PAX: A Cryptographic Processor with Parallel Table Lookup and Wordsize Scalability

Ruby B. LEE, *Fellow, IEEE*, Murat FISKIRAN, Michael WANG,
Yedidya HILEWITZ, *Student Member, IEEE*, and Yu-Yuan CHEN

***Abstract*--Cryptographic algorithms are important components in secure systems. We propose PAX, a tiny processor for both symmetric-key and public-key ciphers. PAX's goal is to provide the flexibility of software implementations, with performance comparable to hardware implementations for important ciphers. Based on workload characterization studies, we propose a few powerful instructions that provide huge speedups for critical operations found in many symmetric-key and public-key ciphers. A novel Parallel Table Lookup instruction enables multiple tables to be accessed in parallel by a single instruction; it also combines these parallel table results in a unique combinatorial tree. We achieve a software implementation of AES-128 in just 22 cycles using PAX-128. PAX also has bit and byte permutations, and binary-field arithmetic. Elliptic-Curve Cryptography speedup up to 25x is achieved.**

A distinctive feature of PAX is wordsize scalability, where the same instruction set can be synthesized into processors with different word-sizes. This is a new dimension in the processor design space, orthogonal to more traditional multi-issue techniques in superscalar or VLIW processors. We show that wordsize scaling provides speedups that are significantly higher with lower implementation complexity, for cryptographic processing. Wordsize scaling can be combined with multi-issue or multi-core scaling for even higher performance.

***Index Terms*— processor, crypto acceleration, AES, Elliptic Curve Cryptography, parallelism, table lookup, scalability, permutation, Instruction Set Architecture (ISA), ASIP, binary field multiplier**

I. Introduction

THIS paper describes the architecture and implementation of PAX, a small processor with a few special instructions for accelerating both symmetric-key and public-key cryptography algorithms. A significant advantage of a programmable processor like PAX over hardware ASIC (Application Specific Integrated Circuit) implementations of ciphers, is that a single chip can implement any number of ciphers. Our goal is to achieve cryptographic processing with the flexibility of software implementations but at a performance comparable to hardware implementations for the most important ciphers, e.g, AES [5]. In addition, while we target mobile devices, we want PAX to be a scalable architecture, for less resource-constrained devices

Manuscript received October 31, 2007. This work was supported in part by the National Science Foundation under Grants 0326372 and 0430487. All authors are with the Princeton Architecture Laboratory for Multimedia and Security (PALMS), Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: rblee@princeton,edu)

which desire higher performance.

Symmetric-key ciphers can be used to encrypt information sent across the public Internet or wireless networks, to protect against eavesdropping or observation attacks [1][2]. They are also useful for encrypting data or programs stored in memory, disks or on-line storage, to provide confidentiality. In symmetric-key ciphers, a plaintext message P is encrypted with a secret key K. The encrypted data (ciphertext) can then be transmitted or stored. It can only be decrypted using the same cipher and secret key. Symmetric-key ciphers are very efficient in encrypting large amounts of data, hence they are preferred for bulk encryption [2]. Examples of widely-used symmetric-key ciphers are 3DES [4] and AES [5].

Public-key cryptography, used with the appropriate security protocols, can provide essential security features such as authentication and digital signatures. This can thwart masquerading attacks [1]. Public-key ciphers use two keys for each party: a private-key, which is always kept secret, and a public-key, which can be posted publicly [2][3]. A plaintext message encrypted with a public key can only be decrypted with the corresponding private key. In the reverse direction, a message encrypted (or signed) by a private key can be decrypted (or verified) by anyone with the corresponding public key. Because public-key ciphers are up to three orders of magnitude slower than symmetric-key ciphers, they are not used for bulk encryption [2]. Instead they are used for user and device authentication, digital signatures, and for setting up symmetric-keys for bulk encryption. Examples of important public-key ciphers are RSA [2] and DSA [6].

Cryptography processing on mobile wireless devices is particularly challenging. The wireless communication medium cannot be physically secured, necessitating continuous use of crypto-processing to protect against eavesdropping attacks. In addition, cryptography is very compute-intensive, whereas mobile devices are typically very resource-constrained. Severe negative impacts of crypto-processing on the performance and power consumption of mobile devices have been documented [7] [8]. Furthermore, wireless link speeds keep increasing: emerging wireless technologies such as 4G and Ultra Wide Band (UWB) promise data rates as high as 100 megabits/second (Mbps) [9]. To fully utilize such high link speeds, the cryptographic performance of mobile devices must be increased while maintaining low energy consumption.

This paper gives a full architectural description of PAX, a general-purpose, tiny, scalable processor for high-performance, low-cost crypto-processing in resource-constrained devices. The PAX instruction set architecture (ISA) is derived by extending a minimalist RISC-like instruction set with a few carefully designed instructions that provide huge speedups in the performance-critical operations used in symmetric-key and public-key ciphers. This includes novel *parallel table lookup* instructions to accelerate symmetric-key ciphers. For public-key ciphers, PAX includes binary-field arithmetic and bit-level permutation instructions. A distinctive feature of PAX is *wordsize scalability,* which refers to the property that the same instruction set can be synthesized into processors with different wordsizes (e.g. 32-bit, 64-bit, or 128-bit). This is a new dimension in the processor design space that is orthogonal to the more traditional techniques like multiple-issue execution used in superscalar or Very Long Instruction Word (VLIW) processors. Our results indicate that wordsize scaling is very effective for improving the performance of both symmetric-key and public-key ciphers.

The rest of this paper is organized as follows. Section 2 provides an overview of the PAX instruction set. Section 3 analyzes the workload characteristics of important symmetric-key ciphers, while Section 4 describes the PAX features to accelerate these. Section 5 discusses workload characteristics of public-key ciphers, while Section 6 describes the PAX instructions that accelerate these. Section 7 describes our implementation of PAX, with area, latency and cycle time analysis. Section 8 presents the performance of PAX processors, showing the benefits of new instructions, wordsize scaling, multiple-issue execution, and combinations of these techniques. Section 9 reviews related past work and Section 10 concludes the paper.

II. Overview of the PAX Architecture

The datapath of a PAX processor is shown in Figure 1. The register file contains 32 architected registers, R0 through R31, where R0 is hardwired to zero. Instructions are 32 bits long and are executed by different functional units: the arithmetic-logic unit (ALU), the shift-permute unit (SPU), the binary-field multiplier and the Parallel Table Lookup (PTLU) module. The PTLU and the binary field multiplier are optional units for the smallest implementations. The PTLU module is an on-chip scratchpad memory used for fast parallel table lookups, while the binary field multiplier does GF(2) multiplications..

A novel feature of the PAX Instruction-Set Architecture (ISA) is that it is *word-size scalable*. i.e., the same instruction set can be synthesized into processors with different wordsizes. The wordsize of a processor is the size of its registers and datapaths. A PAX processor can be implemented with a wordsize of 32 bits, 64 bits or 128-bits, called PAX-32, PAX-64 or PAX-128, respectively. Scaling up from 64-bit words (which is the default) to 128-bit words may be desired to improve performance, or scaling down from 64-bit to 32-bit words may be preferred to limit cost and power.

Like other processors, multiple instruction issue techniques can be used in an implementation of PAX to improve performance. We use the term *IPC scaling* to refer to architectural methods where more than one instruction is issued per cycle. This includes superscalar or VLIW (Very Long Instruction Word) architectures. We show a novel alternative to IPC scaling in the wordsize scaling feature of PAX, which provides much better performance at a lower cost. Because wordsize scaling and IPC scaling are orthogonal dimensions in the processor design space, it is possible to use both methods simultaneously for even higher performance.

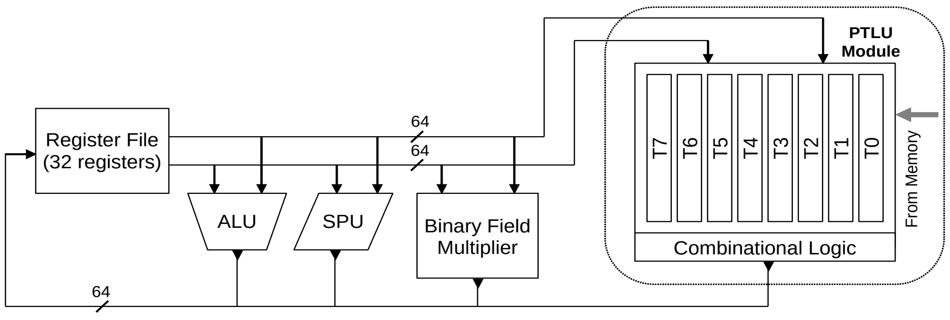


Figure 1: Single-issue PAX-64 processor

The PAX instruction set is shown in Table 1. The section labeled Base ISA includes the arithmetic, logical, shift, load, store, and branch instructions that are typical of a basic RISC instruction set. The section labeled PAX Extensions includes the PAX-specific PTLU, binary field multiply, and permutation instructions. Each of these novel instructions will be fully described in a subsequent section. We select a representative suite of cryptography algorithms and analyze their workload characteristics to identify performance-critical operations which then guide us in the design of PAX instructions for very fast software cryptographic processing.

III. Workload Characterization of Symmetric-Key Ciphers

Table 2 shows the symmetric-key ciphers we selected for this study. For each cipher, we show the block size, typical key size, and the number of rounds. Block size is the amount of data that the cipher can encrypt at a time, and key size relates to the strength of the cipher against cryptanalytic attacks [2],[3]. A round is a sequence of operations on the plaintext block that is repeated to compute the ciphertext. The input of a round consists of the output of the previous round and one or more subkeys, which are derived from the secret key. Common operations used in the rounds are table lookups, addition and subtraction, logical operations, shifts and rotates, multiplication, and permutations [2][10].

Table 1: PAX instruction set

	Instruction Class	Mnemonic	Operation	Explanation
Base ISA	ALU	`add`	Rd ← Rs1 + Rs2	Add These instructions support both signed and unsigned operands.
		`addi`	Rd← Rs + imm	Add 16-bit immediate
		`sub`	Rd ← Rs1 – Rs2	Subtract
		`subi`	Rd ← Rs – imm	Subtract 16-bit immediate
		`and`	Rd ← Rs1 & Rs2	Bitwise AND
		`andi`	Rd ← Rs & imm	Bitwise AND with 16-bit immediate
		`or`	Rd ← Rs1 | Rs2	Bitwise OR
		`ori`	Rd ← Rs | imm	Bitwise OR with 16-bit immediate
		`xor`	Rd ←Rs1 ^ Rs2	Bitwise XOR
		`xori`	Rd ← Rs ^ imm	Bitwise XOR with 16-bit immediate
		`not`	Rd ← !Rs	1’s complement
		`loadi.z.sel`	Rd ← imm	Load 16-bit immediate to an aligned 16-bit field of Rd, selected via the 3-bit sub-op sel, while clearing all remaining bits of Rd to zero.
		`loadi.k.sel`	Rd ← imm	Load 16-bit immediate to an aligned 16-bit field of Rd, selected via the 3-bit sub-op sel, while keeping all remaining bits of Rd unchanged.
	Shift	`sra`	Rd ← Rs1 >> Rs2	Shift right arithmetic by rightmost $\log_2(w)$ bits of Rs2
		`srai`	Rd ← Rs1 >> imm	Shift right arithmetic immediate; imm is $\log_2(w)$ bits
		`srl`	Rd ← Rs1 >> Rs2	Shift right logical by rightmost $\log_2(w)$ bits of Rs2
		`srli`	Rd ← Rs1 >> imm	Shift right logical immediate; imm is $\log_2(w)$ bits
		`sll`	Rd ← Rs1 << Rs2	Shift left logical by rightmost l $\log_2(w)$ bits of Rs2
		`slli`	Rd ← Rs1 << imm	Shift left logical immediate; imm is $\log_2(w)$ bits
		`shrp`	Rd ← (Rs1 || Rs2) >> imm	Concatenate Rs1 and Rs2, and shift right logical by imm bits. Rd receives the right word of the shifted result; imm is $\log_2(w)$ bits
	Memory	`load.sel`	Rd ← MEM[Rs + imm]	Load (store) an aligned word from (to) memory using base+displacement addressing. The sel field selects data size, which can be 4, 8, 16 bytes (but at most equal to w).
		`store.sel`	Rs → MEM[Rs + imm]	
	Branch	`beq`	PC ← PC + imm if Rs1 = Rs2	Branch to PC+displacement if Rs1 is equal to Rs2
		`bne`	PC ← PC + imm if Rs1 ≠ Rs2	Branch to PC+displacement if Rs1 is not equal to Rs2
		`bg`	PC ← PC + imm if Rs1 > Rs2	Branch to PC+displacement if Rs1 is greater than Rs2
		`bge`	PC ← PC + imm if Rs1 ≥ Rs2	Branch to PC+displacement if Rs1 is greater than or equal to Rs2
		`call`	R31 ← PC + 4, PC ← PC + imm	Call subroutine by saving PC+4 to R31, then changing PC to PC+displacement
		`return`	PC ← R31	Return from subroutine by changing PC to R31
		`trap`	Halt execution / transfer to operating system	
PAX Extensions	PTLU	`ptr.x.n, ptr,s.n`		Read w/8 tables in parallel and combine them according to the subop specified (Section 4.1)
		`ptrm.x.n, ptrm.s.n`		Read w/8 tables in parallel, mask the results then combine them according to the subop specified (Section 4.1)
		`ptw.n`		Write a different 32-bit entry in every 4th table (Section 4.2)
		`pti`		Write a 32-bit entry in all tables (Section 4.2)
	Permute	`byteperm`		Permute bytes in Rs1 using indices in Rs2 (Section 4.3)
		`rev`		Reverse the order of bits in Rs1 (Section 6.2)
		`shuffle.lo, shuffle.hi`		Shuffle bits in Rs1 and Rs2 (Section 6.3)
	Binary Field Multiply	`bfmul.lo`	Rd ← Rs1 $\otimes$ Rs2	Multiply binary polynomials in Rs1 and Rs2, and write the left word of the product to Rd (Section 6.1).
		`bfmul.hi`	Rd ← Rs1 $\otimes$ Rs2	Multiply binary polynomials in Rs1 and Rs2, and write the right word of the product to Rd (Section 6.1).

$w \in \{32, 64, 128\}$ is the wordsize. Rd is the destination register; Rs1, Rs2 are source registers. Imm is the immediate field supplied in the instruction. PC is program counter. MEM is memory.

The ciphers in Table 2 are chosen from widely-used network security protocols. Data Encryption Standard (DES) and its variant 3DES [2] were the NIST standards for block encryption from

1976 to 2001. They are used, for example, in the IPSec, TLS, and WTLS standards [1][11]. RC4 is a popular stream cipher developed in 1987 by Rivest [2]. It is used in the IEEE 802.11 wireless LAN standard [12]. Blowfish [2] was designed in 1994 by Schneier and is used in numerous protocols and commercial applications, for example GPG, SSH, SSLeay, JAVA cryptography extensions, and TiVo digital video recorders [13]. Advanced Encryption Standard (AES) [5] is the current NIST standard for block encryption. It was selected in 2001 at the end of a three-year AES development effort [14]. Key size of AES can be 128, 192, or 256 bits. We denote these AES-128, AES-192, and AES-256 respectively. Twofish [15] and MARS [16] are two of the five finalist ciphers in the AES effort [14]. Together with AES, these relatively new ciphers can be said to represent trends in symmetric-key cipher design.

TABLE 2: SYMMETRIC-KEY CIPHER SUITE

Cipher	**Block Size** (bits)	**Key Size** (bits)	**Number of Rounds**	**Number of Tables**	**Table Structure**	**Number of Lookups**
DES	64	56	16	8	$2^6 \times 32$	128
3DES	64	112	48	8	$2^6 \times 32$	384
RC4	8	128	1*	1	$2^8 \times 8$	3 reads, 2 writes
Blowfish	64	128	16	4	$2^8 \times 32$	64
AES-128	128	128	10	4	$2^8 \times 32$	160
AES-192	128	192	12	4	$2^8 \times 32$	192
AES-256	128	256	14	4	$2^8 \times 32$	224
Twofish	128	128	16	4	$2^8 \times 32$	128
MARS	128	128	32	2	$2^8 \times 32$	80

* RC4 does not have an iterated round structure, hence we show the number of rounds as 1.

A. AES and DES rounds

AES-128 is an iteration of 10 rounds after a first XOR operation between the plaintext block and the secret key. A round is made of four operations [5]: SubBytes, ShiftRows, MixColumns and AddRoundKey, except the last round of the data path which does not include the MixColumns transformation. To illustrate how a round operation of AES is typically optimized and implemented in software, we show the AES implementation using table lookups [17] in Figure 2. The input to the ith round is a 128-bit block composed of four 32-bit words, labeled $W3^i$-$W0^i$. The bytes in these words are indexed b0 to b15. TA-TD represent four $2^8 \times 32$ tables.

We use the notation $2^a \times b$ to denote a table with 2^a entries, where each entry is b-bits wide. During the round, the rightmost byte of each word is used as index into TA; the next byte is used as index into TB; and so on, until all tables are accessed four times. Finally, the four table lookup results (for each input word) are rotated and exclusive-or'ed (XORed) together and also XORed with a round subkey. This rotation is seen in Figure 2 as selection from four different columns of the table lookup results for the XOR function of each result word.

For DES (Figure 3), the round input is a 64-bit block, split into its left (WL) and right (WR) halves. TA-TH denote eight $2^6 \times 32$ tables. After WR is expanded into eight bytes, the leftmost six bits in each byte are used as an index into one of the tables, for a total of eight lookups.

The other ciphers in Table 2 also use table lookups. Blowfish, MARS, and Twofish are similar to AES in using multiple $2^8 \times 32$ tables, whereas RC4 uses a single $2^8 \times 8$ table. The last three columns of Table 2 show the number and structure of the lookup tables used by each cipher.

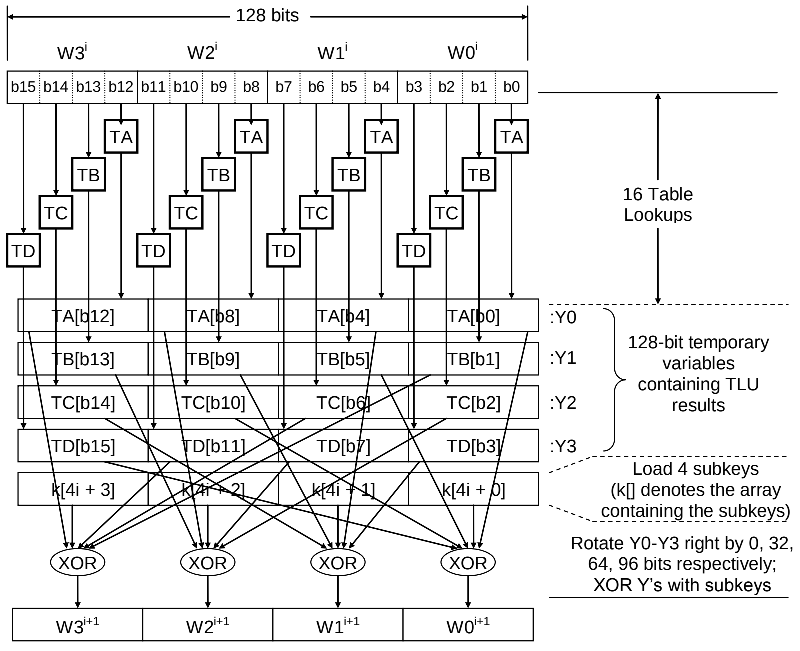

128 bits
W3^i
W2^i
W1^i
W0^i
b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
TA
TB
TC
TD
16 Table Lookups
TA[b12] TA[b8] TA[b4] TA[b0] :Y0
TB[b13] TB[b9] TB[b5] TB[b1] :Y1
TC[b14] TC[b10] TC[b6] TC[b2] :Y2
TD[b15] TD[b11] TD[b7] TD[b3] :Y3
128-bit temporary variables containing TLU results
k[4i + 3] k[4i + 2] k[4i + 1] k[4i + 0]
Load 4 subkeys (k[] denotes the array containing the subkeys)
XOR
Rotate Y0-Y3 right by 0, 32, 64, 96 bits respectively; XOR Y's with subkeys
W3^i+1 W2^i+1 W1^i+1 W0^i+1

Figure 2: AES round

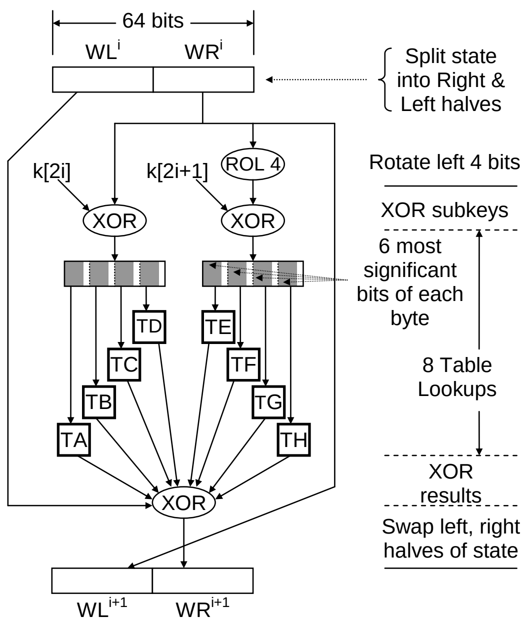


Figure 3: DES round

B. Execution time analysis

For baseline performance data, we implement and optimize the ciphers using the Base ISA in Table 1, which excludes the PAX-specific instructions. We use the PLX toolset [18][19] to simulate and profile each cipher. The simulator is configured to model a 64-bit single-issue processor similar to Figure 1 but excluding the binary field multiplier and the PTLU module. We assume that all instructions (including loads and stores) execute in a single cycle. Table 3 shows the simulation results, which includes: (a) the execution cycles used per block of encryption, (b) the round operations in each cipher, and (c) the fraction of the execution time consumed by these. Our data presented so far enable us to make the following observations:

- Table lookups consume the greatest fraction of the execution time for all ciphers, varying from 34% for MARS to 72% for AES (Table 3). Tables are few (at most eight) and have constant size (Table 2). Except for RC4, all table accesses are reads. Number of entries per table is small (at most 256) and the data read is either 8 or 32 bits (Table 2).
- The round structures of the ciphers generally permit the table lookups to be parallelized. For example, all 16 lookups in an AES round (Figure 2) or all 8 lookups in a DES round (Figure 3) can be performed in parallel, constrained only by hardware resources.

Table 3: Analysis of symmetric-key cipher execution time

		DES	3DES	RC4	Blowfish	AES-128	AES-192	AES-256	Twofish	MARS
	Block size (bits)	64	64	8	64	128	128	128	128	128
	Cycles per block of encryption	1147	3384	18	408	870	1056	1272	1753	1677
% Execution Cycles Spent in …	Table Lookups	38	44	54	36	72	72	72	43	34
	Arithmetic	-	-	14	26	-	-	-	15	10
	Logical	21	24	26	34	24	24	24	32	18
	Multiply	-	-	-	-	-	-	-	-	19
	Fixed shift/rotate	8	9	-	-	-	-	-	4	5
	Variable rotate	-	-	-	-	-	-	-	-	8
	Bit permutation	26	15	-	-	-	-	-	-	-
	Other	7	8	6	4	4	4	4	6	6

IV. Parallel Table Lookup (PTLU) Module

We propose a Parallel Table Look-Up (PTLU) module to accelerate the table lookups commonly used in symmetric key ciphers. The PTLU module consists of *w*/8 small blocks of memory that can be read in parallel, where *w* is the wordsize of the processor. A PTLU instruction reads two source registers and writes one result register, using the register datapaths already present for the other functional units (Figure 1). Hence, it looks like a functional unit rather than a memory module.

Figure 4 shows the details of the PTLU-64 module in PAX-64. There are eight tables with 256 entries each, where each entry is at most *w* bits wide. For PAX, we implement each entry as 32 bits, since this is the widest table entry needed in the cipher suite.[1] During a read, each table is accessed by an 8-bit index from the first source register Rs1. The rightmost byte of Rs1 accesses T0; the next byte accesses T1; and so on. All eight tables can be read in parallel.

The eight 32-bit lookup results, one from each table, are then routed through a combinatorial tree of XOR-Multiplexers (XMUXs) to produce a single result to be written to the destination register Rd. We describe one definition of this XMUX tree for PAX in Table 4 – many other useful definitions are possible.

For a *parallel table read* (`ptr`) instruction, the first two layers of XMUXs, labeled A_XMUX and B_XMUX, each allow selection of the Left (L) or Right (R) input, or an XOR of two inputs, based on the values of two control bits (C1,C0) as shown in the first two rows of Table 4. In addition, the A_XMUXs allow masking with one or two masks, M0 and M1, to be performed first on its two inputs, for the *parallel table read masked* (`ptrm`) instruction, as shown in the third row of Table 4. The final XMUX, labeled XMUX_64 is only a 2:1 MUX, enabling either an XOR of its two inputs or their concatenation, as shown in the last row of Table 4.

Although parallel table lookup requires only one operand to supply indices into the 8 parallel tables, the datapath of most processors allow two source register operands (Figure 1). We use this second source register, Rs2, as an extra input to be combined with the other table results in the XMUX tree (shown as the final XOR in Figure 4), or as an extra input to supply mask bits for the table lookup results, before they are combined in the XMUX tree. This allows only part of a table entry to be selected, and combined with parts of other table entries.

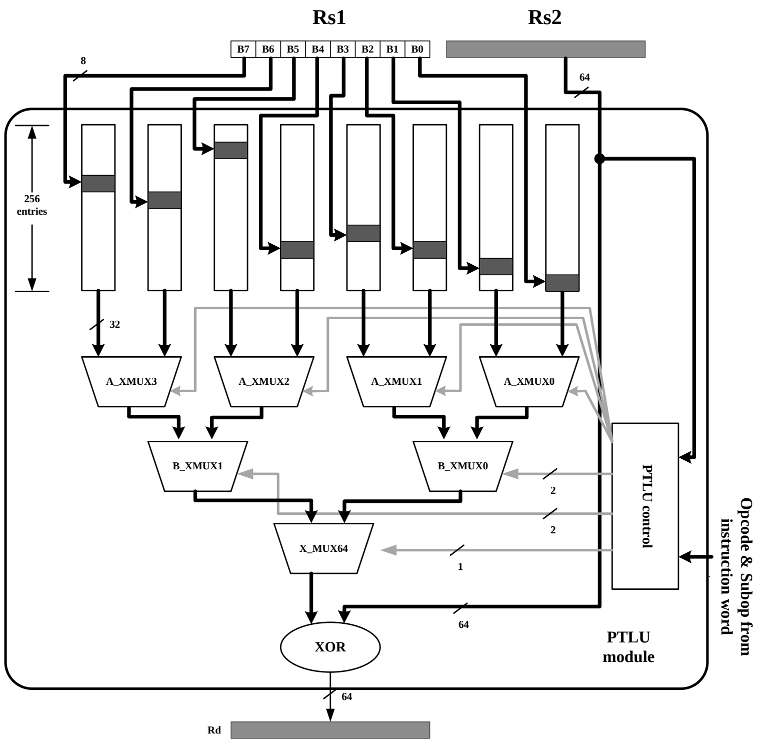


Figure 4: Reading the PTLU-64 module

TABLE 4: OPERATION PERFORMED BY THE XMUXS

		(C0, C1) Value			
		(0, 0)	**(0, 1)**	**(1,0)**	**(1, 1)**
B_XMUX	ptr	L	L XOR R	0	R
A_XMUX	ptr	L	L XOR R	0	R
	ptrm	L & M1	(L & M1) XOR (R & M0)	0	R&M1
XMUX_64		L ‖ R	L ‖ R	0 ‖ L XOR R	0 ‖ L XOR R

[1] In co-designed embedded systems, the number and/or the width of the tables can be scaled down to limit cost and power. Similarly, wider tables may be implemented for higher performance.

By using this XMUX tree, it is possible to realize many common operations that the symmetric-key ciphers perform on table data. For example, any one of the eight lookup results can be selected and written to Rd after being optionally XORed with another value supplied via Rs2. Another possibility is to XOR all eight lookup results. This is very useful for ciphers that XOR the results of multiple table lookups, such as AES (Figure 2) and DES (Figure 3).By allowing concatenation of two 32-bit values (at XMUX_64), we can also achieve two $2^8 \times 32$ table lookups in parallel. Below, we present instructions that we defined for PTLU; however the flexibility of the XMUX tree control allows for a larger set of instructions.

A. *Instructions for reading the PTLU module*

We define two ptr (parallel table read) instructions to read the PTLU module:

ptr.x.n Rd, Rs1, Rs2

ptr.s.n Rd, Rs1, Rs2

In each case, Rd is the destination register; Rs1 is the first source register, which supplies the byte-sized table indices; and Rs2 is the second source register, which is fed to the XOR gate which terminates the XOR tree (Figure 4). This last XOR operation with Rs2 can be easily discarded, by setting Rs2 to R0 (hardwired to zero).

In the first ptr.x.n instruction, the 'x' is a subop signifying 'XOR' -- this instruction is used to XOR multiple table lookup results. The 'n' in the mnemonic signifies the number of consecutive table look-up outputs which are XORed together; n can take three different values: 4, 8 and 16. For PAX-32, there are at most 4 parallel tables, so the only allowed value of n is 4. For PAX-64, there are at most 8 parallel tables, and n can take the values 4 or 8. For PAX-128, there are at most 16 parallel tables, and n can takes the values of 4, 8 or 16.

In the second ptr.s.n instruction, the subop 's' signifies 'select' because this instruction can select and write one of the table lookup results to Rd, after optionally `XOR`ing it with Rs2. The sub-opcode ‘n’ specifies the table result to be written to register Rd.

In this paper, we also define for the first time, two novel `ptrm` (parallel table read *masked*) instructions, which are identical to the above, except that masking of the table lookup results is performed in the first stage of the XMUX tree:

`ptrm.x.n` Rd, Rs1, Rs2

`ptrm.s.n` Rd, Rs1, Rs2

Two masks, M0 and M1, are generated from Rs2, where each bit of Rs2 is expanded into 8 bits to mask a byte of table result. Note that PTLU-*w* will have at most *w*/8 tables with 4-byte entries, hence a mask is (*w*/8)*4 = *w*/2 bits long. Since Rs2 is *w* bits, it can supply two masks, M0 and M1, which are used by the A_XMUXs in the `ptrm` instructions.

These masked versions are useful in many ciphers where some rounds differ from others. For example, in AES, the last round does not implement the MixColumns operation. This results in selecting a byte (instead of all four bytes) in each look-up table output. The masking we have defined is general-purpose and is also useful in other applications, not just for AES.

Concurrent processing of different algorithms which use the parallel lookup tables, without the need for re-loading tables, can be facilitated by using multiple sets of tables [20]. This is accomplished via an additional sub-opcode that specifies which set of tables the `ptr` instructions address. The tradeoff for this performance is the extra area required for each extra set of tables.

B. Instructions for writing the PTLU module

To write the tables in the PTLU module, we define the `ptw` (parallel table write) instruction:

`ptw.n` Rs1, Rs2

The current implementation of PAX considers tables with 32-bit entries, thus a 64-bit register can bring two table entries. Figure 5 depicts how the `ptw` instruction works in PAX-64. The 2-bit sub-op field, `n`, specifies which PTLU tables are written. If `n`=i, the two tables identified by `i` and `i+4` are written, for i=0, 1, 2 or 3, using the index given by the rightmost byte of Rs1. Hence, T0 and T4 are written together, T1 and T5 are written together, and so forth. Rs2 supplies the value to be written to the selected entry of the two tables.

The time taken to write the tables does not degrade cipher performance since writing tables is not needed during encryption or decryption (except for RC4 which uses a single table). However, fast parallel writes may be desired for rapid initialization of tables at setup time. For this, a parallel table initializes instruction, `pti`, can read an entire cache line from memory and write it to a common row of all eight PTLU tables in parallel. All entries of the PTLU module can be written using 256 such `pti` instructions.

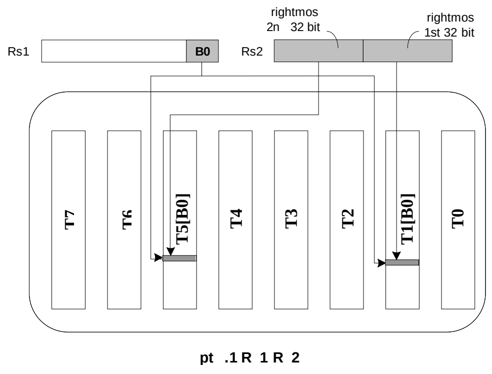


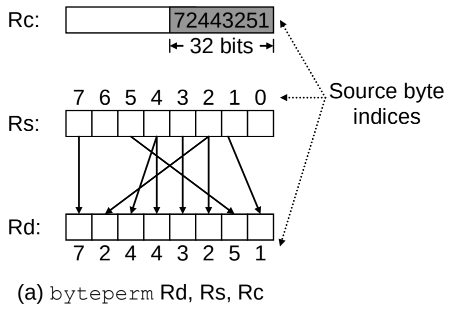
Figure 5: the `ptw` instruction

C. *Instructions for rearranging index bytes*

In the `ptr` instruction, the position of the index byte in Rs1 selects the table that is read. For example, the rightmost byte of Rs1 reads an entry from T0, the next byte reads an entry from T1, and so on. While this reduces the number of bits required to encode the instruction, it also restricts the types of table lookups that can be performed. For example, `ptr` cannot be used if the bytes in Rs1 need to access T0-T7 in a different order. To overcome this, we enable `ptr` to perform a much wider variety of table lookups by also defining a byte permutation instruction that can perform any permutation of the bytes in a source register:

`byteperm` Rd, Rs, Rc

Here, Rs supplies the eight bytes to be permuted and Rc contains the bits that specify the permutation. Figure 6(a) shows an example. The bytes in Rs are indexed from 0 to 7, the rightmost being byte 0. The 32 right-aligned bits in Rc specify the order in which the source bytes are written to Rd; the rightmost nibble in Rc selects the source byte to be written to the rightmost byte of Rd, and so on. The leftmost 32 bits of Rc are unused. (Since there are only 8 bytes in Rs, the most significant bit of each 4-bit nibble is always “0” for PAX-64.) This is similar to the `permute` instruction in MAX-2 [21] and the `pperm` instruction in [22].



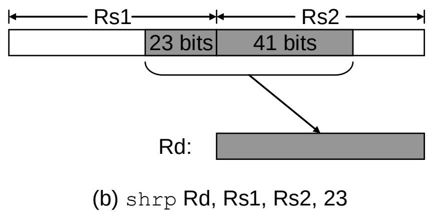


Figure 6: Examples of `byteperm` and `shrp`

To permute bytes in more than one register, `byteperm` can be used together with the shift right pair (`shrp`) instruction, which is shown in Figure 6(b). For example, any arbitrary permutation of sixteen bytes packed in two 64-bit registers can be performed using at most four `byteperm` and two `shrp` instructions. An example of this is given in the next section.

`Byteperm` can be implemented in hardware using eight 8-to-1 multiplexers (each 8-bit-wide). In PAX, we implement `byteperm` in a modified shifter.

D. Optimized AES

Using PTLU-128, an AES-128 block encryption can be done in just 22 cycles in software.

Figure 7(a) shows that a round, for the first 9 rounds, takes just 2 instructions each, using `byteperm` followed by `ptr`. The last round takes 3 instructions: `byteperm`, `ptrm` and `XOR`.

Using PTLU-64, each round takes 10 cycles as shown in Figure 7(b) and illustrated in Figure 8. The 128-bit AES state is supplied in two 64-bit registers (R16, R17). The first six instructions permute (R16, R17) such that R21 and R22 each contain eight indices into tables whose results can be directly `XOR`'ed together. The load.8 instruction loads the first half of the round subkey into R15. The following `ptr.x.4` instruction performs eight lookups using the bytes in R21. These results are `XOR`ed in pairs by A_XMUXs and B_XMUXs. XMUX_64 concatenates the outputs of B_XMUX0 and B_XMUX1, and `XOR`s the result with the round subkey contained in R20. The destination register R16 then contains ($W1^{i+1}$,$W0^{i+1}$). Similarly, the next `ptr.x.4` instruction computes ($W3^{i+1}$,$W2^{i+1}$).

The last AES round is 12 instructions: the `ptr.x.4` instructions are replaced by `ptrm.x.4` instructions, followed by two `XOR`s with the round sub-key. Hence, the total for AES using PTLU-64 is 2+90+12= 104 instructions.

V. Workload Characterization of Public-Key Cryptography

Public-key ciphers derive their cryptographic strength from hard mathematical problems such as the discrete logarithm problem (DLP), the factoring problem (FP), and the elliptic-curve discrete logarithm problem (ECDLP) [2][3][23][24]. Diffie-Hellman (DH) and the Digital Signature Algorithm (DSA) are DLP-based ciphers, while RSA is a FP-based cipher [2][3][6]. The Elliptic Curve Discrete Logarithm Problem, proposed for cryptographic use by Koblitz [23] and Miller [24] independently in 1985, is significantly harder than the discrete logarithm and factoring problems. Hence, ECDLP-based ciphers can use shorter keys while providing the same security as DLP or FP-based ciphers. The ciphers based on ECDLP are collectively called Elliptic Curve Cryptography (ECC).

```
; --------------- AES round operation using PTLU-128 ---------------------
; round input 1 word is stored in r17

; convert state bytes from:  r17 = b15  b14  b13  b12  | b11  b10  b9   b8   | b7   b6   b5   b4 | b3   b2   b1  b0
; to the new order of:       r17 = b15  b10  b5   b0   | b11  b6   b1   b12  | b7   b2   b13  b8 | b3   b14  b9  b4

  byteperm   r17, r17, r6    ; permute bytes in r17 with the ordering of indices specified in r6

  ptr.x.4       r17, r17, r2x  ; lookup 16 tables, XOR the results with round subkey (key for round x stored in register r2x);
                             ; store the round output W0 into r17

; round output 1 word is stored in r17, new AES state
```

(a)

```
; --------------- AES round operation  using PTLU-64  ----------------------
; round input 2 words are stored in r17, r16
; ordering of indices for byte permutation stored in  r5 and  r6

; convert state bytes from:  r17 = b15  b14  b13  b12  | b11  b10  b9   b8   |
;                            r16 = b7   b6   b5   b4   | b3   b2   b1   b0   |
; to the new order of:       r22 = b15  b10  b5   b0   | b11  b6   b1   b12  |
;                            r21 = b7   b2   b13  b8   | b3   b14  b9   b4   |

  byteperm r23, r16, r5      ; r23 = b6   b5   b1   b0   b7   b4   b3   b2  - group bytes for shrp
  byteperm r24, r17, r5      ; r24 = b14  b13  b9   b8   b15  b12  b11  b10 - group bytes for shrp
  shrp r21, r23, r24, #32    ; r21 = b7   b4   b3   b2   b14  b13  b9   b8  - collect bytes in one reg.
  shrp r22, r24, r23, #32    ; r22 = b15  b12  b11  b10  b6   b5   b1   b0  - collect bytes in one reg.
  byteperm r21, r21, r6      ; r21 = b7   b2   b13  b8   b3   b14  b9   b4  -  permute bytes in one reg.
  byteperm r22, r22, r6      ; r22 = b15  b10  b5   b0   b11  b6   b1   b12 - permute bytes in one reg.

; parallel table lookup to generate round output for each word
  load r20, r15, #2          ; load W0 of round subkey from memory - not  enough registers to keep subkeys between rounds
  ptr.x.4 r16, r21, r20      ; lookup 8 tables, XOR the results with round subkey;
                             ; store the round output W0 into r16

  load r20, r15, #3          ; load W1 of round subkey from memory
  ptr.x.4 r17, r22, r20      ; lookup 8 tables, XOR the results with round subkey;
                             ; store the round output W1 into r17

; round output 2 words are stored in r17, r16, new AES state
```

(b)

Figure 7: Optimized AES round with `ptr.x.4` using (a) PAX-128, (b) PAX-64

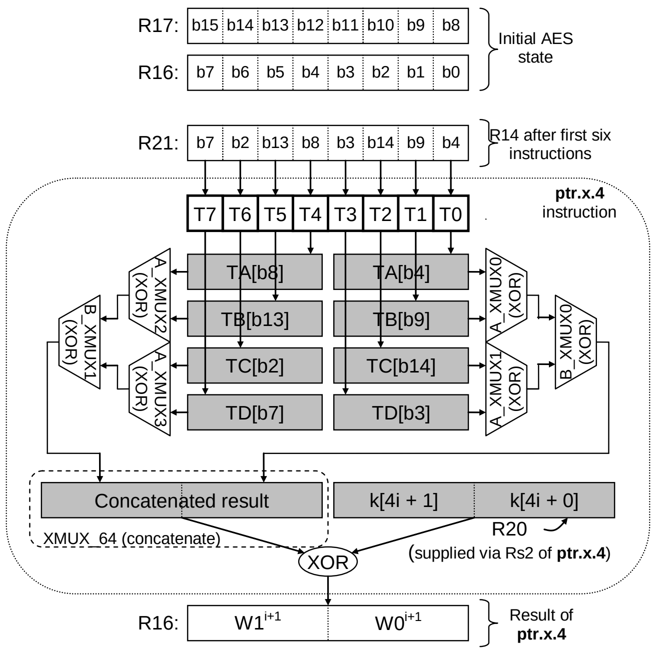


Figure 8: Illustrating the data flow in an AES round (first 8 lookups) using PTLU-64

Table 5 shows that ECC with 160-bit keys offers security equivalent to 1024-bit RSA or DSA [26], which is a 6.4× reduction in key size. This differential continues to widen in favor of ECC as the key size increases. The smaller parameters in ECC translate into savings in computation time, processing power, storage space, bandwidth, and power consumption [27][28]. This makes ECC particularly suitable for resource-constrained mobile devices. ECC has been adopted by major security standards such as ANSI X9.62 [29], FIPS 186-2 [6], IEEE P1363 [30], and ISO 14888-3 [31].

TABLE 5: EQUIVALENT CIPHER STRENGTHS

DLP or FP-based Public-Key Cipher (e.g. DSA, RSA)	ECDLP-based Public-Key Cipher (ECC) (e.g. eDSA)	Reduction in Key Size with ECC
1024	160	6.4
2048	224	9.1
3072	256	12.0
7680	384	20.0
15360	512	30.0

Since PAX is targeted for mobile devices, we optimize it for ciphers most likely to be used in constrained environments, such as ECC. Our cipher suite therefore consists of three representative ECC algorithms: Elliptic Curve Diffie-Hellman (eDH), Elliptic Curve ElGamal (eElGamal), and Elliptic Curve

Digital Signature Algorithm (eDSA). These are used in mamy security protocols and standards such as SSL/TLS, WTLS, SSH, and DSS [1][6][11]

A. ECC operations

ECC uses an operation called point multiplication, where a point $P = (x_0, y_0)$ on an elliptic curve is multiplied by a scalar k [27]. The result of this operation is Q, which is another point on the elliptic curve: $k \times P = Q = (x_1, y_1)$. Due to the ECDLP, it is computationally infeasible to compute k given only P and Q [23][24]. This one-way property provides the security of ECC.

On word-oriented programmable processors, ECC can be most efficiently implemented using binary finite fields, denoted GF(2^m) [28][32]. The elements of GF(2^m) can be represented as binary polynomials of degree at most m-1 and with coefficients from {0, 1}. For example 163-bit ECC can be defined over GF(2^{163}) [6], where the coordinates of the curve points will be binary polynomials of the form:

$$a(x) = a_{162}x^{162} + \ldots + a_2x^2 + a_1x + a_0, \quad \text{where } a_i \in \{0, 1\} \text{ for } i = 0, 1, \ldots, 162.$$

The point multiplication operation is then realized by multiple arithmetic operations on these polynomials [33]. This includes polynomial addition, multiplication, reduction, squaring, and inversion.

Since the operations on polynomial coefficients are performed modulo 2, polynomial addition can be simply performed by XORing the operands. For the remaining polynomial operations, there is a wide variety of optimized algorithms that can be used; Hankerson et al. [32] give a comprehensive survey. In our simulations, we use the fastest methods from this study. These are: *comb* method for polynomial multiplication, *table-lookup* method for polynomial squaring, *word-based* method for polynomial reduction, and the *modified almost inverse algorithm* for polynomial inversion [34].

B. Execution Time Analysis

We implement ECC ciphers with 163-bit, 233-bit, and 283-bit keys specified in FIPS 186-2 [6]. This provides security comparable to DLP-based ciphers with 1024-bit to 3072-bit keys (Table 5), considered by NIST to be adequate until at least the year 2016 [26]. The ciphers are implemented in C and simulated with the Simplescalar toolset configured for the PISA instruction set [35]. The architectural parameters of the baseline processor are given in Table 6.

TABLE 6: BASELINE PROCESSOR

Architectural Parameter	Value
Issue-width	In-order single-issue
Number of load/store pipes	1
L1 I-Cache	64 kB, 2-way, 32 B lines
L1 D-Cache	64 kB, 2-way, 32 B lines
L2 Cache (unified)	1 MB, 4-way, 64 B lines
L1 latency	1 cycles
L2 latency	10 cycle
Memory latency	100 cycle

Table 7 indicates that point multiplication is the dominant operation in ECC. Its fraction of the execution time for 163-bit ECC ranges from 94.1% for eDSA to 99.1% for eDH. Because point multiplication constitutes the bulk of the execution cycles for all ciphers, it is generally used as a proxy to measure overall ECC performance [32][33].

TABLE 7: EXECUTION CYCLES FOR ECC CIPHERS

Key Size (bits)	Cipher	Cycles (×106)	% Cycles Consumed by Point Multiplication
163	eDH	20.342	99.1
	eElGamal encrypt	30.629	98.0
	eElGamal decrypt	20.277	97.4
	eDSA sign	10.813	94.1
	eDSA verify	21.590	97.2
233	eDH	43.125	99.2
	eElGamal encrypt	65.427	98.1
	eElGamal decrypt	44.197	97.6
	eDSA sign	23.583	94.5
	eDSA verify	46.314	97.6
283	eDH	65.867	99.4
	eElGamal encrypt	98.457	98.2
	eElGamal decrypt	65.332	97.7
	eDSA sign	34.839	95.0
	eDSA verify	71.063	98.1

TABLE 8: POLYNOMIAL OPERATIONS IN POINT MULTIPLICATION

		Per Point Multiplication		
Key Size (bits)	Polynomial Operation	Number of Calls	Cycles (×106)	% of Total Cycles
163	Multiplication	975.95	9.064	87.25
	Squaring	807.96	0.657	6.33
	Inversion	1.00	0.157	1.51
	Other	N/A	0.511	4.91
233	Multiplication	1408.35	18.999	88.62
	Squaring	1172.48	1.296	6.05
	Inversion	1.00	0.281	1.31
	Other	N/A	0.864	4.02
283	Multiplication	1703.38	29.896	90.22
	Squaring	1424.65	1.754	5.29
	Inversion	1.00	0.362	1.09
	Other	N/A	1.124	3.39

Table 8 shows that point multiplication consists of three major polynomial operations: multiplication, squaring, and inversion. Polynomial multiplication is the dominant operation, which consumes 87.25% of the point multiplication time for 163-bit ECC. This is followed by polynomial squaring (6.33%) and polynomial inversion (1.51%).

For the polynomial operations, addition is the simplest, followed by squaring, multiplication, and inversion (Table 9). Inversion is the most costly, with a complexity of 15 to 20 times multiplication.

Hence, ECC execution time is dominated by polynomial arithmetic. Table 7 and Table 8 show that more than 90% of the execution time of 163-bit ECC is consumed by two polynomial operations: multiplication and squaring.

TABLE 9: EXECUTION CYCLES FOR POLYNOMIAL OPERATIONS

	Polynomial Size (= ECC key size, bits)		
Operation	**163**	**233**	**283**
Addition	7	9	11
Reduction	476	644	707
Squaring excl. reduction	287	426	527
Squaring incl. reduction	791	1098	1228
Multiplication excl. reduction	8667	12971	16932
Multiplication incl. reduction	9199	13498	17512
Inversion	149836	270477	359102

VI. PAX FEATURES FOR FAST ARITHMETIC ON GF(2^M)

A. Polynomial multiplication

Polynomial multiplication in ECC constitutes up to 90% of the execution cycles (Table 8). This is primarily because a standard integer multiplier cannot be used to multiply two binary polynomials. Instead, ALU and shift instructions are used, which require thousands of execution cycles to compute the product.

With minor changes, a standard integer multiplier can be converted into a dual-field multiplier to also multiply binary polynomials [36][37]. However, this is a large, multi-cycle functional unit. A binary-field multiplier is much smaller and faster than a dual-field multiplier. Omitting the integer multiplier is not likely to hurt cryptographic performance since neither symmetric-key ciphers (Section III) nor ECC requires integer multiplication.

A binary-field multiplier can be implemented as an array of AND gates followed by an XOR-tree

(Figure 9). Unlike an integer multiplier, the partial products are exclusive-or'ed (`XOR`ed) instead of added. Because `XOR` is much faster than integer addition, binary field multiplication can be completed in a single cycle. PAX includes two instructions to use the binary field multiplier: `bfmul.lo` and `bfmul.hi`, which are shown in Figure 10. In `bfmul.hi`, the binary polynomials supplied in two source registers are multiplied and the left word of the product is written to Rd. In `bfmul.lo`, the right word of the product is written to Rd.

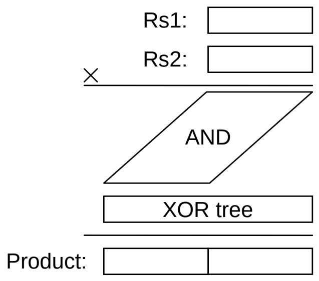


Figure 9: Binary field multiplier

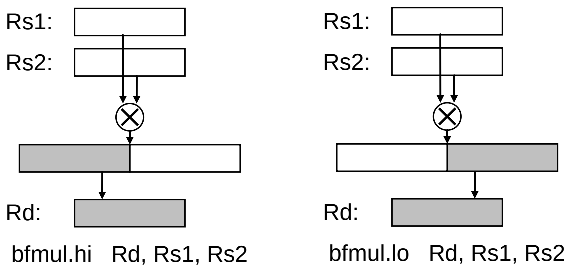


Figure 10: Binary field multiply instructions

A polynomial multiplication for ECC on PAX-128 requires multiple instructions to complete. The software splits each 163-bit operand into two 128-bit words, with zero extension to the higher unused bits. It performs 4 `bfmul.lo`, 4 `bfmul.hi` and 4 `xor` instructions, for a total of 12 instructions, to complete the multiplication of two 163-bit operands. The same 163-bit multiplication takes 22 instructions on PAX-64 and 97 instructions on PAX-32.

B. Area saving for polynomial multiplication

The left and right halves of a binary-field multiplier are nearly symmetric (Figure 9). In [38], we have shown that it is possible to compute the left word of the product by using only the right half of the multiplier, thereby allowing the circuit size to be reduced by about half. A `bfmul.hi` instruction can be realized by: (1) reversing the order of bits in the operands, (2) multiplying the reversed operands using `bfmul.lo`, (3) shifting the product left by one bit, and (4) reversing the order of bits of the result.

C. Shuffle instructions for polynomial squaring

Polynomial squaring involves multiplying a binary polynomial with itself. This can be implemented using a multiplication algorithm, such as the comb method in [32], or it could be optimized by exploiting the linearity of the squaring operation on binary fields. For example, let *a*(*x*) be a binary polynomial used with 163-bit ECC [6]. If we represent *a*(*x*) as $\sum_{i=0}^{162} a_i x^i$, where $a_i \in \{0, 1\}$, then the square of a(x) is $a^2(x) = \sum_{i=0}^{162} a_i x^{2i}$. The result contains no cross product terms since these appear in pairs and vanish when the coefficients are added modulo 2. Hence, the square of a binary polynomial contains the same coefficients as the input, but these appear in front of x terms with twice the original degrees. This corresponds to interleaving the bits of the input polynomial with zeros. For example, if $a(x) = x^3 + x + 1$, then $a^2(x) = x^6 + x^2 + 1$. In vector representation: $a = (1011)$ and $a^2 = (1000101)$.

PAX includes bit shuffle instructions to accelerate this computation. As shown in Figure 11, a shuffle instruction reads bits alternating between two source registers, and writes these to a destination register. Because Rd can accommodate only half of the bits in Rs1 and Rs2, two versions of the instruction are defined to shuffle either the left or the right halves of the source registers, called `shuffle.hi` and `shuffle.lo` respectively. The implementation cost is low, involving only routing of the source bits.

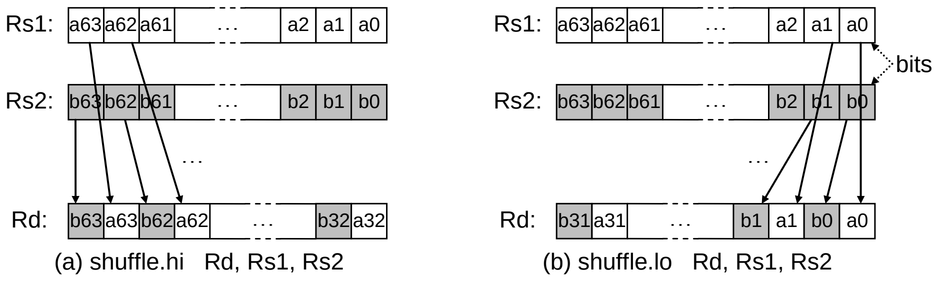


Figure 11: 64-bit shuffle instruction

VII. PAX implementation: Area, Latency and Cycle Time Analysis

A. PAX hardware implementation

We implemented PAX in VHDL with a classic 5-stage pipeline: Instruction Fetch (IF), Decode (D), EXecute (EX), Memory (M) and Write Back (WB), with hazard detection and forwarding. Table 10 shows how the basic PAX instructions are implemented in the pipeline. The implementation is wordsize-scalable. The wordsize (*ws*) indicates register size and data-path width and can be 32, 64 or 128 bits. The choice of *ws* is made by the designer before synthesis. RTL simulation of the design using the Mentor Graphics Modelsim toolset was done for PAX-32, PAX-64 and PAX-128. Simulation results confirm that the 10 rounds of AES-128 can be done in 22, 104 and 248 cycles for PAX-32, PAX-64 and PAX-128 processors, respectively.

TABLE 10: INSTRUCTION EXECUTION IN PAX PIPELINE

`ALU, Shift`	IF | D | EX | - | WB
`Load`	IF | D | EX | M | WB
`Store`	IF | D | EX | M | -
`Branch`	IF | D | EX | - | -
`PTLU (ptr, ptrm, ptw)`	IF | D | EX | - | WB
`Byte-perm`	IF | D | EX | - | WB

B. Wordsize scaling and subword-parallelism

Some PAX instructions (Table 1) are independent of wordsize. These include logical instructions (`AND`, `OR`, `XOR`, etc.) which operate on pairs of bits, regardless of the word size, and control flow instructions.

Shift, multiply, and permute instructions operate on full words, hence their operand size changes with the wordsize. The binary-field multiplier multiplies two wordsize operands. `Byteperm` is slightly different since it always permutes byte-sized data, but the number of source bytes changes with wordsize, from 4 in PAX-32 to 16 in PAX-128.

For the PTLU module, the number of tables is limited by the number of bytes in a word used to index different tables. Hence, we implement 4 tables in PAX-32, 8 tables in PAX-64, and 16 tables in PAX-128 (although fewer tables can be implemented to limit cost and power). We keep the width of the tables at 32 bits regardless of wordsize. While the architecture permits implementing wider tables, this is not generally needed for the block ciphers.

For add and subtract instructions, it is sufficient to support only 32-bit operations for crypto-processing. PAX-64 can perform two parallel 32-bit adds in a cycle and PAX-128 can perform four parallel 32-bit adds in PAX-128. This is called *subword parallelism* and was introduced by Lee [40] for multimedia acceleration. Since the carry-lookahead logic is the critical path in an ALU, using smaller adders reduces latency. Subword parallelism for other instructions is not needed since this provides no performance benefits for crypto-processing [41].

C. Area, Delay and Cycle Time

For area and delay estimates, we performed gate-level synthesis of the functional units using Synopsys tools with a TSMC 90nm technology library. For the PTLU module, we used CACTI 5.0, a tool for estimating the access time, area, and aspect ratio of memory components [39]. Table 11 summarizes our results. For each functional unit, we report absolute area in square-microns and relative area normalized to the ALU (shown in parenthesis). Delay is given as absolute delay in nanoseconds and relative delay normalized to the ALU (shown in parenthesis). Separate functional unit information is given for 32-bit, 64-bit, and 128-bit processors.

Table 11 shows that implementing `byte_perm` and `shuffle` instructions in the modified PAX shifter (the shift-permute unit, SPU) does not increase the shifter latency.

The access time of the PTLU tables is greater than the ALU delay. However, the XMUX tree can always be synthesized so that the total delay through the PTLU module is no longer than twice the ALU latency. If the cycle time is equal to the ALU delay, the `ptr` instructions would have 2-cycle latency. However, the cycle time is more often equal to twice the ALU latency, due to other critical paths in the processor-cache subsystem, including bringing the cache access time down to 1 or 2 cycles (rather than 3-4 cycles from the numbers in Table 11). Hence, a PAX processor that includes a PTLU module would typically set the cycle time to that of the `ptr` instructions, so that all instructions are single-cycle instructions.

The PTLU tables comprise most of area (96-98%) of the PTLU module. While the PTLU module is large relative to the other functional units, it is small compared to the 32 kB to 256 kB caches typical in today's high-end embedded processors, e.g., Marvell PXA320 [42]. Compared to these, the size of the PTLU module is small; about 45% of the 32 kB cache and 5% of the 256 kB cache (PAX-64). Since the PTLU tables at most 8 KB with 8 parallel reads for PAX-64, we may also want to compare this to an 8 kB cache with eight read ports. PTLU has only 8% of the area of such a cache, and is much faster. Hence, the PTLU module provides much greater performance than these caches, and also saves significant area since a PAX processor may not need to implement data caches, or only much smaller ones.

TABLE 11: AREA AND DELAY OF BASELINE AND PAX FUNCTIONAL UNITS

	PAX-32		PAX-64		PAX-128	
Functional Unit / Component	Area (μ2)	Delay (ns)	Area (μ2)	Delay (ns)	Area (μ2)	Delay (ns)
ALU	4629 (1)	0.5 (1)	9635 (1)	0.55 (1)	20520 (1)	0.6 (1)
Standard barrel shifter	4850 (1.0)	0.5 (1)	12754 (1.3)	0.55 (1)	30603 (1.5)	0.6 (1)
PAX shifter (SPU)	6432 (1.4)	0.5 (1)	19130 (2.0)	0.55 (1)	49827 (2.4)	0.6 (1)
PAX binary field multiplier	10041 (2.2)	0.5 (1)	40020 (4.2)	0.55 (1)	163642 (8.0)	0.6 (1)
PTLU: Tables only	130814 (28.3)	0.66	261627 (27.2)	0.66	523254 (25.5)	0.66
PTLU total	134661 (29.1)	0.98 (1.96)	270641 (28.1)	1.05 (1.91)	542271 (26.4)	1.18 (1.97)
32 kB 2-way cache w/ 64-byte blocks	Same →		599104 (62.2)	1.20 (2.18)	← Same	
256 kB 2-way cache w/ 64-byte blocks			5224494 (542.2)	1.82 (3.31)		
8 kB direct-mapped cache w/ 64-byte blocks and 8 read ports			3255712 (337.9)	1.62 (2.95)		

The PAX multipliers are always single cycle units. This shows the advantage of using just binary-field multipliers rather than ones that also do integer multiplication, since the latter typically have at least a 3-cycle latency.

VIII. PERFORMANCE

A. Impact of new instructions

Table 12 shows the symmetric-key cipher speedups obtained with a PAX-64 processor that has the PTLU and `byteperm` instructions. The speedups are relative to the execution cycles needed per block of encryption with a 64-bit baseline processor, i.e., a single-issue processor that implements the Base ISA in Table 1. While all ciphers benefit from PTLU and `byteperm` instructions, some show huge performance gains. The average speedup for DES/3DES and AES are 5.4× and 8.1× respectively. The other ciphers have speedups varying from 1.2× for MARS to 2.8× for Twofish.

Table 13 shows the speedups obtained with the `bfmul` instructions for polynomial multiplication. The speedups are relative to multiplication using the comb method, which is identified in [32] as the fastest algorithm among the ones surveyed. When a full (word-sized) binary-field multiplier is used, the average speedup obtained for the three ECC key sizes [6] is 25.13×. When a half-sized multiplier is used with `rev` (reverse) instructions, the speedup averages 18.03×.

Table 12: Symmetric-key cipher speedup with PAX

Cipher	Block size (bits)	Execution Cycles with Base ISA (64-bit processor)	Speedup with PAX-64 (ptr + byte_perm)
DES	64	1147	5.41 ×
3DES	64	3384	5.32
RC4	8	18	2.00
Blowfish	64	408	1.66
AES-128	128	870	7.84
AES-192	128	1056	8.06
AES-256	128	1272	8.42
Twofish	128	1753	2.81
MARS	128	1677	1.23

Table 14 shows the speedups obtained with the shuffle instructions in polynomial squaring, which are relative to the table lookup method described in [32]. The average speedup is 3.86×.

TABLE 13: POLYNOMIAL MULTIPLICATION SPEEDUP WITH PAX

	Cycles	Speedup		
Polynomial size (=ECC key size) (bits)	Comb method	Comb method	PAX-32 bfmul.hi + bfmul.lo	PAX-32 bfmul.lo + rev
163	8667	×1.00×	24.85×	17.87×
233	12971	1.00	25.16	18.03
283	16932	1.00	25.38	18.18

TABLE 14: POLYNOMIAL SQUARING SPEEDUP WITH PAX

	Cycles	Speedup	
Polynomial size (=ECC key size) (bits)	Table lookup method	Table lookup method	PAX-32 shuffle.lo + shuffle.hi
163	287	×1.00×	3.78×
233	426	1.00	3.86
283	527	1.00	3.95

TABLE 15: ECC POINT MULTIPLICATION SPEEDUP WITH PAX

		Speedup					
ECC Key Size (bits)	Baseline Execution Cycles (×106)	Baseline	bfmul.lo + bfmul.hi	bfmul.lo + rev	shuffle.lo + shuffle.hi	PAX-32 bfmul.lo + bfmul.hi + shuffle.lo + shuffle.hi	PAX-32 bfmul.lo + rev + shuffle.lo + shuffle.hi
163	10.389	×1.00×	×6.06×	×5.59×	×1.05×	×20.25×	×16.01×
233	21.439	1.00	6.44	5.91	1.05	20.63	16.20
283	33.137	1.00	7.38	6.68	1.04	21.46	16.66
Average speedup		1.00	6.63	6.06	1.05	20.78	16.29

Table 15 shows the speedups in ECC point multiplication obtained with various combinations of `bfmul`, `shuffle`, and `rev` instructions. A full multiplier that implements both `bfmul.lo` and `bfmul.hi` instructions provides the highest speedups, averaging 6.63×. To save area and power, a half multiplier can be used that only implements the bfmul.lo instruction. This reduces the average speedup to 6.06×. We implement the full multiplier in our PAX implementation.

Using *only* the shuffle instructions provides insignificant performance improvement (5%) because the fraction of ECC execution time consumed by polynomial squaring is only 6%, whereas polynomial multiplication consumes up to 90% of the baseline execution cycles (Table 8). However, the usefulness of shuffle increases significantly when used together with `bfmul`. Once polynomial multiplication is accelerated by up to 25.38× with a binary field multiplier (Table 13), squaring becomes the new dominant operation in ECC. Accelerating squaring at this point using shuffle instructions increases the cumulative ECC speedups to an average of 20.78×.

B. Wordsize scaling versus Superscalar execution

We now show the performance benefits of the wordsize scalability feature in PAX, and compare this to ILP scaling used in superscalar processors. Since the two scaling techniques are orthogonal, they can be combined to achieve a specific performance-cost target.

For 3DES, AES-128, and ECC point multiplication, Table 16 shows the speedups obtained using Simplescalar with superscalar execution on processors with issue widths from 1 to 8. Speedups are relative to the single-issue 32-bit baseline processor. For DES and ECC, we observe that 2-way and 4-way superscalar execution with a single memory port provides significant speedups for all ciphers (up to 2.06×). Further increasing the issue width to 8 provides only minor additional performance (up to 2.18×). For AES, on the other hand, 8-way superscalar execution provides significant speedup (up to 4.00×). Adding a second memory port increases performance significantly at larger issue widths due to the memory-intensive round structures.

For the same ciphers, Table 17 shows the speedups obtained with the much simpler *single-issue* 32-bit, 64-bit, and 128-bit PAX processors. In PAX-32, we obtain speedups of 3.4×, 3.3×, 7.7× for 3DES, AES, ECC respectively. This should be compared with the single-issue 1/1 machine. For 3DES and ECC, they are even higher than the speedups obtained on an 8-way superscalar processor with two memory ports (Table 16).

In PAX-64, the AES speedup increases to 7.8×. This should be compared to the 1.97× speedup of the 2-way 32-bit processor since both have equivalent degrees of operand parallelism. Similarly, the 34.8× speedup of PAX-128 can be compared to the 3.8× speedup of the 4-way 32-bit processor. *These results clearly indicate that wordsize scaling with PAX is far more effective for improving performance than IPC scaling in superscalar processors.* Compared to a multi-issue processor, a wider single-issue processor saves on register ports, data buses, bypass paths, and instruction dispatch logic [40].

TABLE 16: BASE ISA PERFORMANCE WITH SUPERSCALAR IMPLEMENTATION

	Speedup with Superscalar Implementation (# issue width / # memory ports)						
Cipher	1/1	2/1	2/2	4/1	4/2	8/1	8/2
3DES	1.00×	1.62×	1.85×	1.78×	2.32×	1.88×	2.73×
AES-128	1.00×	1.97×	1.97×	3.49×	3.80×	4.00×	6.33×
ECC Point Multiplication	1.00×	1.64×	1.76×	2.06×	2.24×	2.18×	2.59×

Note: In the notation a/b, a is the issue width and b is the number of memory ports.

TABLE 17: PAX PERFORMANCE WITH WORDSIZE SCALING

	Single-issue Base ISA (32-bit)	Speedup with Single-issue PAX (ptr + byte_perm + bfmu.lo + bfmul.hi + shuffle.lo + shuffle.hi)		
Cipher		PAX-32	PAX-64	PAX-128
3DES	1.00×	3.41×	5.32×	5.32×
AES-128	1.00×	3.33×	7.84×	34.8×
ECC Point Multiplication	1.00×	7.68×	16.58 ×	24.88 ×

C. *Mobile wireless devices and servers*

Network security protocols usually initiate a secure session by using public-key ciphers and then encrypt any subsequent data using symmetric-key ciphers. For example, the WTLS protocol [11] has a handshake phase for authentication (public-key) and a record phase for bulk encryption (symmetric-key). PAX improves the performance in both WTLS phases.

In, Figure 12 we show the processor clock rate required to complete a client-side WTLS handshake (which includes a digital signature verification and other public-key operations) within a given latency. Assuming a 1 second authentication delay, the target clock rate reduces from 131 MHz for the baseline processor, to 22 MHz for PAX-32. Wordsize scaling to PAX-128 further decreases this to 4 MHz.

As 3G wireless multimedia phones and appliances start to proliferate [9], it is interesting to see what it takes to do cryptographic processing of bulk data at link speeds. Figure 13 shows the clock rates required to achieve a desired AES-128 throughput with different PAX processors. On the horizontal axis, we show the data rates of a few important wireless technologies [9]. To achieve 3G link speeds (2.4 Mbps), we only need a 5 MHz PAX-32 processor, a 2 MHz PAX-64 processor, or a .5 MHz PAX-128 processor. A basic RISC processor would need to run at 18 MHz, consuming much more power. To saturate an IEEE 802.11g connection, which has a 54 Mbps maximum data rate, the clock rate of the base processor needs to be 385 MHz, while PAX-32 only needs a 116 MHz clock, and PAX-128 only needs a 11 MHz clock. The clock rate of PAX-32 is approximately one seventh of the ~800 MHz rate used in today's high-end embedded processors [42]. This significantly reduces the energy consumption and area, conserving the battery and cost of resource-constrained wireless devices.

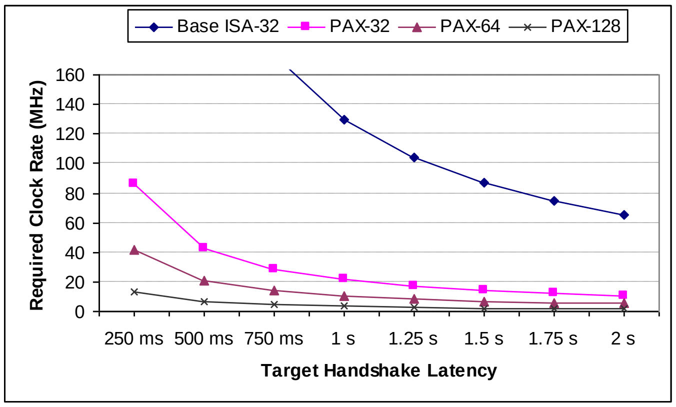


Figure 12: Required clock rate for given WTLS handshake latency

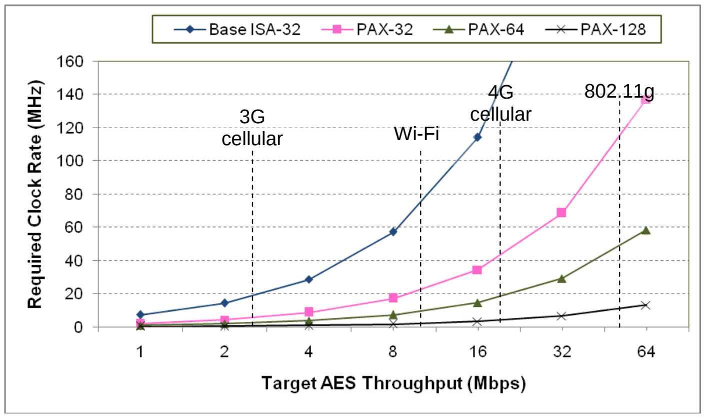


Figure 13: Required clock rate for given AES-128 throughput

The PAX-specific instructions, such as `ptr`, `bfmul`, and `shuffle`, can also be added to the instruction sets of general-purpose microprocessors to accelerate server-side crypto-processing. Table 18 shows how many server-side WTLS handshakes can be performed by a 1 GHz server within a given time period, with and without PAX extensions. (We show 1 GHz to allow easier extrapolation to other GHz rates.) To obtain these results, we used SimpleScalar to simulate a 32-bit. 8-way superscalar, out-of-order processor with 256kB/1MB L1/L2 caches. Our results indicate that a server can increase its authentication throughput by 3.83× by adding a few low-cost instructions to its instruction set.

TABLE 18: SERVER PERFORMANCE WITH PAX EXTENSIONS

	WTLS handshakes in 1 second	Times improvement
Baseline	173	1.00×
With PAX extensions	662	3.83×

IX. PAST WORK

Architectural enhancements to accelerate table lookups in symmetric-key ciphers have been proposed previously, e.g., the sbox instruction in [43] which performs fast lookups of tables in main memory by accelerating the effective address computations. The CryptoManiac processor [44] uses a similar sbox instruction to quickly access four 1 kB on-chip caches. However, unlike our PTLU module, both of these approaches read only a single table with each sbox instruction. To read two or more tables simultaneously, multiple-issue techniques are needed, such as the 4-

way VLIW used in CryptoManiac. In contrast, our PTLU module allows multiple tables to be read in parallel on a single-issue PAX processor using a single `ptr` instruction.

None of the previous work has proposed an XMUX tree, which is a distinctive feature of our PTLU module. This low-cost combinational logic, performing simple operations on the table output, is a key contributor to the huge speedups obtained. While the XMUX units we presented in this paper XOR's or selects table data, they can be adapted for other applications.

While multimedia instructions in IA-64 [45] and PLX [18][19] include instructions like `shuffle` for byte-sized and larger data, these instructions operate on individual bits in PAX. A bit-level shuffle instruction is also used in the TI C64x DSP [46] but this is a 2-cycle instruction that can only shuffle two halves of the same 32-bit source register. Our work is also new in its application and evaluation of bit-level permutation instructions in public-key cryptography.

Using modified functional units to support binary field arithmetic was first alluded to by Nahum in 1995 [47] but no specifics were given regarding hardware or instruction design. Later, binary-field multiplication instructions were added in [36] to a single-issue 16-bit RISC processor core. Our work differs from these studies in that: (i) we consider dedicated binary field multipliers, which are smaller and faster than dual-field multipliers, (ii) we describe how a half multiplier can be used without much performance degradation but significant area savings, (iii) we consider the performance of binary-field multipliers in combination with other architectural techniques such as IPC scaling and wordsize scaling .

CryptoManiac [44] and Cryptonite [48] are two crypto-processors similar in design goals to PAX, but these have only considered symmetric-key cryptography. [44] was also proposed before AES became a NIST standard [5][14]. Because public-key ciphers are an inevitable component of secure processing, we designed PAX for high-performance processing of both public-key and symmetric-key ciphers. Also, PAX provides support for Elliptic Curve Cryptography, a relatively newer class of public-key ciphers suitable for resource-constrained mobile devices.

X. Conclusions

We presented the architecture and implementation of PAX, a small, scalable processor with very fast crypto-processing. The PAX instruction set is derived by extending a minimalist RISC instruction set with a few PAX-specific instructions that provide huge speedups for important operations in symmetric-key and public-key ciphers. PAX includes a PTLU module for fast parallel table lookups in symmetric-key ciphers, and polynomial multiplication, squaring and bit permutation instructions for Elliptic-Curve Cryptography on binary fields. These instructions are also useful in many other applications that use binary finite fields, such as random number generators, combinatorics, and coding theory [49].

PAX has a concise instruction set, suitable for providing low-cost yet high-performance cryptography processing in resource-constrained environments such as mobile wireless devices. We showed how PAX-based processors can be used to perform public-key authentication within a given latency at a low processor clock rate, hence reducing energy consumption, area, and cost, while preserving performance and security. We also showed how PAX processors can be scaled to provide encryption throughputs that can saturate the link speeds of existing and emerging network technologies such as 3G and 4G wireless at low clock rates.

A major contribution is the demonstration of a software AES-128 implementation at 22 cycles per block encryption using PAX-128. This is equivalent to a rate of 1.38 cycles/byte, rivaling hardware ASIC implementations. This allows a processor with a very low MHz rate to achieve link speed encryption. Furthermore, the PAX processor can also implement other ciphers, including public-key ciphers, unlike a dedicated AES ASIC chip.

The security of the implementation of some of these ciphers is improved as well with the PTLU module. Cache side-channel timing attacks have recently been shown to be viable against cryptographic algorithms like AES [50] that use lookup tables stored in cache. Using the PTLU module of PAX to perform the table lookups precludes these timing attacks from taking place, as the tables do not reside in cache. Table access time is always a constant for all tables in the PTLU module. Consequently, the use of PTLU for AES not only provides tremendous performance improvements but also increases the security of the implementation of AES and other ciphers that use table lookup.

Another major contribution of this paper is the demonstration of the effectiveness of wordsize scaling as a technique for significantly improving performance for both symmetric-key and public-key cryptographic processing. For algorithms like AES and ECC, we showed that the speedup obtained with wordsize scaling is higher than increasing the number of instructions executed per cycle (IPC scaling) in superscalar or VLIW execution, with lower implementation complexity. Furthermore, wordsize scaling can be combined with ISA improvements, IPC scaling, and multicore processors for even higher performance.

REFERENCES

[1] W. Stallings, Network Security Essentials, 2nd Edition, Prentice Hall, Nov. 2002.
[2] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley and Sons, 1996.
[3] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, Oct. 1996.
[4] National Institute of Standards and Technology, Data Encryption Standard (DES), FIPS Publication 46-3, Oct. 1999. Available at <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>.
[5] National Institute of Standards and Technology, Advanced Encryption Standard (AES), FIPS Publication 197, Nov. 2001. Available at <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
[6] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, FIPS Publication 186-2, Jan. 2000. Available at <http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf>.
[7] D. Boneh and N. Daswani, “Experimenting with Electronic Commerce on the PalmPilot”, Proc. Financial Cryptography, Feb. 1999, pp. 1-16.
[8] S. Ravi, A. Raghunathan, and N. Potlapally, “Securing Wireless Data: System Architecture Challenges”, Proc. Int. Sym. System Synthesis (ISSS), Oct. 2002, pp. 195-200.
[9] T.S. Rappaport et al., “Wireless Communications: Past Events and A Future Perspective”, IEEE Communications Magazine, vol. 40, no. 5, May 2002, pp. 148-161.
[10] Z. Shi, X. Yang, and R.B. Lee, “Arbitrary Bit Permutations in One or Two Cycles”, Proc. IEEE Int. Conf. Application-Specific Systems, Architectures and Processors (ASAP), Jun. 2003, pp. 237-247.

[11] Open Mobile Alliance, Wireless Transport Layer Security Specification WAP-261-WTLS-20010406-a, Apr. 2001. Available at <http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf>.
[12] IEEE, IEEE 802.11 Wireless LAN Standards, <http://grouper.ieee.org/groups/802/11/>.
[13] B. Schneier, The Blowfish Encryption Algorithm, <http://www.schneier.com/blowfish.html>.
[14] National Institute of Standards and Technology, Advanced Encryption Standard (AES) Development Effort, Jan. 1997-Nov. 2001, <http://csrc.nist.gov/CryptoToolkit/aes/index2.html>.
[15] B. Schneier et al., "Twofish: A 128-bit Block Cipher", Jun. 1998, <http://www.schneier.com/twofish.html>.
[16] C. Burwick et al., "MARS – A Candidate Cipher for AES", Sep. 1999, <http://www.research.ibm.com/security/mars.pdf>.
[17] B. Gladman, AES Second Round Implementation Experience, source code for AES finalists available at <http://fp.gladman.plus.com/cryptography_technology/aesr2>.
[18] Princeton Architecture Laboratory for Multimedia and Security, PLX Project, <http://palms.ee.princeton.edu/PLX>.
[19] R.B. Lee and A.M. Fiskiran, "PLX: An Instruction Set Architecture and Testbed For Multimedia Information Processing", Journal of VLSI Signal Processing, vol. 40, 2005, pp. 85-108.
[20] W. Josephson, R. Lee, and K. Li, "ISA Support for Fingerprinting and Erasure Codes, Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)," July 2007.
[21] R.B. Lee, "Subword Parallelism with MAX-2", IEEE Micro, vol. 16, no. 4, Aug. 1996, pp. 51-59.
[22] R.B. Lee, Z. Shi, and X. Yang, "Efficient Permutation Instructions for Fast Software Cryptography", IEEE Micro, vol. 21, Dec. 2001, no. 6, pp. 56-69.
[23] N. Koblitz, "Elliptic Curve Cryptosystems", *Mathematics of Computation*, vol. 48, no. 177, 1987, pp. 203-209.
[24] V.S. Miller, "Use of Elliptic Curves in Cryptography", *Lecture Notes in Computer Science*, vol. 218, Springer-Verlag, 1986, pp. 417-426.
[25] A.K. Lenstra and E.R. Verheul, "Selecting Cryptographic Key Sizes", *Journal of Cryptology*, vol. 14, no. 4, Dec. 2001, pp. 255-293.
[26] National Institute of Standards and Technology, "Key Management Guideline", 2nd Key Management Workshop, Nov. 2001. Available at <http://csrc.nist.gov/CryptoToolkit/kms/key-management-guideline-(workshop).pdf>.
[27] A. Menezes, Elliptic Curve Public Key Cryptosystems, Kluwer Academic Publishers, 1993.
[28] M. Rosing, Implementing Elliptic Curve Cryptography, Manning, 1998.
[29] ANSI, ANSI X9.62 - Public Key Cryptography for Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 1999.
[30] IEEE, IEEE P1363 Standard Specifications for Public-Key Cryptography, <http://grouper.ieee.org/groups/1363>.
[31] ISO/IEC 14888-3, Information Technology – Security Techniques – Digital Signatures – Part 3: Certificate-Based Mechanisms, 1998.
[32] D. Hankerson, J.L. Hernandez, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography Over Binary Fields", *Lecture Notes in Computer Science*, vol. 1965, Jan. 2000, pp. 1-24.
[33] J. Lopez and R. Dahab, "Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation", Lecture Notes in Computer Science, vol. 1717, 1999, pp. 316-327.
[34] R. Schroeppel et al., "Fast Key Exchange with Elliptic Curve Systems", Lecture Notes in Computer Science, vol. 963, 1995, pp. 43-56.
[35] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0", Computer Architecture News, Jun. 1997, pp. 13-25.
[36] J. Großschädl and G.-A. Kamendje, "Instruction Set Extension for Fast Elliptic Curve Cryptography Over Binary Finite Fields GF(2m)", Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP), Jun. 2003, pp. 455-468.
[37] E. Savas, A.F. Tenca, and C.K. Koc, "A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^m)", Lecture Notes in Computer Science, vol. 1965, Jan. 2000, pp. 277-292.
[38] A.M. Fiskiran and R.B. Lee, "Evaluating Instruction Set Extensions for Fast Arithmetic on Binary Finite Fields", Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP), Sep. 2004, pp. 125-136.
[39] HP Labs, CACTI, < http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html>.
[40] R.B. Lee and A.M. Fiskiran, "Multimedia Instructions in Microprocessors for Native Signal Processing", Programmable Digital Signal Processors, Yu Hen Hu, ed., Marcel Dekker, Dec. 2001, pp. 91-145.
[41] A.M. Fiskiran and R.B. Lee, "Fast Parallel Table Lookups to Accelerate Symmetric-Key Cryptography", Proceedings of the International Conference on Information Technology Coding and Computing, Embedded Cryptographic Systems Track, April 2005, pp. 526-531.
[42] Marvell PXA320 Processor Series, Marvell, Document ID PXA320-001, available at <http://www.marvell.com/files /products/cellular/application/PXA320_PB_R4.pdf>.
[43] J. Burke, J. McDonald, and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography", Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), Nov. 2000, pp. 178-189.
[44] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication", Proc. Annual Int. Symposium on Computer Architecture (ISCA), Jun. 2001, pp. 110-119.
[45] R.B. Lee, A.M. Fiskiran, and A. Bubshait, "Multimedia Instructions in IA-64", Proc. IEEE Int. Conf. Multimedia and Expo (ICME), Aug. 2001, pp. 281-284.
[46] Texas Instruments, "TMS320C6000 CPU and Instruction Set Reference Guide", doc. SPRU189F, Oct. 2000, available at <http://www.ti.com>.
[47] E.M. Nahum et al., "Towards High-Performance Cryptographic Software", Proc. IEEE Workshop Architecture and Implementation of High-Performance Communication Subsystems (HPCS), 1995, pp. 69-72.
[48] D. Oliva, R. Buchty, and N. Heintze, "AES and the Cryptonite Crypto Processor", Proc. Int. Conf. Compiler, Architectures, and Synthesis for Embedded Systems, Oct. 2003, pp. 198-209.
[49] R. Lidl and H. Niederreiter, Introduction to Finite Fields and Their Applications, Cambridge University Press, 1986.
[50] D.A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," Cryptology ePrint Archive, Report 2005/271, 2005.