Leveraging the Multiprocessing Capabilities of Modern Network Processors for Cryptographic Acceleration*

Gunnar Gaubatz and Berk Sunar
Cryptography & Information Security Laboratory
Worcester Polytechnic Institute, Massachusetts, U.S.A.
{gaubatz,sunar}@wpi.edu

Abstract

The Kasumi block cipher provides integrity and confidentiality services for 3G wireless networks, but it also forms a bottleneck due to its computational overhead. Especially in infrastructure equipment with data streams from multiple connections entering and leaving the network processor the critical performance issue needs to be addressed. In this paper we present a highly scalable bit sliced implementation of the Kasumi block cipher for the Intel IXP 28xx family of network processors. It can achieve a maximum theoretical encryption rate of up to 2 Gb/s when run in parallel on all 16 on-chip microengines.

1 Introduction

The security of third generation wireless communication devices is ensured through the use of strong cryptography in a modular and extensible framework specified by the 3GPP working group [3]. The centerpiece of this security architecture that consists of the algorithms $f8$ (confidentiality) and $f9$ (integrity) is the block cipher Kasumi [4]. Cellular network base stations need to provide the capability of processing multiple *Radio Network Layer* (RNL) data streams at high rates that can be multiplexed into high-speed backbone links [6]. This processing also involves security functions which due to their computational complexity form a bottleneck. An efficient strategy for encrypting multiple data streams in parallel is hence required.

Network processing units (NPU) form the centerpiece of data and voice network routers. They typically contain multiple programmable packet processing engines optimized for TCP/IP traffic. Frequently 16 or more of these engines are present on a single chip in order to provide high processing throughput on the order of tens of Gb/s. Only recently, however, manufacturers have addressed the aspect of network security by providing cryptographic accelerators that can sustain high data rates in the form of either on-chip, flow-through or look-aside co-processors. In this paper we are concerned with providing fast encryption and decryption capabilities to network processors that have already been deployed in the field without any form of hardware acceleration. To do so we rely on two concepts: (1) on the explicit parallelism of the aforementioned multiple programmable packet processors, and (2) on an efficient data representation that enables software parallelization of the encryption procedure.

*This work was supported by a grant from Intel Corporation, U.S.A.

2 Microengine Architecture

Microengine architectures differ significantly from modern general purpose processors, since the primary intended use is for packet processing and not for computationally intensive tasks like cryptographic operations. The high number of parallel microengines necessitates certain trade-offs due to silicon area constraints. Typical architectures possess neither caches nor super-scalar out-of-order execution units. They do, however, possess large general purpose register files, special purpose memory transfer registers and low-latency local memory. As such the optimization process for typical applications differs significantly from that for general purpose processors. The allocation of variables, for example, to either registers or various types of RAM has a tremendous influence on the performance and needs to be thoroughly considered during the planning phase. The target of our implementation is the Intel IXP 2xxx family of network processors [5], although the same principles should apply similarly to other NPU architectures. The NPUs in this family contain either 8 or 16 RISC-based microengines operating at clock frequencies of up to 1.4 GHz (IXP 2800). Each microengine contains 8 Kwords of instruction RAM and two banks of 128 general purpose registers each. 2560 bytes of 32-bit addressable low-latency local memory serve as temporary storage for larger data structures. External memory controllers offer larger storage capacity, however, latencies of up to 300 clock cycles may be incurred upon accessing these slow memories.

3 The Kasumi Block Cipher

The design of the Kasumi block cipher is based on a similar cipher design called Misty1 [7]. Both were developed by Mitsuru Matsui and share most of the design principles that provide strong resistance to linear and differential cryptanalysis. They encrypt blocks of 64 bits using a Feistel network of eight rounds. In each round, alternating between the left and right halves of the state, one half enters the outer round function FO() for processing. The output of the round function is then XOR'ed to the opposite half of the state and operation continues with the next round. FO() itself consists of three rounds of an inner round function FI() in a similar recursive Feistel structure with half the block size. In a third level of recursion FI() itself is composed of either three (Misty1) or four (Kasumi) rounds of nonlinear S-Box transformations arranged in a Feistel structure. Both ciphers require 128 bits of secret key material which is expanded into 64 16-bit round keys. The main difference between Kasumi and Misty1 is a slightly altered S-Box definition, a simplified key schedule facilitating efficient hardware implementations and the mode of using the additional sub function FL().

4 Kasumi on IXP 28xx Microengine

The main goal of our implementation of Kasumi on the Intel IXP 28xx was to maximize the throughput while making efficient use of the available resources of a single microengine. The latency of off-chip memory access is prohibitively high, which mandates that all variables have to fit into the available registers and local memory. Such memory placement optimization was applied to the example C-implementation given in the appendix of the Kasumi specification [4], resulting in an 11-fold speed-up. To improve the performance yet further, however, alternative concepts must be explored. Upon closer inspection of the Kasumi specifications we made some interesting observations. The S-Boxes S7 and S9 can be specified as look-up tables, but they are also specified in terms of single-bit operations. Can we somehow compute the S-Box substitution more efficiently than using table look-up? Look-up table operations require the isolation of the 7- or 9-bit index from the state using masking operations, shifting and memory pointer initializations. Can we avoid costly bit-manipulations and pointer set-ups, and rather spend cycles on the actual operation? Look-up table sizes use up valuable space in local memory that might be better used otherwise, while only a single block of data is produced every eight rounds. Can we store multiple data blocks in local memory and process them in parallel?

4.1 Bit sliced Data Representation

In 1997 Eli Biham presented a fast software implementation of DES [2] using a technique that is now commonly referred to as "bit slicing". The scheme can be visualized as follows: In a processor with a native word size (data-path width) of w bits data is typically processed w bits at a time. If an operation is to only manipulate specific single bits of a word, this requires additional operations in order to isolate these bits, modify them, and finally re-assemble the word. By collecting the data of up to w words of data in a $w \times w$-bit matrix and computing the transpose, it is possible to allocate all bits of the same significance into w separate words. Single bit manipulations can now be performed in parallel on w blocks of data with a single instruction. This w-fold single-bit SIMD operation can help to speed up algorithms that require a lot of costly bit manipulations, such as typical block ciphers. We took the same basic idea and implemented a bit sliced variant of Kasumi on the IXP microengine. The feasibility of this approach ultimately depends on the types of operations in a given algorithm. Obviously there is a trade-off involved since, for example, S-Box substitutions cannot be performed as table look-ups anymore and data conversion from regular to bit sliced representation introduces additional overhead before and after encryption. Bit slicing is also not applicable to algorithms that mostly consist of arithmetic operations, e.g. RC5, RC6 or MARS. Certain bit-level operations, on the other hand, can simply be ignored in the bit sliced representation since the same effect can be achieved through register renaming.

4.2 Efficient Data Conversion

The conversion overhead due to bit slicing is the computation of four transposes per data block (32×64 bits of data split into two matrices, one transpose each before and after encryption). For each new set of keys the setup phase of the key schedule also requires the computation of four transposes due to the 128-bit keys. It is therefore crucial to use an efficient conversion algorithm. Algorithm 1 below is used to compute the transpose of a 32×32-bit block of data. It is loosely based upon the BitReversal algorithm often encountered in applications computing the Fast Fourier Transform. A similar algorithm was published in [8] under the name SwapMove. It is optimized specifically for binary matrices organized in a 32-word array and runs in time $\Theta(n \log_2 n)$. Our implementation was written in microcode, the IXP equivalent of assembly language, in order to minimize conversion overhead. It recursively computes the transpose of a 32×32-bit data block by dividing the matrix into four equally sized sub-matrices in each step and swapping the two quadrants on opposite sides of the main diagonal. The algorithm proceeds until sub matrices only contain a single bit, which is the case after five recursion levels. All data manipulation happens in-place in local memory, thereby reducing the memory footprint significantly. The integrated barrel-shifter of the microengine's ALU is particularly helpful for this algorithm which takes around 600 clock cycles for a 32×32-bit block of data.

Algorithm 1 Algorithm for computing the transpose of a binary matrix

```
Input:  $A$            ▷ $n \times n$-bit Matrix in form of a linear array
Output: $A^T$
 1: procedure PRECOMPUTATION
 2:     $\mathsf{mask}_0 \leftarrow (55\dots5)_{16}$
 3:     $\mathsf{mask}_1 \leftarrow (33\dots3)_{16}$
 4:     $\mathsf{mask}_2 \leftarrow (FF\dots F)_{16}$
 5:     $\dots$
 6:     $\mathsf{mask}_j \leftarrow (\{0\}^{2^j}\{1\}^{2^j})_2$
 7: end procedure

 8: procedure BINARYMATRIXTRANSPOSE(A)
 9:     for $j$ from $0$ to $(\log_2 n) - 1$ do
10:         $k \leftarrow 2^j$
11:         for $i$ from $0$ to $n-1$ do
12:             $l \leftarrow 2(i - (i \bmod k)) + i \bmod k$
13:             temp $\leftarrow (A_l \wedge \mathsf{mask}_j) \vee (A_{l+k} \wedge \mathsf{mask}_j) << k$
14:             $A_{l+k} \leftarrow (A_{l+k} \wedge \mathsf{mask}_j) \vee (A_l \wedge \mathsf{mask}_j) >> k$
15:             $A_l \leftarrow$ temp
16:         end for
17:     end for
18: end procedure
```

4.3 S-Box Optimization

Kasumi uses two differently sized non-linear substitution functions $S_7 : \{0,1\}^7 \to \{0,1\}^7$ and $S_9 : \{0,1\}^9 \to \{0,1\}^9$. In [7] Matsui only gives partial information about the exact selection of parameters for their construction. Both $S_7 : x \mapsto x^{81}$ and $S_9 : x \mapsto x^5$ are almost perfectly non-linear power functions over the fields GF(2^7) and GF(2^9) that have a compact logic description with algebraic degree of 3 and 2, respectively [1]. In the algorithm specification of Kasumi the set of logic equations for both functions are given in two-level XOR sum-of-products form. The intent was to give hardware implementors a circuit definition with a short critical path in addition to the contents of the look-up table. The same definition can be used for our bit sliced software implementation. But since we are computing all the terms sequentially, there is no reason why we have to implement the equations in two-level XOR-SOP form. Basic multi-level logic optimization techniques can be used to minimize the overall number of operations needed to compute all values of the S-Boxes. By applying the distributive law to the product terms we can isolate common factors. Often this leaves us with terms similar to $x_i(x_j \oplus 1)$ which include constant terms. These can be rewritten as $x_i \neg x_j$. Just by applying simple optimizations like these we can achieve an overall reduction of 35% in the number of instructions.

4.4 Simplified Bit-Operations and Key Schedule

Bit sliced representation of data has positive ramifications throughout the Kasumi algorithm. Shifts and rotates by a fixed-amount, truncations, zero-extensions and permutations, which otherwise would require costly logic and shift operations, can now be emulated using clever operand indexing. The large number of registers per microengine is tremendously helpful for an efficient implementation that can afford only a small amount of local memory, especially since the amount of data is about 32 times of that in a non-bit sliced variant. Not surprisingly, resource management is a considerable challenge, especially in comparison to other typical network processors applications. A few short examples help demonstrate the benefits: A frequently encountered operation in Kasumi is rotation of 16-bit data, which is not available as a native instruction. In bit sliced notation we simply modify the index of the bit slice modulo 16 and rotate implicitly. In FI() the 16-bit data path is split into a 7- and a 9-bit half. XOR operations between the two halves require truncations and zero-extensions involving bit-level masking operations. In bit sliced notation these can be simply ignored.

Being able to address single bit slices separately also helps to reduce storage requirements of the key schedule. A total of eight 16-bit sub-keys are derived from the main key K and a related key K' in every round by means of bit-rotation. Regular implementations pre-compute the entire key schedule ahead of time, consuming 128 bytes. Doing the same in the bit sliced variant with 32 keys would require 4096 bytes of storage. But since the sub-keys are related to the main key and its derivative up to a rotation, we only require a quarter of the space. The rotation can be achieved through index manipulation.

5 Implementation Results

In Table 1 we compare the performance of our implementation to highly optimized software implementations of MISTY1 on Alpha and Pentium III processors [10, 9]. A direct comparison, however, is misleading due to the profound architectural differences between these two superscalar processors and the cache-less single-issue IXP microengine. Considering that the Alpha's data path is twice as wide as that of an IXP microengine, it seems surprising that a single microengine of the IXP can achieve throughput rates of only a small factor below those high-end processors. One reason is the relatively high clock frequency of 1.4 GHz, which is much higher than the 500, 667 and 800 MHz used in the Alpha and Pentium III benchmarks. The disadvantages associated with the simplicity of the microengine architecture turn into benefits when we make use of the multi-processing capabilities of the IXP architectures. The high-end model IXP 2800 contains 16 microengines on the same chip that can operate completely independent of each other. Since our Kasumi implementation uses local memory exclusively, memory bottlenecks can be avoided and hence the performance scales linearly. The maximum theoretical throughput therefore reaches an unparalleled 2 Gb/s performance. Obviously this is only a theoretical figure, since there are other tasks that need to be performed

	Encryption (cycles/block)		Key Schedule (cycles/key)		Max. Throughput (Mb/s)	
	reg.	bit sliced	reg.	bit sliced	reg.	bit sliced
Kasumi IXP single ME	2,333	708	1,507	101	38	126
Kasumi IXP 16 MEs	2,333	708	1,507	101	614	2,010
MISTY1 Alpha 21164 [10]	305	111	n/a	n/a	105	288
MISTY1 Alpha 21264 [9]	197	68	200	17	217	601
MISTY1 Pentium III [9]	207	169	230	46	247	303

Table 1. Comparison of Implementation Results

by a network processor in addition to encryption. Nonetheless, this result shows the advantage of highly parallel on-chip multi-processors for specialized tasks. Incidentally, this multi-processing approach taken by network processor vendors bears a certain resemblance to that taken by Sony, IBM and Toshiba with their latest Cell Processor architecture [11].

5.1 Modes of Operation and their Implications

The Kasumi algorithm by itself is merely a single building block in the concept of 3G wireless security. The algorithm $f8$ for confidentiality is based on Kasumi operating in a variation of output feedback (OFB) mode. The integrity algorithm $f9$ computes a MAC on the input message as a variant of standard CBC-MAC mode. In both modes the output of one iteration of the cipher is XOR-ed onto the input of the next iteration. This inherently sequential process makes it impossible to use the bit slicing technique developed in this paper for acceleration of a single data stream. A network processor, however, is designed to handle hundreds of data streams in parallel, each with different security settings and keys. We can simply collect the working sets of up to 32 different data streams into one session of bit sliced Kasumi, and take advantage of the performance benefits. Using the performance results from Table 1 we can determine the break-even point for the minimum number of sessions required such that data transposition offer a performance advantage over a regular Kasumi implementation. The encryption time of 708 cycles per block refers to the best case of 32 parallel data sets, and we therefore break even with at least $N_{\min} \geq \lceil \frac{32 \cdot 708}{2333} \rceil = 10$ parallel sessions.

6 Conclusions

In this paper we presented platform specific details and performance evaluation of our Kasumi block cipher implementation on the Intel IXP 28xx family of network processors. We demonstrated how a computationally intensive cryptographic algorithm can be implemented efficiently and in a scalable manner despite the architectural limitations of a packet processor microengine that was not designed with the intention of running such algorithms. Our implementation achieves a throughput of more than 126 Mb/s on a single microengine. The theoretical maximum that can be achieved on 16 microengines running in parallel is thus close to 2 Gb/s. The bit slicing technique allows us to achieve a more than three times higher data rate than with standard data representation. Due to an efficient algorithm for conversion between data representations the performance penalty remains negligible. Finally, we showed that the problems typically associated with OFB modes of operation do not prevent the application of the bit sliced method in a network processor setting in which multiple independent data streams need to be processed simultaneously.

References

[1] 3GPPSAGE. KASUMI evaluation report. SAGE v. 2.0, 3GPP, Oct 2000.

[2] E. Biham. A fast new DES implementation in software. In E. Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 260–272, Heidelberg, Germany, Jan 1997. Springer Verlag.

[3] ETSI/SAGE. 3GPP confidentiality and integrity algorithms; document 1: f8 and f9. 3GPP TS 35.201, ETSI, Sophia-Antipolis Cedex, France, Dec 1999. Draft.

[4] ETSI/SAGE. 3GPP confidentiality and integrity algorithms; document 2: KASUMI. 3GPP TS 35.202, ETSI, Sophia-Antipolis Cedex, France, Dec 1999. Draft.

[5] Intel Corporation. *Intel IXP 2800 Network Processor Hardware Reference Manual*, May 2004.

[6] Intel Corporation. A modular approach to radio network controller design using ATCA and Intel IXP2xxx network processors. http://www.intel.com, Feb 2004.

[7] M. Matsui. New block encryption algorithm MISTY. In E. Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 54–68, Berlin, Jan 1997. Springer-Verlag.

[8] L. May, L. Penna, and A. Clark. An implementation of bit-sliced DES on the pentium MMX processor. In *ACISP*, volume 1841 of *LNCS*, pages 112–122. Springer-Verlag, 2000.

[9] J. Nakajima, T. Ichikawa, and T. Kasuya. Cipher algorithm implementation. In *MEA: Cryptography Edition*, volume 100, pages 13–17. Mitsubishi Electric Co., Dec 2002.

[10] J. Nakajima and M. Matsui. Fast software implementations of MISTY1 on alpha processors. *IEICE Trans. Fundamentals*, E82-A(1):107–116, Jan 1999.

[11] D. Pham et al. The design and implementation of a first-generation CELL processor. In *Int. Solid State Circuits Conference*. IEEE SSCS, IEEE Press, Feb 2005.