Modelling `Cryptonite`
On the Design of a Programmable High-Performance Crypto Processor

Rainer Buchty

University of Karlsruhe

Institute for Computer Design and Fault Tolerance

PO Box 6980, Zirkel 2

76128 Karlsruhe, Germany

`buchty@ira.uka.de`

Abstract:

Cryptographic algorithms – even when designed for easy implementability on general purpose architectures – still show a huge performance gap between implementations in software and those using dedicated hardware. Such hardware is usually only able to deal with one single algorithm or a very narrowly defined set of algorithms. The tradeoff between speed/throughput and flexibility can be eased by programmable crypto architectures. These can be existing general purpose architectures enhanced by specialized functional units which fulfill the requirements of typical cryptographic algorithms. Alternatively, a fully custom architecture can be designed.

In this paper we describe the methods used to design a programmable crypto architecture from scratch. We will introduce a set of typical cryptographic algorithms, investigate their requirements, and finally show the weighted result leading to our `Cryptonite` architecture.

1 Introduction

Although recent cryptographic algorithms have been designed to be easy to implement on such standard architectures, they still greatly benefit from special functional units and architectural features. Several hardware cores exist which support either a single cryptographic algorithm or a set of these needed for typical applications such as IPsec. So far, only few fully programmable (as opposed to parameterizable) architectures exist: the probably most well-known one is CryptoManiac [WWA01], which is based on a common RISC-style architecture which was extended to especially support cryptographic algorithms. This extension evolved from an instruction set architecture (ISA) [BMA00] based on exhaustive algorithm analysis.

In contrast, our CRYPTONITE architecture [Bu02, OBH03, BHO04] was designed from scratch. Aim was to keep the architecture as simple as possible while ensuring maximum performance for a broad range of algorithms at minimal manufacturing costs. A typical set of cryptographic algorithms was analyzed to determine the architectural requirements.

The results of the analysis were then collected and weighted against current and future proliferance/importance/use of each algorithm.

We will start with an overview over the selected algorithms and the analysis results and will then discuss the influence of each algorithm on the architecture and finally sum up the architectural features of our architecture[1]. We will also give an example of how we implemented the AES algorithm using the architecture's special features.

2 Algorithm Analysis

We chose the following algorithms for analysis:

DES: The Data Encryption Standard was developed in the early 1970s and became the official encryption standard in 1977. Despite its age it is still widely used, but will most likely be replaced by the new Advanced Encryption Standard (see below). Unlike modern algorithms, DES is very hardware-centric.

AES: The new encryption standard was introduced in 2001. When the work on the Cryptonite architecture started, it was not yet clear which algorithm participating in the AES competition would be chosen as the new standard. Therefore, we selected the most favored candidates, Rijndael and RC6.

IDEA: The International Data Encryption Algorithm was mainly selected because of its 16-bit design. Its most prominent application is within the Pretty Good Privacy (PGP) software where it is used for securing email transmissions.

MD5 and SHA1: We also selected the two hashing algorithms MD5 and SHA1; these algorithms are used within IPsec for authentication purposes, especially MD5 is also frequently used to provide fingerprints of download data used to ensure data integrity or detect unauthorized alterations.

These algorithms divide into two main classes, **hardware-oriented algorithms**, represented by DES, were originally modelled to run on dedicated hardware. Because of this, they are sometimes problematic to be efficiently implemented on general purpose architectures. DES, for example, is operating on a wide set of data sizes.

The second class are the **software-oriented algorithms**; although designed for easy implementation in software, these algorithms usually prefer certain data sizes. In addition, a huge performance gap might exist between a straight-forward and an optimized version. AES is an example for an algorithm where the implementation strategy has huge impact on performance.

We will now give an overview over the selected algorithms and the observations made regarding these algorithms. The interested reader is referred to [Bu02] for a more detailed elaboration.

---

[1]A detailed discussion of the CRYPTONITE architecture will be given on the ARCS'04 conference held in conjunction with this workshop.

2.1 DES

As already mentioned, DES [BGP93] is a hardware-oriented algorithm. It takes two 64-bit input quantities, data and key, to produce a 64-bit output. The main encryption process is a series of applied transformations, some of which are simple bit-transformations, and others non-linear transformations based on table lookups. Although the input and output data size of DES is 64 bits, we find numerous data sizes during DES computation: within round computation 32- and 48-bit sizes are used. The round key generation, in term, uses a 2x28-bit representation. These various operand sizes get transformed into each other using bit-permutation which are easy to realize in hardware but expensive on general purpose architectures, and table-lookup functions.

With the round key computation being independent from the encryption process, coarse-grained parallelism can be exploited by two parallel computation units. These need to be able to forward results to its sibling unit.

DES greatly benefits from an 8-parallel lookup mechanism supporting the S-box permutation. Apart from this, DES is resource friendly requiring only two registers per computation unit. Then, a swapping mechanism is needed to exchange the encryption variables L and R. In addition, basic loop support with the ability to use the loop counter for memory addressing (round constant lookup) and conditional branching is needed.

Further speedup of DES is only possible by the introduction of very specialized monolithic functions performing a sequence of bit-permutation and XOR operations. Otherwise only minimal speed-up compared to general purpose architectures can be achieved.

2.2 AES

The AES algorithm was chosen through a competition. Although it was not entirely clear which algorithm would win the competition when this work started, there was already a strong bias towards two algorithms. Therefore, we took both candidates – RC6 and Rijndael – into account.

2.2.1 RC6

Much like DES, RC6 [RRSY98] can be easily analyzed through its round data flow graph: It shows, that the round computation separates into two almost independent strands. Round key generation was omitted, since it is rather heavyweight and might lead to performance issues [RRY00] caused by heavy use of multiplication and modulo operations. In this respect, RC6 reminds more of IDEA and is quite different from DES and Rijndael.

Although it looks like two operands per strand might be enough for RC6, a closer look reveals the need for additional storage to forward intermediate results to later computation stages; to ease register pressure, four computation registers per strand are recommended.

RC6 is not bound to a specific data and key size. It is rather defined as RC6-$w/r/b$ where

w denotes the size of data chunks in bits, r the number of computation rounds and b the length of the encryption key in bytes. For AES configuration, RC6 has been locked to RC6-32/20/16, i.e. 32-bit data chunks, 20 rounds of computation and 16-byte (128-bit) keys. Referring to this configuration, all computations are based on 6 distinct operations which are integer addition, subtraction and multiplication (all modulo 2^{32}), bitwise XOR, plus left- and right-rotation of 32-bit operands by the least significant 5 bits of a second operand.

Recalling that we already defined the operand size as 64-bit for DES and RC6 requires twice as much registers but at half the size, it is quite natural to solve RC6's requirements by introducing a mechanism which splits the existing 64-bit registers into two 32-bit registers. Similarly, carry-based arithmetic operations such as addition and subtraction can be implemented in a way that propagation between bits 32 and 33 will take place (64-bit operation) or not (32-bit operation).

Memory access requirements of RC6 are moderate: only the precomputed round key has to be fetched from memory. Computation speed definitely benefits from memory access being done in parallel to computation.

2.2.2 Rijndael

Our analysis of Rijndael [DR01] is quite exhaustive and fills a paper on its own. We therefore refer the interested reader to [OBH03]. At this point, we will only concentrate on the main points of this analysis.

It can be shown that the parallelism scheme required for RC6 also serves Rijndael. Two strands with parallel arithmetic and memory access operations are sufficient. Similarly, Rijndael benefits from the "splitting" of 64-bit registers and operations. The 8-parallel memory lookup mechanism needed for DES can be adapted for Rijndael's substitution function, but needs to be extended to 8-bit address and 8-bit data.

Rijndael also requires some basic rotation functions; unlike DES and RC6 which rotate bit-wise, Rijndael rotates byte-wise. This requires a slight modification of the already introduced rotation mechanism.

The major issues with Rijndael are embedding the round key generation into the encryption process[2] and to efficiently implement the column mixing operation.

Like RC6, also Rijndael is configurable in data and key size. The number of computation rounds, however, is dependent on the selected data and key sizes. For AES, Rijndael has been locked to a data size of 128 bits, key size can be either 128, 192, or 256 bits. Since Rijndael became AES in 2001, these configurations are commonly known as AES-128, AES-192, and AES-256.

Fetching the precomputed keys from memory as done in the AES reference implementation [DR01], is not sensible for embedded systems and/or high-bandwidth applications: depending on the configuration, between 40 and 56 32-bit round keys need to be precom-

---

[2]We decided to not pursue further investigation on RC6, especially how to embed round key generation, since by then Rijndael had been selected as the AES algorithm.

puted and fetched from memory. Neglecting the delay resulting from precomputation, this not only requires up to 224 bytes of memory per secured data stream but also demands a very high memory bandwidth: an entire computation would need up to 56 32-bit lookups per processed data packet. Given the presence of 32-bit memory, securing a single 100Mbit connection would therefore require over 43 million memory accesses per second, i.e. a memory bandwidth of 175MByte/s.

In [OBH03] we show a mechanism to embed round key generation into the running computation process. Doing so, it is no longer necessary to fetch precomputed round keys from memory thus avoiding the problems mentioned above. The drawback of this solution is the need for supporting instructions performing cascaded XOR operations on single register contents and specialized "mingle" instructions interweaving selected bytes of two 64-bit registers into a 64-bit result.

An additional impact of our approach is increased register pressure: because of the integration of the round key generation into the main computation process, our implementation greatly benefits from the presence of additional registers. We decided to extend the use of the memory I/O buffering registers: instead of transparently buffering memory transfers, these registers become fully addressable by the arithmetic unit and serve as a dedicated output (to memory) and an input/output (from memory or intermediate computation result) register. By doing so, we solve the specific requirements of this algorithm while keeping the core architecture mostly unchanged. As a side effect, we also require the register merge operations to accept not only individual registers as input (i.e. merge top 32-bits of one register with bottom 32-bits of another register) but also the two 32-bit halves of a single register (resulting in a swap operation).

In the column mixing stage, the Rijndael state (please refer to [DR01] for terminology) is taken as one matrix and a set of factors as a second one to perform a matrix multiplication in GF(256). Each matrix consists of four rows holding 32-bits each where each row dissects into four 8-bit values.

Addition in GF(256) thankfully maps to XOR and is therefore easy to implement; multiplication, however, is rather complex. The method proposed by the standard is to first transform the factors into their "logarithms" (based on an irreducible polynom) using a transformation table. These result values then get added (i.e. XORed) and the resulting sum is re-transformed using an "anti-log" table acting as the reverse to the aforementioned "logarithm" table. However, this approach is rather time consuming since a total of 64 multiplications is needed resulting in 192 table lookups per round or up to 2688 lookups per entire computation. Following the above 100Mbit example results in a stunning 2.1GBit/s of memory bandwith.

We follow a different approach: first, the multiplication is condensed down to a single memory lookup instead of three consecutive ones by providing distinct tables for each type of multiplication. Rijndael requires only a 6 different multiplication tables so the increase in memory size compared to the log/antilog version is only a factor of 3.

Second, we reuse the 8-parallel lookup mechanism introduced with DES's S-box permutation. Also the Rijndael substitution function benefits from this mechanism, after being extended to 8-bit address and data, as it enables fast transformation of the Rijndael state.

In addition, such a unit can be used to speed up multiplication: given the fact that multiplication in GF(256) is also just a table-based transformation with both, factors and results, not exceeding 8 bits in size, we can perform 8 parallel multiplications using the 8-parallel memory lookup mechanism. This, of course, requires a careful alignment (vector packing) of data both in registers and memory and the application of the aforementioned "32-bit splitting" of the 64-bit architecture. A full discussion of this topic is presented in [OBH03].

2.3   IDEA

IDEA [Sc96] was presented in 1991 and introduced a higher level of security. Where data size remains 64 bit (like DES), key size was doubled to 128 bit. It was especially designed for easy implementability on 16-bit architectures, thus all intermediate computation steps are performed on 16-bit quantities.

In terms of parallelism, this algorithm showed some differences to the previous ones: following the IDEA data flow graph, also two strands can be identified at first sight. However, tighter coupling of the individual computation flow leads to inherent sequentiality. Certain computation stages, in term, would greatly benefit from a 4-wide arithmetic unit. Additionally, IDEA shows huge register pressure requiring not less than 8 registers.

Similar to RC6, key generation is highly dependent on modulo arithmetics (2^{16}, though); unfortunately, also division is needed, which requires a dedicated modulo-2^{16} multiplication/division unit. Where all computation is done 16-bit, round key generation requires a 128-bit rotation resulting in sequential steps when being mapped to any smaller data size.

2.4   MD5 and SHA1

We chose MD5 [Ri92] and SHA1 [BP93, EJ01] to represent typical hashing algorithms because of their common use e.g. within IPsec. Although similar in concept, the algorithms differ slightly in exploitable parallelism: with MD5, inherent parallelism can be only exploited within the non-linear functions (NLF). With SHA1, ideally two fixed left-rotate operations can be performed in parallel to the NLFs.

If not implemented as monolithic functions but based on single Boolean operations, these NLFs show some exploitable parallelism. Two parallel computations are sufficient for MD5, SHA1 would benefit from three parallel computations in addition to the aforementioned parallel shift operations. Unlike MD5, SHA1 also employs some sort of "round key generation" (W[t]), i.e. it performs certain accumulating computations on the input message chunk.

Both algorithms show a high need for multi-input functions, especially multi-input adders and multi-input XOR functions. Memory-wise, MD5 requires a simple modulo-addressing mode for "wrap-around" indexing into constant tables. SHA1 requires multiport memory for W[t] generation.

3 Algorithm Influence

In the previous section we discussed a set of cryptographic algorithms and their architectural requirements. In this section we will now show how we processed that data to aquire the appropriate architecture parameters.

3.1 Data Sizes

To determine, how each algorithm influences the architecture, we we first collected the required data sizes by summing up all data size requirements into table 1.

Algorithm	64	56	48	32	16	8
(3)DES	+	+	+	+	–	(+)
RC6	(+)	–	–	+	–	–
Rijndael	–	–	–	+	–	+
IDEA	–	–	–	–	+	–
MD5	–	–	–	+	–	+
SHA1	–	–	–	+	–	–

Table 1: Data sizes used by the algorithms

Most algorithms show a strong preference for 32-bit datatypes – with the exception of DES, which seems to require an incredible amount of data type support, but the majority of these mainly arises from intermediate permutation operations. I/O operations require only 64-, 32- and (see below) 8-bit operands. A minor preference for 8-bit datatypes arises from the fact that transposition tables (S-Boxes, substitution tables, GF(256) multiplications) are either 8-bit by nature or – in case of DES – can easily be mapped to this data type.

In any of these cases, the 8-bit data type is only required for table lookups which usually take place as an 8-parallel lookup (resulting in a 64-bit memory access). Simple 8-bit accesses, as needed for MD5, do not need to be explicitly supported: with slight memory waste, 8-bit values can be laid out to 32-bit boundaries.

IDEA is the only algorithm specifically requiring 16-bit data types. Since IDEA's core operations are modulo-2^{16} anyway, there is no need for explicitly supporting that data type. Also, by using precomputed round keys, the 128-it data type is not required.

As a result of the algorithm analysis, the architecture should support only 64-bit and 32-bit scalars and a special 8x8-bit packed vector format.

3.2 Functional Units

Next, we listed the architectural requirements for each algorithm and weighted it against possible reuse (i.e. if another algorithm might benefit from the named architectural feature) and how big the impact on the algorithm performance is. This led to the weighting scheme shown in table 2.

Algorithm	Special Requirements	Reuse	Benefit
(3)DES	Permutation Functions	−	+
	8-parallel lookup	+	+
RC6	Modulo Arithmetics	○	+
	Extended Shifting Capabilites	+	+
Rijndael	Additional Registers	+	+
	Swap Operations	+	+
	Fold-Operations	○	+
	Bit-Mingle Operations	−	+
IDEA	128-bit Shifter	−	+
	Extended Modulo Arithmetics	−	+
Hashes	Modulo Addressing	+	+
	>5-way Parallelism	−	○−
	Boolean & Not Operation	+	+

Table 2: Weigthing of Requirements

Recalling SHA1 data flow, the >5-way parallelism demanded by this algorithm is not strictly necessary. Although SHA1 certainly would benefit from multiple parallel units, restricting the architecture to the parallelism scheme shared by all other algorithms is not prohibitively limiting. Therefore, this requirement can already be safely denied for the sake of a more streamlined architecture.

Other requirements were weighted considering current and future importance of individual algorithms. This instantly led to the removal of any IDEA-specific requirements: because of the great differences to the other investigated crypto algorithms and based on the fact that IDEA is rarely used for real-time stream cipher but rather for securing email correspondence on a per-message base (PGP), we did not consider any of the IDEA requirements for the proposed architecture.

With Rijndael becoming the Advanced Encryption Standard, there is also no need for specific RC6 support. For this reason we dropped the modulo arithmetics from our architecture. The extended shifting capabilities, however, remain since they are also needed for DES, AES/Rijndael, and the hashing algorithms.

Algorithm analysis revealed, that certain supporting functions are necessary for efficent AES/Rijndael and DES implementation. The special functions required for AES/Rijndael might be of some use to other algorithms, but the DES-specific permutation functions can be only used for this one algorithm. Thus, we first decided to condense the DES-

specific functions into one big parameterizable function which performs all necessary DES operations, i.e. a series of permutations followed by an 8-parallel memory access.

Because of this, and the fact that no further interaction with other ALU resources is needed, we removed this function from the ALU. Instead, we placed it right inside the memory unit next to the data memory. This not only keeps the ALU free of too algorithm specific functionality, but also enables this unit to perform latency-free single cycle memory accesses.

4   Resulting Architecture

We started with a basic architecture model: an ALU holding the register file and the necessary arithmetic operations, the memory unit consisting of data memory and an address generation unit. Targeting a clock rate of 400MHz, we decided to restrict memory access strictly to special registers. Therefore, the memory address is provided by address registers. Data read from or written into the memory must pass buffer registers. This model was adopted and extended to fulfill all needs of the cryptographic algorithms as described in the previous chapter which we summarize as follows:

1. The architecture consists of two individual strands. Each strand contains an arithmetic unit, a memory unit, and associated data memory. Limited value exchange/forwarding between the strands is possible through a simple interlink mechanism.

2. Operand and data bus size are 64 bit. A "split–mode" exists which enables use of 64-bit resources as two 32-bit resources supporting 32-bit algorithms.

3. Unlike typical ALUs, our arithmetic unit dissects into the main ALU and a dedicated XOR unit. The ALU follows a standard three-address architecture; the XOR unit, however, allows up to five inputs. By doing so, not only further fine-grain parallelism can be exploited, but also can the XOR unit serve as a simple register copy unit, if needed. The ALU's register file holds four 64-bit registers which may act as eight virtual 32-bit registers. In addition to instructions supporting the combination and extraction of 32-bit halves into or from 64-bit registers, certain AES-supporting instructions which perform more sophisticated select/merge operations were added.

4. The memory unit's transfer/buffer registers may also be used as auxilliary ALU registers. Also, the memory unit holds the DES unit. The memory unit's address generation unit supports standard addressing modes and a special 8-parallel access for packed 8-bit vectors. Standard modes can be combined with a modulo operation for easy 8-bit table wraparound.

5   Summary

In this paper we presented our approach for developing a programmable crypto processor. We explained the need for specific crypto hardware in general and the benefits of a programmable crypto architecture in particular. We then presented a set of algorithms

and the reason why these were chosen; following this presentation we discussed the architectural needs of the selected algorithms. These findings were then weighted against the current and projected future use to determine the architectural parameters and the resulting architecture was sketched. Finally, we gave an application example: AES-128 was implemented on our architecture making use of several architectural features resulting from the algorithm analysis.

References

[BGP93] Brown, R., Good, M., und Prabhakar, A.: 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).

[BHO04] Buchty, R., Heintze, N., und Oliva, D.: The Cryptonite Crypto Processor. *to appear in ARCS'04 Conference Proceedings*. March 2004.

[BMA00] Burke, J., McDonald, J., und Austin, T.: 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.

[BP93] Brown, R. und Prabhakar, A.: FIPS180-1: Secure Hash Standard (SHA). *Federal Information Processing Standards Publication (FIPS)*. May 1993. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[Bu02] Buchty, R.: *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.

[DR01] Daemen, J. und Rijmen, V.: Advanced Encryption Standard (AES) (FIPS 197). Technical report. Katholijke Universiteit Leuven / ESAT. Nov 2001. http://csrc.nist.gov/-publications/fips/fips197/fips-197.pdf.

[EJ01] Eastlake, D. und Jones, P. RFC3174: US Secure Hash Algorithm 1 (SHA1). September 2001. http://www.ietf.org/rfc/rfc3174.txt.

[OBH03] Oliva, D., Buchty, R., und Heintze, N.: AES and the Cryptonite Crypto Processor. *CASES'03 Conference Proceedings*. S. 198–209. October 2003.

[Ri92] Rivest, R. RFC1312: The MD5 Message-Digest Algorithm. April 1992. http://www.ietf.org/rfc/rfc1321.txt.

[RRSY98] Rivest, R., Robshaw, M., Sidney, R., und Yin, Y.: The RC6^{TM} Block Cipher. August 1998. http://www.rsasecurity.com/rsalabs/rc6/.

[RRY00] Rivest, R. R., Robshaw, M., und Yin, Y.: The Case for RC6 as the AES. May 2000. http://www.rsasecurity.com/rsalabs/rc6/.

[Sc96] Schneier, B.: 13.9: IDEA. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*. S. 370–377. 1996. ISBN 3-89319-854-7.

[WWA01] Wu, L., Weaver, C., und Austin, T.: Cryptomaniac: A fast flexible architectore for secure communication. In: *28th Annual International Symposium on Computer Architecture (ISCA 2001)*. June 2001.