Cryptonite – A Programmable Crypto Processor Architecture for High-Bandwidth Applications

Rainer Buchty[*], Nevin Heintze, and Dino Oliva

Agere Systems
101 Crawfords Corner Rd
Holmdel, NJ 07733, USA
{buchty|nch|oliva}@agere.com

Abstract. Cryptographic methods are widely used within networking and digital rights management. Numerous algorithms exist, e.g. spanning VPNs or distributing sensitive data over a shared network infrastructure. While these algorithms can be run with moderate performance on general purpose processors, such processors do not meet typical embedded systems requirements (e.g. area, cost and power consumption). Instead, specialized cores dedicated to one or a combination of algorithms are typically used. These cores provide very high bandwidth data transmission and meet the needs of embedded systems. However, with such cores changing the algorithm is not possible without replacing the hardware. This paper describes a fully programmable processor architecture which has been tailored for the needs of a spectrum of cryptographic algorithms and has been explicitly designed to run at high clock rates while maintaining a significantly better performance/area/power tradeoff than general purpose processors. Both the architecture and instruction set have been developed to achieve a bits-per-clock rate of greater than one, even with complex algorithms. This performance will be demonstrated with standard cryptographic algorithms (AES and DES) and a widely used hash algorithm (MD5).

1 Introduction and Motivation

Hardware ASIC blocks are still the only available commercial solution for high-bandwidth cryptography. They are able to meet functionality and performance requirements at comparably low costs and, importantly for embedded systems applications, low power consumption.

Their chief limitation is their fixed functionality: they are limited to the algorithm(s) for which they have been designed. In contrast, a general purpose processor is a much more flexible approach and can be used to implement any algorithm. The current generation of these processors has sufficient computing power to provide moderate levels of cryptographic performance. For example, a high-end Pentium PC can provide encryption rates of hundreds of MBits/sec. However, general purpose processors are more than

[*] Rainer Buchty is now a member of the University of Karlsruhe, Institute for Computer Design and Fault Tolerance, Chair for Computer Architecture and Parallel Processing. You can reach him via buchty@ira.uka.de

100x larger and consume over 100x more power than dedicated hardware with comparable performance. A general purpose processor is simply too expensive and too power hungry to be used in embedded applications.

Our goal in this paper is to provide a better tradeoff between flexibility and performance/area/power in the context of embedded systems, especially networking systems. Our approach is to develop a programmable architecture dedicated to cryptographic applications. While this architecture may not be as flexible as a general purpose processor, it provides a substantially better performance/area/power tradeoff.

This approach is not new. An early example is the PLD001 processor [17]. While this processor is specifically designed for the IDEA and RSA algorithms, it is in fact a microprogrammable processor and could in principle be used for variations of these algorithms.

A more recent approach to building a fully programmable security processor is CryptoManiac [28]. The CryptoManiac architecture is based on a 4-way VLIW processor with a standard 32-bit instruction set. This instruction set is enhanced for cryptographic processing [7] by the addition of crypto instructions that combine arithmetic and memory operations with logical operations such as XOR, and embedded RAM for table-based permutation (called the SBOX Cache). These crypto instructions take one to three cycles. The instruction set enhancement is based on an analysis of cryptographic applications. However, the analysis made assumptions about key generation which are not suitable for embedded environments.

CryptoManiac is very flexible since it is built on a general purpose RISC-like instruction set. It provides a moderate level of performance over a wide variety of algorithms, largely due to the special crypto instructions. However, the CryptoManiac architecture puts enormous pressure on the register file, which is a centralized resource shared by all functional units. The register file provides 3 source operands and one result operand per functional unit [28], giving a total of at least 16-ports on a 32x32-bit register array, running at 360 MHz. This is still a challenge for a custom design using today's 0.13μm technology. It would be even more difficult using the 0.13μm library-based ASIC tool flows typical for embedded system chips.

Unlike CryptoManiac, CRYPTONITE was designed from the ground up, and not based on a pre-existing instruction set. Our starting point was an in-depth application analysis in which we decomposed standard cryptographic algorithms down to their core functionality as described in [6]. CRYPTONITE was then designed to directly implement this core functionality. The result is a simple light-weight processor with a smaller instruction set, a distributed register file with significantly less register port pressure, and a two-cluster architecture to further reduce routing constraints. It natively supports both 64-bit and 32-bit computation. All instructions execute in a single cycle. Memory access and arithmetic functions are strictly separated.

Another major difference from CryptoManiac is in the design of specialized instructions for cryptographic operations. CryptoManiac provides instructions that combine up to three standard logic, arithmetic, and memory operations. In contrast, CRYPTONITE supports several instructions that are much more closely tailored to cryptographic algorithms such as parallel 8-way permutation lookups, parameterized 64-bit/32-bit rotation, and a set of XOR-based fold operations. CRYPTONITE was designed to minimize implementation complexity, including register ports, and size, number and length of the internal data paths.

A companion paper [10] focuses on AES and the software for implementing AES on CRYPTONITE. It describes novel techniques for AES implementation, the AES-relevant aspects of the CRYPTONITE architecture and how AES influenced the design of CRYPTONITE.

In this paper we focus on the overall architecture and design methodology as well as give details of those aspects of the architecture influenced by DES and hashing algorithms such as MD5, including the distributed XOR unit and the DES unit of CRYPTONITE.

2 Key Design Ideas

CRYPTONITE was explicitly designed for high throughput. Our approach combines single-cycle instruction execution with a three-stage pipeline consisting of simple stages that can be clocked at a high rate. Our architecture is tailored for cryptographic algorithms, since the more closely an architecture reflects the structure of an application, the more efficiently the application can be run on that architecture. The architecture also addresses system issues. For example, we have designed CRYPTONITE to generate round keys needed for encryption and decryption within embedded systems.

To achieve these goals while keeping implementation complexity to a minimum, CRYPTONITE employs a number of architectural concepts which will be discussed in this section. These concepts arose from an in-depth analysis of several cryptographic algorithms described in [6], namely DES/3DES [4], AES/Rijndael [9,8], RC6 [21], IDEA [22], and several hash algorithms (MD4 [19,18], MD5 [20], and SHA-1 [5]).

2.1 Two-Cluster Architecture

Most other work on implementing cryptographic algorithms on a programmable processor focuses solely on the core encryption algorithm and does not include round key generation (i.e. the round keys have to be precomputed). For embedded system solutions, however, on-the-fly round key generation is vital because storing/retrieving the round keys for thousands or millions of connections is not feasible. Coarse-grain parallelism can be exploited; round key calculation is usually independent of the core cryptographic operation. For example, in DES [4], the round key generation is completely independent of encryption or decryption. The loose coupling is where the round key is fed into the encryption process. Certain coarse-grained parallelism exists also within hash algorithms like MD5 [20] or SHA [5]: these algorithms consist of the application of a non-linear function (NLF) followed by further adding of table-based values. In particular, the hash function's NLF can be calculated in parallel with summing up the table-based values.

Our analysis revealed that many algorithms show similar structure and would benefit from an architecture providing two independent computing clusters. Algorithm analysis further indicated that two clusters are a reasonable compromise between algorithm support and chip complexity. Adding further clusters would rarely result in speeding up computation but rather increase silicon.

2.2 XOR Unit

XOR is a critical operation in cryptography[1]. Several algorithms employ more than one XOR operation per computation round or combine more than two input values. Therefore, using common two-input functions causes an unnecessary sequentiality; a multi-input XOR function would avoid this sequentiality. Such a unit is easy to realize as the XOR function is fast, simple and cheap in terms of die size and complexity. Thus, CRYPTONITE employs a 6-input XOR unit which can take any number from one to 6 inputs. These inputs are the four ALU registers, data coming from the memory unit, result data linked from the sibling ALU, and an immediate value. As the XOR unit can additionally complement its result, it turns into a negation unit when only one input is selected.

Signal routing becomes an issue with bigger data sizes. For this reason the 6-input XOR function was embedded into the data path: Instead of routing ALU registers individually into the XOR unit, the XOR function has been partially embedded into the data path from the registers to the XOR unit. Thus only one intermediate result instead of four source values has to be routed across the chip and routing pressure is taken from the overall design. This reduces die size and increases speed of operation.

2.3 Parameterizable Permutation Engine

Another basic operation of cryptographic operations is permutation, commonly implemented as a table lookup. In typical hardware designs of DES, these lookup tables are hardwired. However, for a programmable architecture, hardwired permutation tables are not feasible as they would limit the architecture to the provided tables. Supporting several algorithms would require separate tables and hence increase die size.

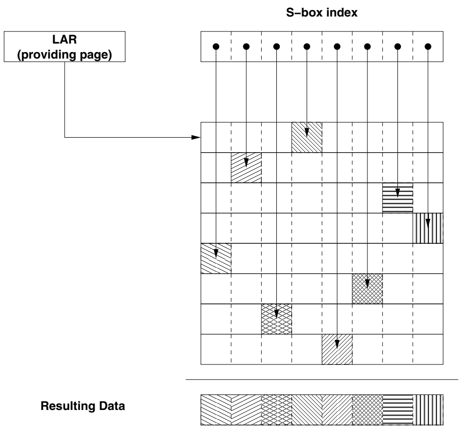

Fig. 1. Vectored Memory Access

[1] XOR is a self-invertible operation. We can XOR data and key to generate cipher text and then XOR the cipher text with the key to recover the data.

Instead, a reconfigurable permutation engine is necessary. Cryptonite employs a novel vector memory unit as its reconfigurable permutation engine. Algorithm analysis showed that permutation lookups are mostly done on a per-byte or smaller basis (e.g. DES: 6-bit address, 4-bit output; AES: permutation based on an 8-bit multiplication table with 256 entries). Depending on the input data size and algorithm, up to 8 parallel lookups are performed. In Cryptonite, the vector memory unit receives a vector of indexes and a scalar base address. This is used to address a vector of memories (i.e. n independent memory lookups are performed). The result is a data vector. This differs from a typical vector memory unit which, when given a scalar address, returns a data vector (i.e. the n data elements are sequentially stored at the specified address).

In non-vector addressing mode, the memory address used is the sum of a base address (from a local address register) and an optional index value. Each memory in the vector of memories receives the same address, and the results are concatenated to return a 64-bit scalar. The vector addressing mode is a slight modification to this scheme: we mask out the lower 8 bits of the base address provided by a local address register (LAR) and the 64 bits of the index vector are interpreted as eight 8-bit offsets from the base address as illustrated by Figure 1. Cryptonite's vector memory unit is built from eight 8-bit memory banks.

2.4 AES-Supporting Functions

The vector memory unit described above is important for AES performance. Cryptonite also provides some additional support instructions for AES[2]. Eight supporting functions are listed in Table 1. Unlike typical DES functions, the AES-supporting functions implement relatively general fold, rotate and interleave functionality and should be applicable to other crypto algorithms.

Table 1. AES-supporting ALU functions

Function	Description
swap(x_{32}, y_{32})	$f(x,y) = y \mid x$
upper(x_{64}, y_{64})	$* \ f(x,y) = x_7 \mid x_3 \mid y_7 \mid y_3 \mid x_6 \mid x_2 \mid y_6 \mid y_2$
lower(x_{64}, y_{64})	$* \ f(x,y) = x_5 \mid x_1 \mid y_5 \mid y_1 \mid x_4 \mid x_0 \mid y_4 \mid y_0$
rbl$_m$(x_{64})	$\star \ f(x) = (x_{63..32} \lll (m*8) \mid (x_{31..0} \lll ((m+1)*8))$
rbr$_m$(x_{64})	$\star \ f(x) = (x_{63..32} \ggg (m*8) \mid (x_{31..0} \ggg ((m+1)*8))$
xor_rbl$_m$(x_{64}, y_{64})	$f(x,y) = rbl_m(x \oplus y)$
fold(x_{64}, y_{64})	$\diamond \ f(x,y) = (x_1 \oplus y_0) \mid (x_0 \oplus y_1 \oplus y_0)$
ifold(x_{64}, y_{64})	$\diamond \ f(x,y) = (x_0 \oplus y_1) \mid (y_1 \oplus y_0)$

$*$ indices denote bytes $\star$ indices denote bits $\diamond$ indices denote 32-bit words

With these functions, it it possible to implement an AES decryption routine with 81 cycles (8 cycles per round, 6 cycles setup, 3 cycles post-processing) and encryption

[2] We note that the need for such functions mainly arises from AES round key generation.

with just 70 cycles (7 cycles per round, 1 cycle setup, 6 cycles post-processing)[3]. Both routines include on-the-fly round-key generation. [10] elaborates on the AES analysis and how supporting functions were determined. It also presents the AES implementation on the CRYPTONITE architecture.

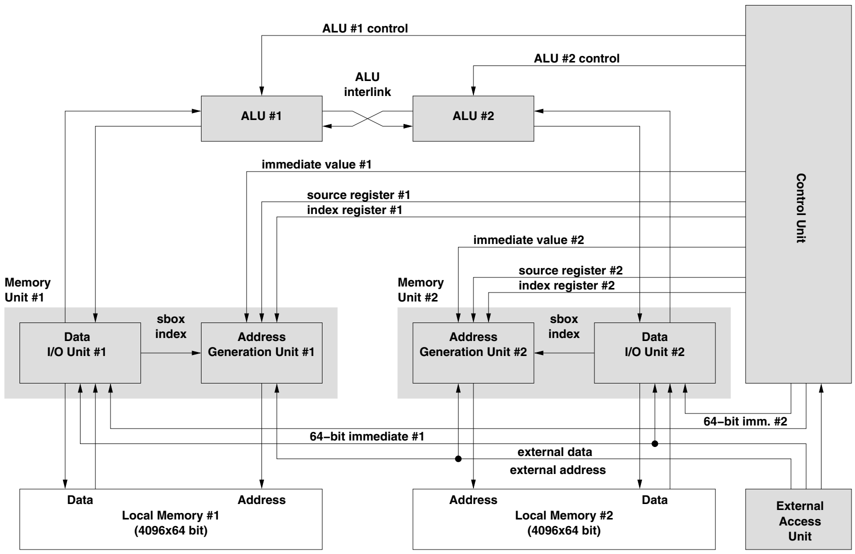

Fig. 2. Overview of the CRYPTONITE architecture

3 The CRYPTONITE Architecture

A high-level view of CRYPTONITE is pictured in Figure 2.

As mentioned previously, application analysis led to the two-cluster architecture. Each cluster consists of an ALU and its accompanying data I/O unit (DIO) managing accesses to the cluster's local data memory. A crosslinking mechanism enables data exchange between the ALUs of both clusters. The overall system is controlled by the control unit (CU) which parses the instruction stream and generates control signals for all other units. A simple external access unit (EAU) provides an easy method to access or update the contents stored in both local data memories: on external access, the CU puts all other units on hold and grants the EAU access to the internal data paths.

The CU also supplies a set of 16 registers for looping and conditional branching. 12 of these are 8-bit counter registers, the remaining four are virtual registers reflecting the two ALU's states: we use these registers to realize conditional branches on ALU results such as zero result return (BEQ/BNE or JZ/JNZ) or carry overflow/borrow (BCC/BCS or JC/JNC). The CRYPTONITE CU is depicted in Figure 3. The use of special purpose

[3] The asymmetry arises from the fact that AES, although symmetric in terms of cryptography, is asymmetric in terms of computation. Decryption needs a higher number of table lookups.

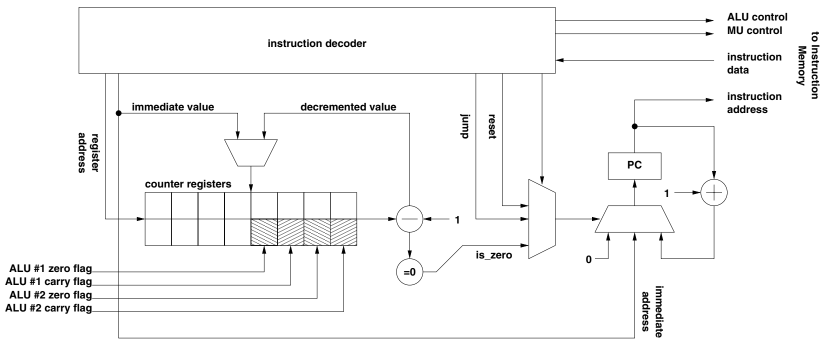

Fig. 3. The CRYPTONITE Control Unit

looping registers reduces register port pressure, and routing issues. In addition, from our application analysis it was clear that most cryptographic algorithms have relatively small static loop bounds. In fact, data-dependent branching is rare (IDEA being one exception). Finally, the use of special purpose registers in conjunction with a post-decrement loop counter strategy allows us to reduce the branch penalty to 1 cycle.

3.1 The CRYPTONITE ALU

Much effort was put into the development of the CRYPTONITE ALU. Our target clock frequency was 400 MHz in TSMC's 0.13 μm process. To reach this goal, we had to carefully balance the ALU's features (as required for the crypto algorithms) and its complexity (as dictated by technology constraints).

One result of this tradeoff is that the number of 64-bit ALU registers in each cluster was limited to four. Based on our application analysis, this was judged sufficient. To compensate for the low register count, each register can be either used as one 64-bit or two individually addressable 32-bit quantities. The use of a 64-bit architecture was motivated by both the requirements of DES and AES as well as parameterizable algorithms like RC6. To enable data exchange between both blocks the first register of each ALU is crosslinked with the first register in the other cluster. This crosslink eases register pressure as it allows cooperative computation on both ALUs (this is critical for AES and MD5). To further reduce register pressure, each ALU employs an accumulator for storing intermediate results.

The ALU itself consists of the arithmetic unit (AU) and a dedicated XOR unit (XU). The AU provides conventional arithmetic and boolean operations but also specialized functions supporting certain algorithms. It follows the common 3-address model with two source registers and one destination register. These registers are limited to the ALU's accumulator, the four ALU registers, data input and output registers of the associated memory unit, and an immediate value provided by the CU. The XU may choose any number of up to 6 source operands. In addition, the XU may optionally complement the output of the XOR. Thus, with only one source operand, the XU can act like a negation unit.

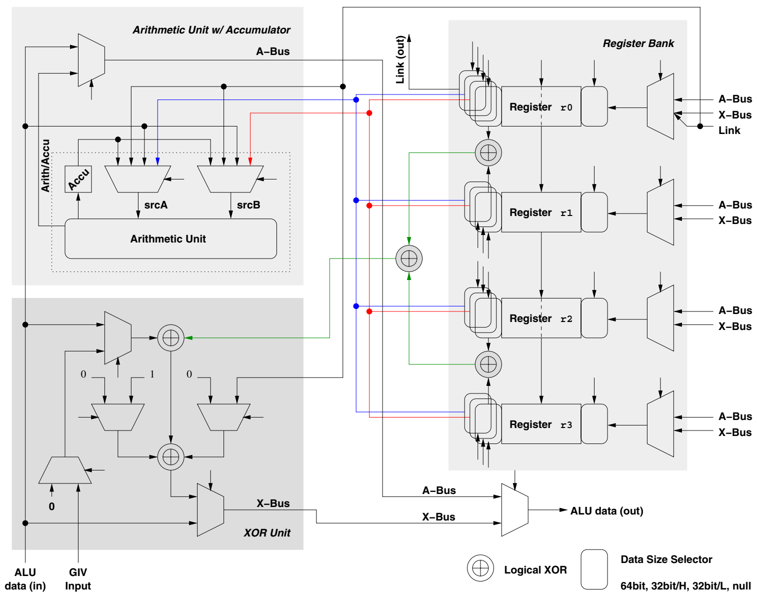

Fig. 4. Overview of CRYPTONITE's ALU

The 64-bit results of AU and XU operations are placed on separate result buses. From these buses, either the upper 32-bits, lower 32-bits or the entire 64-bit value can be selected and stored in the assigned register (or register half). It is not possible to combine two individual 32-bit results from both result buses into one 64-bit register. Results may also be forwarded to the data unit. Figure 4 illustrates the CRYPTONITE ALU with its sub-units.

3.2 The CRYPTONITE Memory Unit

Access to local data memory is handled by the memory unit. It is composed of an address generation unit (AGU) and a data I/O unit (DIO). The address generation unit depicted in Figure 5. It generates the address for local memory access using the local address registers (LAR). The AGU contains a small add/sub/and ALU for address arithmetic. This supports a number of addressing modes such as indexed, auto-increment and wrap-around table access as listed in Table 2.

Furthermore, the SBox addressing mode performs eight parallel lookups to the 64-bit memory with 8-bit indices individual to each lookup. For a detailed description of this addressing mode please refer to Section 2.3.

The DIO, shown in Figure 6, contains two buffer registers which are the data input and data output registers (DIR and DOR). They buffer data from and to local memory. The DOR can also be used as an auxiliary register by the ALU. The DIR also serves as the SBox index to the AGU.

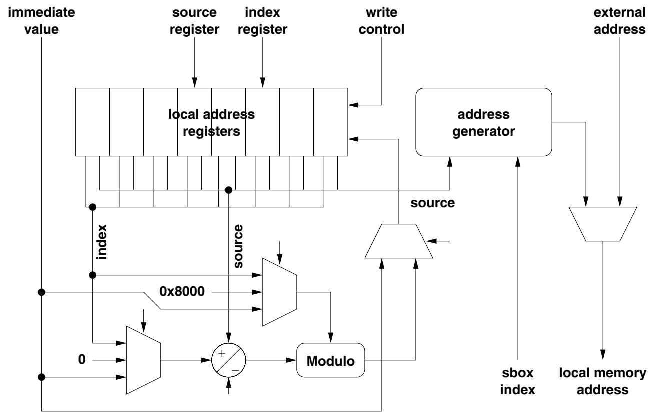

Fig. 5. The Address Generation Unit

Table 2. Addressing modes supported by CRYPTONITE's AGU

Addressing Mode	Address Computation	LAR Update
direct	$addr = LAR$	
", w/ register modulo	$addr = LAR_x$	$LAR_x = LAR_x \,\%\, LAR_y$
", w/ immediate modulo	$addr = LAR_x$	$LAR_x = LAR_x \,\%\, idx$
S-Box	$\forall 0 \leq i \leq 7 : addr_i = (LAR \wedge \texttt{0x7f00}) \vee idx_i$ (LAR unchanged)	
immediate-indexed	$addr = LAR$	$LAR = LAR + idx$
ditto, w/ register modulo	$addr = LAR_x$	$LAR_x = (LAR_x + idx) \,\%\, LAR_y$
register-indexed	$addr = LAR_x$	$LAR_x = LAR_x + LAR_y$
ditto, w/ immediate modulo	$addr = LAR_x$	$LAR_x = (LAR_x + LAR_y) \,\%\, idx$

Addressing modes written in italics are based on architectural side-effects and have not been designed in by purpose.

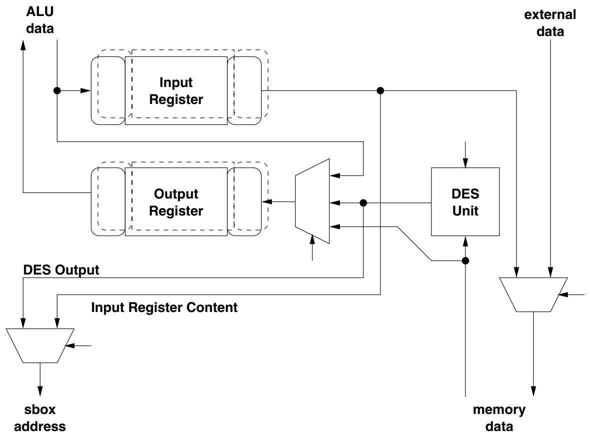

Fig. 6. Data I/O from embedded SRAM

The DIO also contains a specialized DES unit. Fast DES execution not only requires highly specialized operations but also SBox access to memory. Hence the DES support instructions are integrated into the memory unit rather than the ALU.

3.3 The Cryptonite DES Unit

As mentioned in Section 3.2, the DES unit has been implemented into the memory unit instead of the ALU. The reason for doing so was to not bloat the ALU with functions which are plain DES-specific and cannot be reused for other algorithms. Even the most primitive permutation, compression, and expansion functions required for DES computation are clearly algorithm-specific as discussed in detail in [6]. In addition to these bit-shuffling functions, DES computation is based on a table-based transposition realized through an SBox lookup and therefore needs access to the data memory.

For this reason, the DES unit has been placed directly into the memory unit rather than incorporated into the ALU. Doing so, no unnecessary complexity – i.e. die size and signal delay – is added to the ALU. Futhermore, penalty cycles resulting from data transfer from memory to ALU and back are avoided.

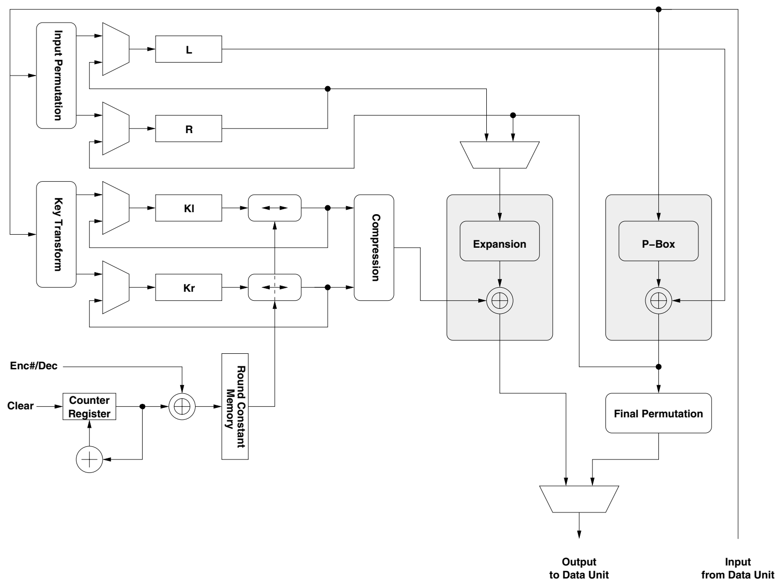

Fig. 7. Cryptonite's DES Unit

The DES unit is pictured in Figure 7. It consists of data (L and R) and key (K) registers, a round counter and a constant memory of 16x2 bits providing the round constant for key shifting. The computation circuitry provides two selectable, monolithic functions performing the following operations:

1. expand R_{i-1}, shift & compress key, and XOR the results
2. permutate the S-Box result using P-Box shuffling and XOR this result with L_{i-1}; forward R_{i-1} to L_i

These functions can be selected independently to enable either initial computation (function 1) or final computation (function 2) needed for the first and last round, or back-to-back execution (function 2 followed by function 1) for the inner rounds of computation.

4 Results

Several algorithms were investigated and implemented on a custom architecture simulator. Based on the simulation results, the architecture was fine-tuned to provide minimum cycle count while maintaining maximum flexibility. In particular, the decision to incorporate the DES support instructions within the memory unit instead of the ALU (see Section 3.2) was directly motivated by simulation results. In this section we will now present the simulation results for a set of algorithm implementations which were run on our architecture simulator.

4.1 DES and 3DES

As CRYPTONITE employs a dedicated DES unit, the results for the DES [4] and 3DES implementations were not surprising. CRYPTONITE reaches throughput of 732 MBit/s for DES and 244 MBit/s for 3DES. In contrast, the programmable CryptoManiac processor [28] achieves performance of 68 MBit/s for 3DES.

To quantify the tradeoffs of programmability versus performance, we give some performance numbers for DES hardware implementations. Hifn's range of cores ([16], 7711 [11], 7751 [12], 7811 [13] and 790x [14,15]) achieve performance of 143-245 MBit/s for DES and 78-252 MBit/s for 3DES. The OpenCore implementation of DES [27] achieves performance of 629 MBit/s. Arguably the state-of-the-art DES hardware implementation is by SecuCore [26]. SecuCore's high-performance DES hardware implementation (SecuCore DES/3DES Core [24]) achieves 2 GBit/s, just a factor of 2.73 better than CRYPTONITE. These results are summarized in Figure 8.

4.2 Advanced Encryption Standard (AES)

Figure 9 compares the AES performance of CRYPTONITE against a set of hardware implementations from Amphion [1], Hifn, and Secucore as well as the programmable CryptoManiac. CRYPTONITE running at 400 MHz outperforms a number of hardware implementations by a factor of 1.25 to 2.6. Compared with CryptoManiac, CRYPTONITE

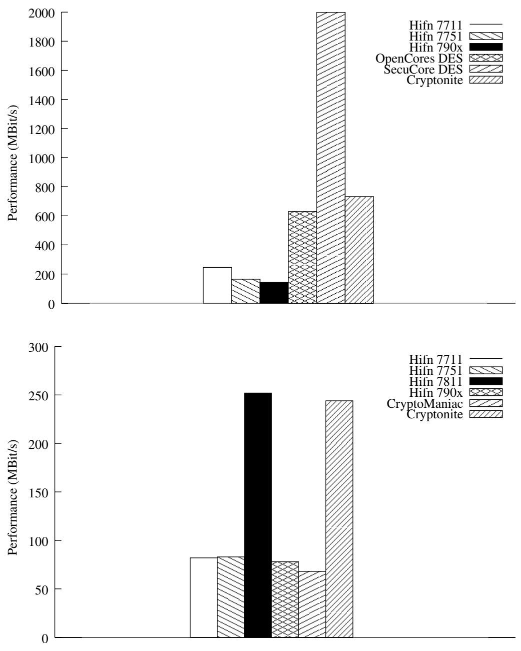

Fig. 8. DES and 3DES Performance Comparison

shows an almost two times better performance.[4] This result justifies our decision to go for simple ALUs providing more specialized functionality.

High-performance hardware AES implementations provided by Amphion CS5210-40 High Performance AES Encryption Cores [2] and CS5250-80 High Performance AES Decryption Cores [3] and SecuCore (Secucore AES/Rijndael Core [23]) are able to outperform Cryptonite by a factor of 2.64. In addition, an extremely fast implementation from Amphion is even able to reach 25.6 GBit/s performance. This performance, however, is paid with an enormous gate count (10x bigger than other hardware solutions) which is why this version has not been included in the comparison chart shown in Figure 9.

4.3 MD5 Hashing Algorithm

Cryptonite performance on MD5 was 421 MBit/s at 400 MHz clock speed. It outperforms the Hifn hardware cores (7711 [11], 7751 [12], 7811 [13], and 790x [14,15])

[4] We remark that the CryptoManiac results appear to exclude round key generation whereas Cryptonite includes round key generation. In the RC4 discussion, [28] mentions impact from writing back into the key table. A similar note is missing for the AES implementation which suggests that only the main encryption algorithm (i.e. excluding round key generation) was coded. The cycle count of just 9 cycles per round without significant AES instruction support seems consistent with this assumption.

by factors of 1.12 to 7.02. SecuCore's high-performance MD5 core (SecuCore SHA-1/MD5/HMAC Core [25]), is a factor of 2.97 faster than CRYPTONITE, highlighting the programmability tradeoff. A comparison with CryptoManiac is omitted because the performance of MD5 is not reported in [28]. Figure 10 summarizes the results for MD5.

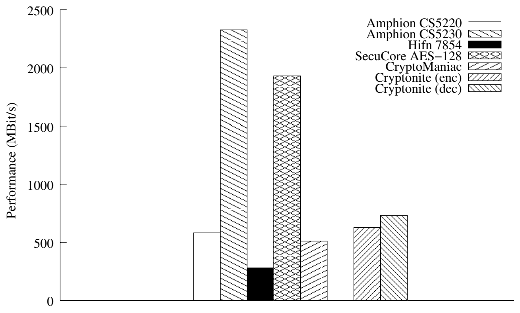

Fig. 9. AES-128/128 Performance Comparison

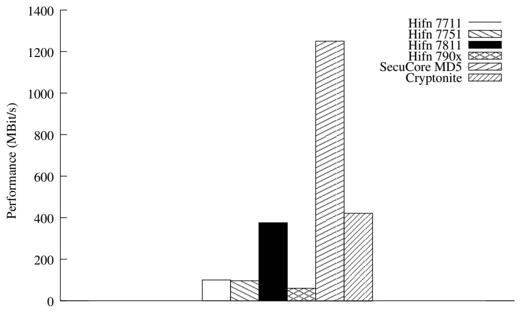

Fig. 10. MD5 Performance Comparison

5 Summary

We have presented CRYPTONITE, a programmable processor architecture targeting cryptographic algorithms. The starting point of this architecture was an in-depth application analysis in which we decomposed standard cryptographic algorithms down to their core functionality. The CRYPTONITE architecture has several novel features, including a distributed multi-input XOR unit and a parameterizable permutation unit built using new form of vector-memory block. A central design constraint was simple implementation, and many aspects of the architecture seek to reduce port counts on register files, number and width of internal buses and number and size of registers. In contrast, CryptoManiac has a number of implementation challenges, including a heavyweight 16-port register file. We expect the CRYPTONITE die size to be significantly smaller than CryptoManiac.

A number of algorithms (including AES, DES and MD5) were implemented on the architecture simulator with promising results. CRYPTONITE was able to outperform numerous hardware cores. It outperformed the programmable CryptoManiac processor by factors of between two and three and comparable clock speeds. To determine the tradeoff between programmability and dedicated high-performance hardware cores, CRYPTONITE was compared to cores from Amphion and Secucore: these outperform CRYPTONITE by about a factor of 3.

References

1. Amphion Semiconductor Ltd. Corporate Web Site. 2001. http://www.amphion.com.
2. Amphion Semiconductor Ltd. CS5210-40 High Performance AES Encryption Cores Product Information. 2001. http://www.amphion.com/acrobat/DS5210-40.pdf.
3. Amphion Semiconductor Ltd. CS5210-40 High Performance AES Decryption Cores Product Information. 2002. http://www.amphion.com/acrobat/DS5250-80.pdf.
4. Ronald H. Brown, Mary L. Good, and Arati Prabhakar. Data Encryption Standard (DES) (FIPS 46-2). *Federal Information Processing Standards Publication (FIPS)*, Dec 1993. http://www.itl.nist.gov/fipspubs/fip46-2.html (initial version from Jan 15, 1977).
5. Ronald H. Brown and Arati Prabhakar. FIPS180-1: Secure Hash Standard (SHA). *Federal Information Processing Standards Publication (FIPS)*, May 1993. http://www.itl.nist.gov/fipspubs/fip180-1.htm.
6. Rainer Buchty. *Cryptonite – A Programmable Crypto Processor Architecture for High-Bandwidth Applications*. PhD thesis, Technische Universität München, LRR, September 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/buchty.pdf.
7. Jerome Burke, John McDonald, and Todd Austin. Architectural support for fast symmetric-key cryptography. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.
8. J. Daemen and V. Rijmen. The block cipher Rijndael, 2000. LNCS1820, Eds: J.-J. Quisquater and B. Schneier.
9. J. Daemen and V. Rijmen. Advanced Encryption Standard (AES) (FIPS 197). Technical report, Katholijke Universiteit Leuven / ESAT, Nov 2001. http://csrc.nist.gov/publications/-fips/fips197/fips-197.pdf.
10. Nevin Heintze Dino Oliva, Rainer Buchty. AES and the Cryptonite Crypto Processor. *CASES'03 Conference Proceedings*, pages 198–209, October 2003.
11. Hifn Inc. 7711 Encryption Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0001-04-7711.pdf.
12. Hifn Inc. 7751 Encryption Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0013-03-7751.pdf.
13. Hifn Inc. 7811 Network Security Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0018-02-7811.pdf.
14. Hifn Inc. 7901 Network Security Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0023-01-7901.pdf.

15. Hifn Inc. 7902 Network Security Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0040-00-7902.pdf.
16. Hifn Inc. Corporate Web Site. 2002. http://www.hifn.com.
17. Jüri Pöldre. Cryptoprocessor PLD001 (Master Thesis). June 1998.
18. R. Rivest. RFC1186: The MD4 Message-Digest Algorithm. October 1990. http://www.ietf.org/rfc/rfc1186.txt.
19. R. Rivest. The MD4 message digest algorithm. *Advances in Cryptology - CRYPTO '90 Proceedings*, pages 303–311, 1991.
20. R. Rivest. RFC1312: The MD5 Message-Digest Algorithm, April 1992. http://www.ietf.org/rfc/rfc1321.txt.
21. Ronald R. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6^{TM} Block Cipher. August 1998. http://www.rsasecurity.com/rsalabs/rc6/.
22. Bruce Schneier. 13.9: IDEA. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*, pages 370–377, 1996. ISBN 3-89319-854-7.
23. SecuCore Consulting Services. SecuCore AES/Rijndael Core. 2001. http://www.secucore.com/secucore_aes.pdf.
24. SecuCore Consulting Services. SecuCore DES/3DES Core. 2001. http://www.secucore.com/secucore_des.pdf.
25. SecuCore Consulting Services. SecuCore SHA-1/MD5/HMAC Core. 2001. http://www.secucore.com/secucore_hmac.pdf.
26. SecuCore Consulting Services. Corporate Web Site. 2002. http://www.secucore.com/.
27. Rudolf Usselmann. OpenCores DES Core. Sep 2001. http://www.opencores.org/projects/des/.
28. Lisa Wu, Chris Weaver, and Todd Austin. Cryptomaniac: A fast flexible architectore for secure communication. In *28th Annual International Symposium on Computer Architecture (ISCA 2001)*, June 2001.