DESIGN OF A FLEXIBLE CRYPTOGRAPHIC HARDWARE MODULE

Andrew W. H. House and Howard M. Heys
Electrical and Computer Engineering
Memorial University of Newfoundland
St. John's, NL A1B 3X5, Canada
Email: {ahouse, howard}@engr.mun.ca

Abstract

Key elements of communication security include block ciphers and hash functions. Software implementations of these algorithms are relatively easy, but may not provide the speed necessary for some applications. For such applications, hardware implementations in ASIC or FPGA technology provide speed, but are difficult and time-consuming to develop. The SHERIF architecture described in this paper seeks to provide a flexible cryptographic hardware platform targeted for 0.18 micron CMOS technology, capable of greater speeds than software implementation while allowing for an ease of implementation not found in traditional hardware environments.

A survey of major cryptographic algorithms was performed to identify a small number of common operations, and components to implement each of those operations were designed and implemented so as to be configurable. A suitable arrangement of the basic components was finalized to facilitate data flow, and a key management solution was devised to handle provision of subkeys to the appropriate components. The system is fully configurable and is intended to support multiple algorithms such as AES, DES, RC6, and Camellia, as well as different modes of operation.

The VHDL model of the system has been successfully simulated, and synthesis is in progress to produce area and timing information.

Keywords: Cryptographic hardware; reconfigurable hardware; cryptography; computer architecture.

1. INTRODUCTION

The ubiquity of modern telecommunications has expanded the need for security beyond the traditional scope of military, government, and financial institutions. As more people and businesses incorporate communications technology into their daily routines, the volume of sensitive information being transmitted over public networks such as cellular phone networks or the Internet has increased dramatically. Furthermore, as communication capabilities increase, the desire for secure high-bandwidth applications further drives the need for hardware-based security solutions.

Data encryption via block ciphers and authentication via hash functions are important parts to solving the security problem [1]. Traditionally, the easiest means of implementing such algorithms has been in software. For applications in which plenty of computing power is available, or speed of transmission is not a concern, software is an ideal means of implementation, offering flexibility and upgradability. However, applications requiring high speed or high throughput typically need a dedicated hardware solution (such as a custom ASIC or FPGA). The major deficiencies of such hardware solutions is that they are difficult and time-consuming to develop, and offer little in the way of flexibility.

This dichotomy provides the motivation for this research, which seeks to develop a middle ground between software implementation and custom hardware implementation. The goal of the **SHERIF** (Security Hardware Enhanced for Rapid Implementation and Flexibility) project is to provide a cryptographic hardware module targeted to 0.18 micron CMOS technology which can be easily configured to implement common block cipher and hash function algorithms, while providing greater speed than software implementation. This has resulted in the design of a novel architecture optimized for implementing rounds of block ciphers and hash functions.

2. ALGORITHM ANALYSIS

To determine the components needed for the SHERIF architecture, a number of possible algorithms that might be implemented on the platform were analyzed. The analysis focused on algorithms that were current standards or were candidates in the NESSIE (New European Schemes for Signatures, Integrity, and Encryption) standards process [2]. A set of algorithms was selected that was considered

to be representative of modern block ciphers and hash functions, while still demonstrating significant variation.

The algorithms considered were the Advanced Encryption Standard (AES) [3] and its predecessor Data Encryption Standard (DES) [4], hash functions SHA-1 [5] and MD5 [1] (which are the basis for the more recent standardized versions of SHA), and NESSIE block cipher finalists Camellia [6], RC6 [7], SAFER++$_{128}$ [8], and SHACAL (based on the SHA hash function) [9]. Note that Camellia and SHACAL-2 were selected as recommended algorithms by NESSIE [10].

Analysis of these algorithms in [11] provided a list of common operations: basic Boolean logic such as XOR, AND, OR, and NOT; bitwise rotations or shifts; addition and subtraction with varying sizes of operand; modular multiplication; substitutions (S-boxes or look-up tables); byte permutations; bitwise permutations; multiplication in a Galois Field; bit expansions; and other linear transformations (like the P() function in Camellia [6]). This small common set of basic operations provides the basis for the further development of our architecture.

3. COMPONENT DESIGN

Implementation of the basic operations was not a trivial task, even though many of the operations are very simple. The different algorithms may all use the same operations, but they require different operand sizes, different sequences, and different quantities of each operation. Thus, it was necessary to reduce each component to a configurable core logic block that could form the basis of the rest of the architecture, and so six major components were designed. Note that a 128-bit block size was selected, since it is a standard size for most modern block ciphers. Unfortunately, this limits SHERIF's ability to implement hash functions, many of which require larger block sizes.

3.1 Boolean Logic Component

This component groups the four Boolean operations together. It takes two 16-byte inputs, and produces one 16-byte output based on a bitwise Boolean operation. The Boolean operation (AND, OR, NOT, XOR) for each pair of corresponding input bytes is selectable. This component, though simple, provides a great deal of functionality at little cost.

3.2 Shifter Component

The shifts and rotations used in any of the considered algorithms are all on 32-bit words, so this is the obvious size to use for the shifter elements. Thus, this component takes four 32-bit inputs (one to each of four shifters), and four corresponding 5-bit shift amount inputs. The shifters themselves can be configured for left/right and rotate/shift-in-0. Shifting is done in one clock cycle, since the shifter is implemented via layered multiplexers.
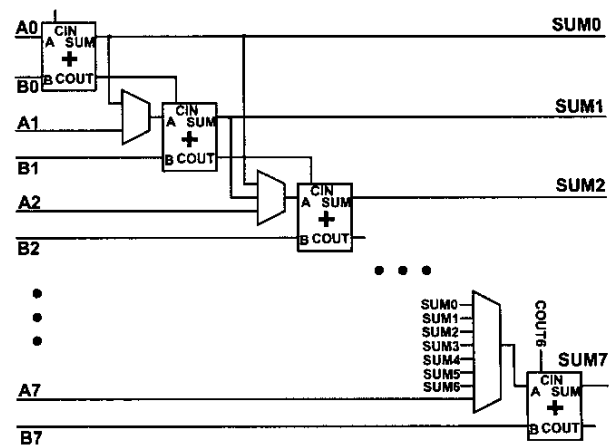

Fig. 1. Half of add/subtract component.

3.3 Add/Subtract Component

The addition/subtraction component proved to be difficult to design, since the different algorithms required modular addition/subtraction on different sizes of operands. The natural inclination is to use adders of the largest size needed, which can of course implement smaller additions. However, this does not account for the large discrepancy in the *quantity* of additions needed – SAFER++$_{128}$ [8] requires over four-hundred 8-bit additions [11], and providing that many 32-bit adders (as needed by RC6, say [7]) would be prohibitive.

Thus, it was decided to base the design around fast 8-bit adder components (which can also be used to perform 2's complement subtraction). The basic 8-bit adders can be cascaded together to build larger adders. Furthermore, SAFER++$_{128}$ [8] requires additions in sequence – that is, the output of one 8-bit addition is input into the next 8-bit addition. Rather than require the use of multiple add/subtract components to implement this scheme, the design incorporates this type of data routing as an option internal to the component.

The add/subtract component consists of 16 8-bit adders, arranged in two banks of 8 adders each. The adders in each bank may be cascaded together to form up to one 64-bit adder, two 32-bit adders, and so on. Furthermore, in each adder bank, every adder after the first can either use the datapath input as one of its inputs, or else use the output of any preceding adder in the same bank. Subtraction is configured in the same way. An example of one bank of adders is shown in Figure 1. Note that the control signals that select between addition/subtraction and connected/unconnected are not shown in the figure.

3.4 Multiplier Component

Of the algorithms considered, only RC6 [7] uses multiplication. However, since it is a fairly fundamental operation, it is conjectured that new algorithms might incorporate multiplication, and so a multiplier component is included despite its current limited use.

The multiplier component contains four 32-bit Wallace tree multipliers, each of which performs modulo 2^{32} multiplication on two 32-bit inputs to produce a 32-bit output. This architecture was based upon the Wallace tree multiplier used in [12], since it provides a good compromise of speed and complexity. Note that the multiplier executes in a single clock cycle.

3.5 LUT Component

This component performs substitution operations. While many modern algorithms may define their substitutions mathematically as Boolean functions, in the general case it is easiest to implement such substitutions (or S-boxes) as a look-up table (LUT). For our LUT component, an array of sixteen 8 × 8 LUTs is used, which is sufficient for any of the algorithms under consideration. Each LUT takes an 8-bit input and produces the corresponding 8-bit substitution value as output.

Normally, LUTs would be implemented as memory. However, to avoid the latency introduced by memory, the LUTs have been implemented as banks of flip-flops which are selected via a large multiplexer. This contributes significantly to the area of the device in test synthesis, but is adequate for prototyping purposes.

3.6 XORnet Component

The final component serves many purposes, such as implementation of Galois field multiplication by a constant (the MixColumn() operation of AES [3]), bitwise permutations, bit expansions, and other bitwise manipulations and linear transformations (such as Camellia's $P()$ function [6]).

The XORnet component operates such that each output bit is generated from the XOR of any number of the input bits. The logic to generate a single output bit is shown in Figure 2. The structure is simply repeated in parallel to produce the output for each desired output bit, so a 32-bit XORnet would repeat the structure in Figure 2 32 times.

It is shown in [13] that multiplication by a constant in a Galois field can be implemented in such a fashion. Such operation also mirrors the definition of Camellia's $P()$ function [6]. With respect to bitwise permutations and expansions, it is trivial to see that the XORnet can implement these by having the output bits consist of the XOR of only the desired input bit to be permuted/expanded to that output.

The major difficulty introduced by this component is that it requires a large amount of configuration data (the
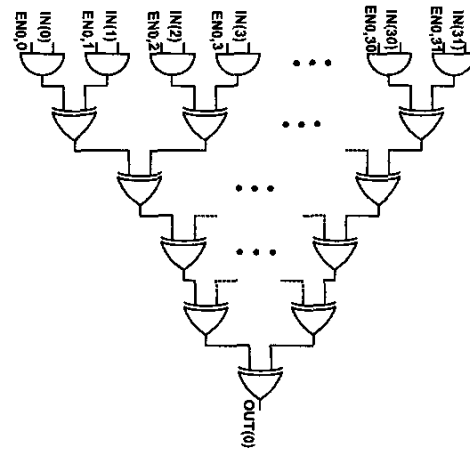

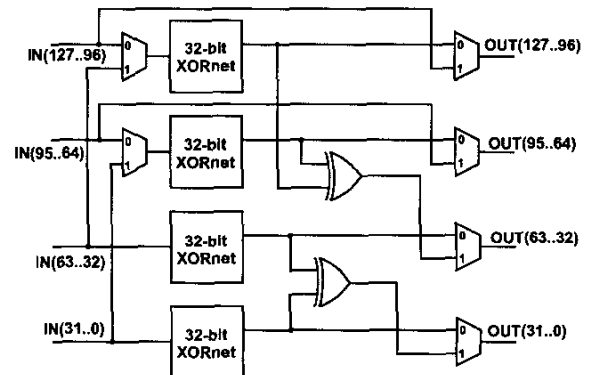Fig. 2. Logic to generate single output bit for XORnet.

Fig. 3. Overall arrangement of XORnet component to allow flexibility.

enable signals seen in Figure 2) -- each output bit requires a number of configuration bits equal to the number of input bits to allow potential selection of any of the inputs. Thus, the memory requirement for storing configuration data for the XORnet becomes a major problem -- a 32-bit XORnet (which has a 32-bit input and 32-bit output) requires 1024 bits of configuration data.

The XORnet component consists of 4 32-bit XORnets in parallel. A 32-bit XORnet is not sufficient for Camellia, however, so the option of combining the 4 32-bit XORnets into a single 64-bit XORnet exists by repeating the 64-bit input across the second two XORnets and then running the outputs of all 4 through an XOR tree to produce a single 64-bit output. This arrangement is shown in Figure 3.

4. PROCESSING ELEMENT DESIGN

The most common means for implementing fast encryption hardware is to use a fully loop-unrolled and pipelined implementation that operates on a full block of data. This sort or architecture allows for high throughput, even at modest clock speeds. The design of SHERIF seeks to mirror this sort of architecture in order to exploit the round-oriented nature of block ciphers and hash functions. This leads to the design of a processing element (PE) meant to implement one round of an encryption algorithm or hash function. This processing element forms the basis of the overall system architecture.

The PE is highly configurable, allowing components to be activated or bypassed, controlling the operation of components, and controlling data routing internal to the PE. The configuration data is set by an external device, and is shifted in to the configuration registers before the system can operate.

Each PE is identical, and consists of a number of instances of the basic components. It is desirable to minimize the number of large components (LUT, XORnet, and multiplier), so they are arbitrarily set in fixed positions in the PE datapath and then surrounded with multiple instances of the smaller components (Boolean, add/sub, shifter). This is shown in Figure 4.

Each basic component above is preceded by an input switch which selects the inputs to the components from the datapath, the scratchpath (a parallel data storage path), pre-set constant values, or key values provided by the key memory. The core logic of the component either operates on the inputs or is bypassed, and the resulting output is then run through a byte reordering component, allowing the output bytes to be reordered into any sequence. The byte reordering consists of sixteen 16×1 multiplexers, allowing duplication of any of the outputs. This overall arrangement is depicted in Figure 5.

The issue of key management is a significant one. While SHERIF is meant to be configured only when the system is initially powered on, the keys used by the algorithms are expected to change relatively frequently. Thus, a key management solution is necessary.

It was decided to assume that an external processor would implement the key schedule and generate the round subkeys, which could then be provided to SHERIF. A key memory component is implemented in each PE to hold this data. The key memory is not trivial, since the PE can be used in one of two ways: either in a loop-unrolled pipelined architecture, or an iterative architecture. If used in a pipelined architecture, the subkey provided will be the same every clock cycle, until a new key is used. However, if used in an iterative architecture, every clock cycle (or every few clock cycles) will require a different subkey to be provided.
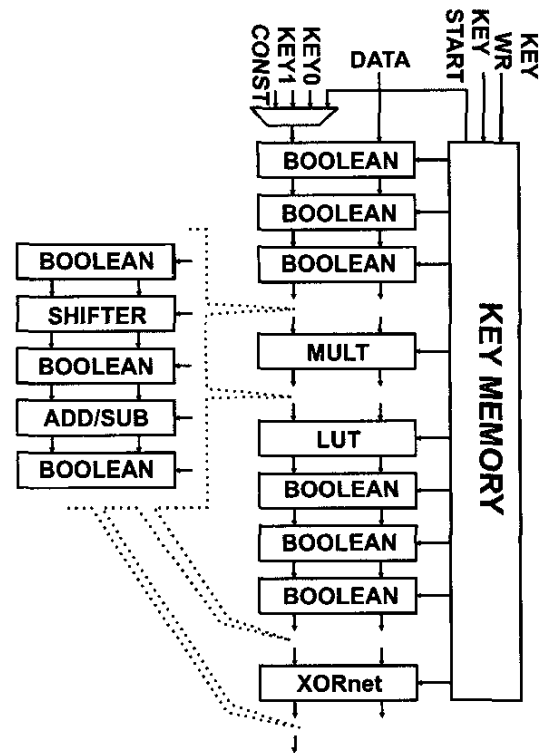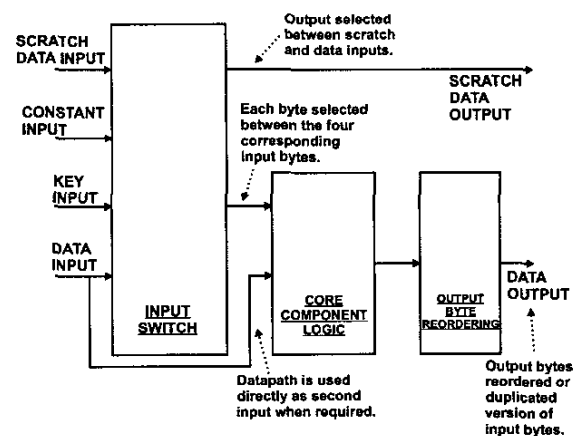

Fig. 4. Internal organization of the PE.

Fig. 5. Input/output switching of basic components.

To provide this capability, the key memory in a PE consists of a large 1664-bit shift register, enough for 13 128-bit subkeys. The subkey data is written into the key memory 128 bits at a time, and all 1664 bits must be written before use. A state machine controls the shift behavior, waiting for a start signal to indicate valid data is being processed, then counting a number of clock cycles before shifting to access the next key data. The shift amount is set in multiples of 32 bits to a maximum of 256 bits in one clock cycle. The number of clocks between shifts and the shift amount are selected by the user in the configuration data. When the state machine reaches the end of its defined behavior, it resets the shift register back to its original values using a backup non-shifting register, and then waits for the next start signal.

The user also configures the key memory to select which of two 128-bit words is to be routed to each basic component. This selection is internal to the key memory, so only the output key data is shown in Figure 4. The key memory can be used for pipelined architectures and iterative architectures (iterating through one or more PEs) due to the configurable waiting times.

5. SYSTEM ARCHITECTURE

With the PE defined, the remaining question is how to incorporate it into a top-level design. Initially, a 2D array of PEs was thought to be desirable, but the timing issues related to communication with adjacent processing elements made this infeasible. A sequential arrangement of PEs more closely resembled the arrangement of rounds of the algorithms themselves, and thus seemed sensible. For the proof-of-concept SHERIF device, ten PEs are instantiated in sequence. This is sufficient to implement all 10 rounds of AES. Other algorithms would be constrained to iterative implementations. As can be seen in Figure 6, the PEs (labeled P0 to P9) are connected by routing nodes (labeled R0 to R10).

The routing nodes are responsible for controlling the flow of data. They allow the user to configure pipelined and iterative implementations of algorithms by selecting how valid data flows between PEs and to the system output. Each routing node has 3 inputs (feedback, datapath, and system) and three corresponding outputs. It is controlled by three internal state machines, each of which is configured by an external device and is triggered by start signals from the address decoder. The states control which data values are put onto each output. The routing node also stores initialization vector (IV) values and feedback values, allowing the implementation of different modes of operation such as cipher-block-chaining (CBC) mode [1].

Other implementations of this architecture could incorporate more processing elements. The address decoder depicted in Figure 6 uses a 5-bit address, allowing it to
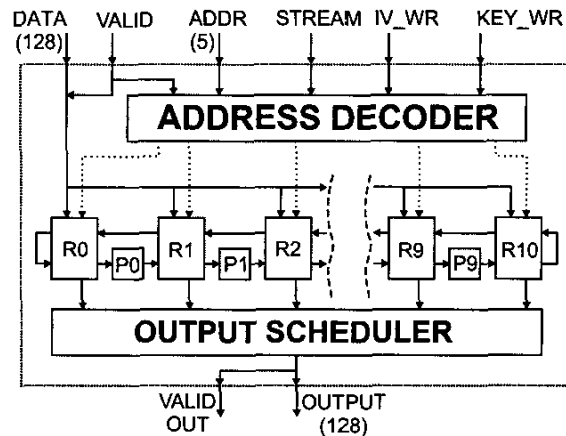

Fig. 6. SHERIF system organization.

scale relatively easily to up to 30 PEs or 31 routing nodes (the address 11111 is a reserved address that writes to every routing node). The address decoder also provides other control signals to each routing node to facilitate the correct flow of data.

The outputs of the routing nodes are sent to the output scheduler. This component stores valid outputs until there is a chance to put them on the system output. It serves the routing node outputs in round-robin order, and is capable of switching outputs every clock cycle. However, it is up to the system user to ensure that on average there will be at most 11 output requests in an 11 clock cycle span, and that there is enough leeway in their design to ensure that a new valid output will not overwrite an existing valid output that has not yet reached the system output.

6. DESIGN DRAWBACKS

The major drawback to the architecture presented here is that it requires a large amount of configuration data. Each processing element needs 45274 bits of configuration data (including the LUTs). Since the configuration data is stored in flip-flops in the current implementation, this lends itself to comprising more than two-thirds of the area of the PE. Since initial synthesis results in a PE with more than 800k gates, this is a major concern. Unfortunately, there is no obvious way to reduce the size of configuration data. Using memory rather than flip-flops to implement the LUTs would help somewhat, but would introduce added latency, and so that option was ignored for this architectural investigation. The large size of the processing element also limits the number that can be incorporated into a system.

The current implementation also has limited pipelining – the inputs and outputs of the processing elements are

registered, as are the routing nodes, but that still leaves large pools of combinational logic which will limit clock speed. Fortunately, even a modest clock speed will allow this architecture to surpass software speeds, and further pipelining could be used to increase clock speed at the cost of latency, and would require only minimal changes throughout the rest of the system.

The last problem is one of redundancy – with all the hardware in a processing element, only a small fraction of it will be used to implement a round of any given algorithm, and the rest will be bypassed. This is inefficient, but unavoidable given the flexibility offered by the current architecture.

7. CONCLUSION

The SHERIF cryptographic hardware module has been implemented in VHDL, and is undergoing functional simulation to verify the correctness of the architecture. Test synthesis on the components has been performed, but synthesis of the whole system is in the process of being optimized. It is still necessary to create several test implementations of selected algorithms and run simulations of them in order to verify the scope of applicability of the SHERIF architecture to the targeted algorithms.

References

[1] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[2] NESSIE Consortium, "NESSIE: New European Schemes for Signatures, Integrity, and Encryption." Online at http://www.cryptonessie.org/.

[3] NIST, "Federal information processing standards publication 197 – Advanced Encryption Standard." Online at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001.

[4] NIST, "Federal information processing standards publication 46-3 – Data Encryption Standard." Online at http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, October 1999.

[5] NIST, "Federal information processing standards publication 180-2 – Secure Hash Standard." Online at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf, August 2002.

[6] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "NESSIE submission: Camellia." Online at http://www.cryptonessie.org/workshop/submissions/camellia.zip, September 2000.

[7] J. Jonsson and B. Kaliski, "NESSIE submission: RC6." Online at http://www.cryptonessie.org/workshop/submissions/rc6.zip, September 2000.

[8] J. L. Massey, G. H. Khachatrian, and M. K. Kuregian, "NESSIE submission: SAFER++." Online at http://www.cryptonessie.org/workshop/submissions/safer++.zip, September 2000.

[9] H. Handschuh and D. Naccache, "NESSIE submission: SHACAL." Online at http://www.cryptonessie.org/workshop/submissions/shacal.zip, September 2000.

[10] NESSIE Consortium, "Portfolio of recommended cryptographic primitives." Online at http://www.cryptonessie.org/deliverables/decision-final.pdf.

[11] A. W. H. House and H. M. Heys, "Preliminary design of a flexible cryptographic hardware module," in *Proceedings of the Twelfth Newfoundland Electrical and Computer Engineering Conference*, (St. John's, Newfoundland, Canada), Memorial University of Newfoundland, November 2002.

[12] M. Riaz, "The hardware implementation of private-key block ciphers," Master's thesis, Memorial University of Newfoundland, St. John's, Newfoundland and Labrador, Canada, 2000.

[13] L. Xiao and H. M. Heys, "Hardware design and analysis of block cipher components," in *Information Security and Cryptology – ICISC 2002*, Lecture Notes in Computer Science 2587, pp. 164–181, Springer-Verlag, 2002.