceedings of the International Conference on Computer Design (ICCD 2001), pp. 542-545, September 2001.
16. NIST, "Secure Hash Standard (SHS) – FIPS Pub. 180-2," August 2002.
17. Darrel Hankerson, Julio Lopez Hernandez, and Alfred Menezes, "Software Implementation of Elliptic Curve Cryptography Over Binary Fields," *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000), pp. 1-24*, August 2000.
18. A. Murat Fiskiran and Ruby B. Lee, "Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms for Constrained Environments," *Proceedings of the 5th Annual IEEE International Workshop on Workload Characterization (WWC-5)*, *pp. 127-137*, November 2002.
19. WAP Forum, "Wireless Application Protocol 2.0 Technical White Paper", http://www.wapforum.org/.