Vol. 42 No. 4 IPSJ Journal Apr. 2001

*Regular Paper*

# A Novel Dynamically Reconfigurable Hardware-based Cipher

Zaldy Andales,[†,††] Yukio Mitsuyama,[†] Takao Onoye[†††] and Isao Shirakawa[†]

This paper describes a 64-bit-block, 128-bit-key, dynamically reconfigurable hardware-based cipher, called Chameleon, in which two 32-cell, 8-context dynamically reconfigurable hardware units are employed to generate new data-dependent subkeys for each of the 16 iterations in the encryption/decryption process. The proposed architecture has been implemented by means of the 0.6 $\mu$m CMOS 3LM technology, using 65.6 K transistors and attaining a maximum throughput of 317.5 Mbps. The new approach provides distinctive features of enhanced complexity and flexibility, while demonstrating suitability for embedded encryption/decryption applications.

## 1. Introduction

Dynamically reconfigurable hardware devices [1)~8)], which can change their own functional configurations during runtime, are promising candidates for hardware-based encryption/decryption, because their capacity for implementation of bit-serial operations and fast context shifting is superior to that of processor-based approaches. Furthermore, they provide enhanced decryption complexity by means of a number of sophisticated contexts dedicated to encryption.

Recently, various approaches for applying reconfigurable hardware to cryptography have been attempted. For example, a general-purpose Dynamically Reconfigurable Logic Engine (DRLE) [1)] has been devised, which can execute the Data Encryption Standard (DES) [9)] algorithm at a speed one order of magnitude higher than a software implementation. A 133 MHz MIPS processor with a reconfigurable coprocessor, called Garp [2)], is reported to run the DES 24 times faster than a 167 MHz UltraSPARC. An architecture called PipeRench [3)], which runs at 100 MHz by using a dynamically reconfigurable pipeline, implements the International Data Encryption Algorithm (IDEA) [10)] with a throughput of 126.6 Mbytes/s.

However, these conventional cipher schemes with reconfigurable logic facilities are trivial implementations of known software approaches like the DES and IDEA. No cipher scheme has ever adopted an inherently dynamically reconfigurable hardware-based multi-context approach.

To provide specific features of dynamically reconfigurable functions dedicatedly for cryptography, this paper devises a novel hardware-based cipher called "Chameleon," which is intended for embedded hardware encryption/decryption applications.

The rest of this paper is organized as follows. Section 2 outlines the architecture of Chameleon, Section 3 describes the VLSI implementation of Chameleon, Section 4 discusses the distinctive features of the new cipher scheme, Section 5 presents a brief comparative performance evaluation, and Section 6 offers some concluding remarks.

## 2. Chameleon: A 64-bit Block Cipher Algorithm

### 2.1 Overview

Chameleon is a secret-key cryptosystem which operates on 64-bit blocks of data one at a time through the use of a 128-bit secret key. As shown in **Fig. 1**, the operation of Chameleon is similar to that of DES. The main difference consists in the sophisticated data-dependent subkey generation without the use of a Feistel network [11)].

The Chameleon architecture is outlined in **Fig. 2**. The 64-bit input data are divided into two parts: one consisting of the right 32 bits and the other consisting of the left 32 bits. Initially, the 32 bits of data on the right are loaded into register $Y$, while the 32 bits of data on the left are loaded into register $X$.

Each iteration process can be divided into two: the encryption datapath and the subkey

† Graduate School of Engineering, Osaka University
†† Institute of Mathematical Sciences and Physics, University of the Philippines at Los Banos
††† Graduate School of Informatics, Kyoto University

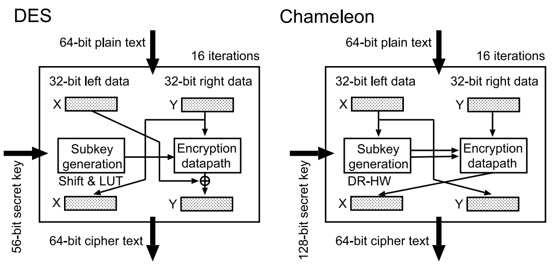


**Fig. 1** DES and Chameleon.

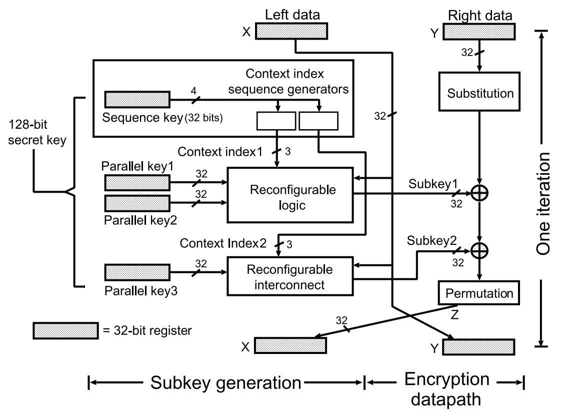


**Fig. 2** Chameleon architecture for encryption.

generation. The former encrypts the data from $Y$, while the latter produces new subkey pairs by using the 128-bit secret key and data from $X$. These subkeys are XORed with the ciphering data in the encryption datapath process.

## 2.2 Encryption Datapath

In the encryption datapath, the data of $Y$ pass through a *Substitution* unit, which consists of 8 S-box tables placed in parallel, where an S-box consists of a 4-bit address × 4 bits of data memory, like that employed in the *GOST algorithm*[12]. As shown in Fig. 2, the output data of this *Substitution* are XORed with a 32-bit subkey (*Subkey 1*), and then XORed with another 32-bit subkey (*Subkey 2*). The resulting data, in turn, are input to a unit of *Permutation*, which rearranges the bit order of the data for use in the next iteration, as indicated in **Fig. 3**.

To complete each iteration, the left 32 bits (in $X$) are input to $Y$, while the iteration result ($Z$) is input into $X$. This iteration is repeated 16 times, with the processes for left and right data executed alternatively, each 8 times.

The final 64-bit encrypted result is obtained by combining the result ($Z$) of the 15th iteration (left 32 bits of cipher text) and the result ($Z$) of the 16th iteration (right 32 bits of cipher text).

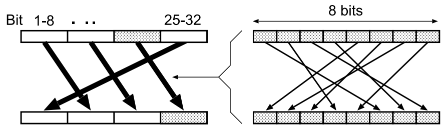


**Fig. 3** Permutation.

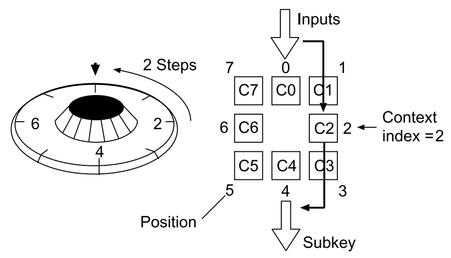


**Fig. 4** Circuit implementation of a rotary dial combination lock (Shifting 2 Steps to Position 2 in the lock corresponds to choosing Circuit $C2$ at Position 2 for subkey generation).

The decryption process is performed in the opposite order to the encryption process. Specifically, the data are first input into a unit of *Inverse Permutation*, next the output is XORed with *Subkey 2*, and then with *Subkey 1*, and finally the resulting data are fed to a unit of *Inverse Substitution*.

Since the encryption and decryption datapaths are distinct from each other, multiplexers are placed in the data loop in order to arbitrate between these datapaths.

## 2.3 Subkey Generation

The subkey generation process is based on the principle of a rotary dial combination lock, which is commonly used in vaults and safes. **Figure 4** shows a dial with 8 positions. To open such a lock, a correct sequence of turning steps must be performed in accordance with a specified secret code/key. In effect, the state of the lock changes at each iteration of the knob setting from one position to another. An 8-context dynamically reconfigurable hardware unit is employed to implement this dial lock, where each context configuration (i.e., each of the circuits $C0, C1, C2, \ldots, C7$) corresponds to a position (0 to 7) on the dial. A 3-bit *Context Index* specifies a context configuration (i.e., $Ck$) to be used, as will be described later.

It can be seen from Fig. 2 that

(1) A 128-bit secret key is divided into four segments such that the first 32-bit key is designated as *Sequence Key*, and the subsequent three as *Parallel Keys 1*, *2*, and *3*.

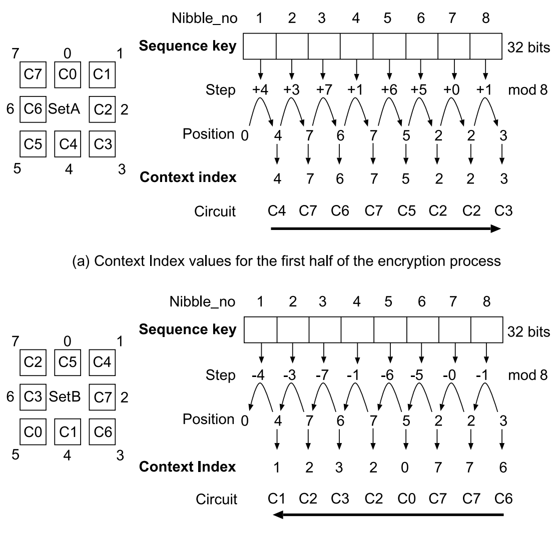
(2) The subkey generation process is performed by two major components, one composed of two *Context Index Sequence Generators* (*CISG*s), and the other of units of *Reconfigurable Logic* (*RL*) and *Reconfigurable Interconnect* (*RI*), where the former *CISG*s are to generate the *Context Indices* which fix a configuration of *RL* and *RI*, and the latter *RL* and *RI* are to produce *Subkeys 1* and *2*, respectively.

The *Context Indices 1* and *2* to be input to *RL* and *RI*, respectively, should be different at each iteration, and therefore two *CISG*s are employed.

The data of *Sequence Key* are loaded into the two *CISG*s, which produce *Context Indices* to be input to *RL* and *RI*. The data of *Parallel Key 1*, *Parallel Key 2*, and *X* together with *Context Index 1* are input to *RL* to be transformed into *Subkey 1*. Similarly, the data of *Parallel Key 3* and *X* together with *Context Index 2* are input to *RI* to be transformed into *Subkey 2*. These key transformations should keep pace with the encryption datapath operations.

### 2.4 Context Index Sequence Generators

32-bit *Sequence Key* is used to produce *Context Indices 1* and *2*, each constituting a sequence of sixteen 3-bit numbers. The mechanism for generating *Context Index* is exemplified in **Figs. 5** (a) and (b), where the first and second halves are generated, respectively.



Nibble_no 1 2 3 4 5 6 7 8
Sequence key
32 bits
C7 C0 C1
C6 SetA C2
C5 C4 C3
Step +4 +3 +7 +1 +6 +5 +0 +1 mod 8
Position 0 4 7 6 7 5 2 2 3
Context index 4 7 6 7 5 2 2 3
Circuit C4 C7 C6 C7 C5 C2 C2 C3
Nibble_no 1 2 3 4 5 6 7 8
Sequence key
32 bits
C2 C5 C4
C3 SetB C7
C0 C1 C6
Step -4 -3 -7 -1 -6 -5 -0 -1 mod 8
Position 0 4 7 6 7 5 2 2 3
Context Index 1 2 3 2 0 7 7 6
Circuit C1 C2 C3 C2 C0 C7 C7 C6

(a) Context Index values for the first half of the encryption process

(b) Context Index values for the second half of the encryption process

**Fig. 5** Sample *Context Index* sequence generation.

**Figure 6** shows the algorithm for generating *Context Index*. Given a 32-bit *Sequence Key*, first consider the generation of the first half of *Context Index*. As shown in Fig. 5 (a), the process of calculating 3-bit *Steps*, 3-bit *Positions*, and 3-bit *Context Index* numbers is executed for *Nibbles* $1, 2, \ldots, 8$, one at a time in this order. Now, consider the generation of the second half. In this case, as shown in Fig. 5 (b), the process of calculating 3-bit *Steps*, 3-bit *Positions*, and 3-bit *Context Index* numbers is executed in the reverse order, that is, first for *Nibble* 8, then for *Nibble* 7, and so forth.

Thus the process flow of *CISG* can be drawn as shown in **Fig. 7**.

In connection with this *Context Index* generation, it should be noticed that

(1) For ease of implementation, *Step* at *Nibble* $k$ $(1 \leq k \leq 8)$ in the second half is

```
Procedure Generation of Context Index(): (encryption)
1  Position = 0, Nibble_no = 0,
   Nibble Array[1 to 8] = array of
      Sequence Key nibbles from left to right
2  repeat (First half)
3    Nibble_no = Nibble_no +1
4    Nibble (= $[b_3, b_2, b_1, b_0]$)
        = Nibble Array[Nibble_no]
5    Step (3 bits) = $\underline{\text{Nibble\_to\_Step}}$ $(b_3, b_2, b_1, b_0)$
6    Position = $\overline{\text{Position + Step}}$ mod 8
7    Context Index = Position
       (set A Position-Context assignment)
8  until (Nibble_no = 8)
9  repeat (Second half)
10   Context Index = Position $\oplus$ $\underline{\text{Transform constant}}$
       (set B Position-Context assignment)
11   Nibble (= $[b_3, b_2, b_1, b_0]$)
        = Nibble Array[Nibble_no]
12   Step (3 bits) = $\underline{\text{Nibble\_to\_Step}}$ $(b_3, b_2, b_1, b_0)$
13   Position = $\overline{\text{Position − Step}}$ mod 8
14   Nibble_no = Nibble_no − 1
15 until (Nibble_no = 0)
```

(a) Algorithm

$$\textit{Nibble\_to\_Step} = \begin{cases} \left[(b_0 \oplus b_2), (\overline{b_3} \oplus b_2), (b_1 \oplus \overline{b_2})\right] \oplus \\ \quad ([b_3, b_3, b_3] \wedge (\textit{Transform constant})); \\ \hfill \textit{for RL} \\ \left[(b_3 \oplus \overline{b_0}), (\overline{b_2} \oplus b_0), (b_1 \oplus \overline{b_0})\right] \oplus \\ \quad ([b_3, b_3, b_3] \wedge (\textit{Transform constant})); \\ \hfill \textit{for RI} \end{cases}$$

$$\textit{Transform constant} = \begin{cases} 101; & \textit{for RL} \\ 011; & \textit{for RI} \end{cases}$$

(b) Nibble_to_Step and *Transform constant*

**Fig. 6** Algorithm for generating *Context Index*.

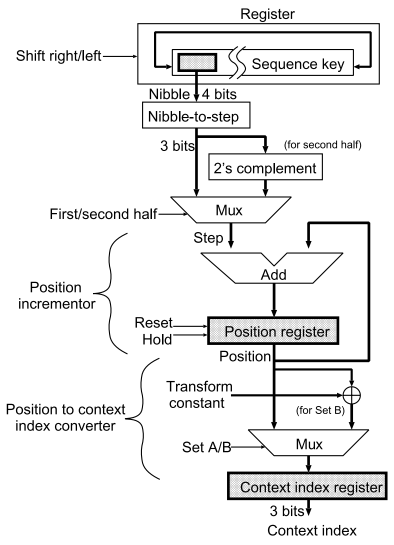


**Fig. 7** Process flow of the *Context Index Sequence Generator.*

equal to the negative value of *Step* at the same *Nibble* in the first half, so that the the values of initial and final positions can both be 0.

( 2 ) The decryption process can be obtained by exactly reversing the order of the encryption process, and hence *Position* always starts and ends at 0 in both the encryption and decryption processes.

( 3 ) *Context Indices 1* and *2* can be obtained through the use of *RL* and *RI*, respectively, in parallel.

Finally, it should be added that a user can specify *Transform constant* at random: here we have fixed its values as 101 and 011 for *RL* and *RI*, respectively.

### 2.5 Reconfigurable Units

Details of the structures of the 8-context *RL* and *RI* are shown in **Figs. 8** (a) and (b), respectively. *RL* consists of an 8 × 4 array of basic cells, each of which is labeled according to the subkey output bit. A basic cell is composed of an 8-context 3-input lookup table (LUT) and an *8:1 Multiplexer*, which has been already adopted in DeHon[6] and Tau, et al.[7] As shown in Fig. 8 (a), the 3-bit data in the three registers (*X*, *Parallel Key 1*, and *Parallel Key 2*) are input to a basic cell. The configuration of *RL* is determined by *Context Index 1*.

In the example shown in the figure, *Context Index 1* addresses "context 3" of the context memory, while the 3-bit input to *8:1 Multi-*

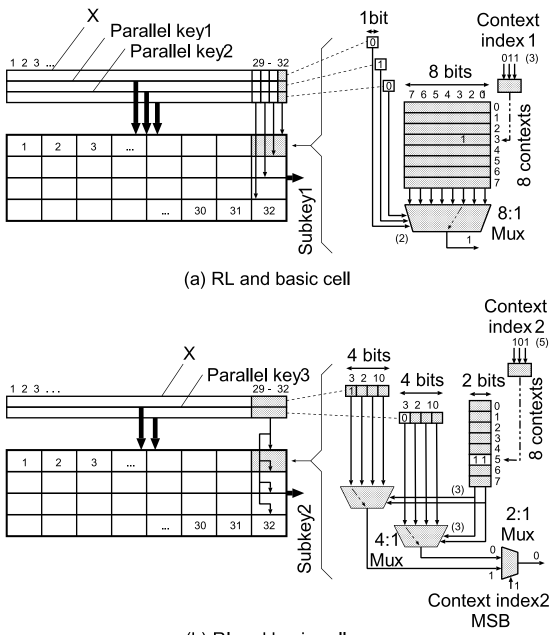


**Fig. 8** 8-context reconfigurable units.

*plexer* from *X* and *Parallel Keys 1* and 2 addresses "bit 2" of the context memory. Thus the context data "1" at this location serves as the final output of the cell.

*RI* consists of an 8 × 4 array of basic cells, each of which contains two *4:1* and one *2:1 Multiplexers*. The input to the left *4:1 Multiplexer* is from *X*, while the input to the right *4:1 Multiplexer* is from *Parallel Key 3*. *Context Index 2* is used to single out one configuration among the 8 contexts. The output of either of the two *4:1 Multiplexers* is selected by the *2:1 Multiplexer* according to the MSB (most significant bit) of *Context Index 2*. In effect, each cell functions as a context-based multiplexer.

In the example shown in the figure, *Context Index 2* selects "context 5" in the context memory, and the data at this context is "3," which causes each of the two *4:1 Multiplexers* to select "bit 3" of the 4-bit inputs. Finally, the MSB "1" of *Context Index 2* selects the chosen bit from *Parallel Key 3* (bit 3 = 0) to generate the final cell output "0".

The context configuration data necessary for *RL* occupy 2,048 bits (32 cells × 8 contexts × 8 bits/cell), while those necessary for *RI* occupy 512 bits (32 cells × 8 contexts × 2 bits/cell). Thus the context configuration data necessary for *RL* and *RI* occupy a total of 2,560 bits (320 bytes). To produce random subkeys, the

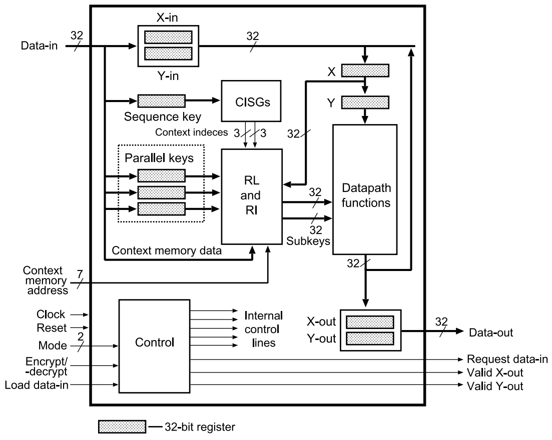


**Fig. 9** Overall architecture of the Chameleon cipher chip.

context data should be as random as possible, with all the contexts in a basic cell functionally distinct from one another. With this restriction, basic cells in *RL* and *RI* can have ${}_{28}P_8$ and $({}_{2^2}P_4)^2$ sets of possible permutations of context values, respectively.

## 2.6 Overall Architecture

**Figure 9** shows the overall architecture of the proposed Chameleon cipher chip. I/O lines consist of distinct 32-bit *Data-in*/*Data-out* lines, 7-bit *Context memory address* lines, and so forth. The 2-bit *Mode* lines select one of the 4 operation modes, namely, *idle*, *context-load*, *key-load*, or *run* mode.

In the *context-load* mode, 32-bit context configuration data are loaded a total of 80 times through the *Data-in* lines of Fig. 9 into the context memories of *RL* and *RI*, in each of which the memory address is specified by the 7-bit *Context memory address* lines. The reprogrammable property of the context data will be described in Section 4.2.

In the *key-load* mode, 32-bit key data are loaded through *Data-in* lines into the registers of *Sequence Key* and *Parallel Keys*, one at a time, to form the whole 128-bit secret key.

In the *run* mode, a 3-stage pipeline process is invoked, which consists of a data input loading stage, a 16-iteration encryption/decryption processing stage, and a data output reading stage. In the first stage, *Data-in* lines are connected alternately to registers *X-in* and *Y-in* to load the left and right halves of the 64-bit input (plain/cipher text) data, respectively. The next stage performs the encryption/decryption on the loaded data, which outputs the results into registers *X-out* and *Y-out*, respectively. In the last stage, *Data-out* lines output alternately the left and right halves of the final 64-bit data obtained in the encryption/decryption stage.

**Table 1** Main characteristics of the Chameleon cipher chip.

| | |
|---|---|
| Technology | 0.6 $\mu$m CMOS 3LM |
| Core size | $3.1 \times 3.1\,\text{mm}^2$ |
| Transistors | 65.6 K |
| Max. clock frequency | 178.6 MHz |
| Max. data throughput | 317.5 Mbps |
| Pins | 82 (signal) + 32 (power) 120-pin package |
| I/O | 32-bit parallel × 2 (plain text/key/context, cipher text) |
| Supply voltage | 5 V |

## 3. VLSI Implementation Results

The Chameleon cryptosystem has been synthesized through Verilog HDL description. The synthesis tools used were Verilog-XL, the Synopsys Design compiler, and Avant! Apollo. However, the reconfigurable units, which consist mainly of SRAM memory cells, were implemented by using the custom layout of Cadence Virtuoso.

**Table 1** shows the implementation results obtained by using the 0.6 $\mu$m CMOS 3LM technology. The critical path, starting from $X$, passing through *RL*, two *XOR*s, decryption *Inverse Substitution*, and terminating at $X$, is 5.6 ns. Thus we have obtained a maximum clock frequency of 178.6 MHz. With each 64-bit block encryption/decryption processed in 36 clock cycles, the corresponding maximum data throughput is 317.5 Mbps. The encryption and decryption use the same subkey generation, but are multiplexed in the datapath functions. Both are set to run at the same speed, determined by the decryption-based critical path.

**Figure 10** shows a micrograph of the obtained chip, where *RL* and *RI* occupy almost 30% of the total chip area. **Figures 11** and **12** show the basic cell architecture of *RL* and a part of the simulation waveforms for the subkey generation in *RL*, respectively, while **Figs. 13** and **14** show the same for *RI*.

As can be seen from the example of Fig. 12, when the context selector signal WL1, inputs IN0 and IN1, and the SENSE signal all rise to a certain level, the generation of Data3 taken from "bit 3" of the context memory is triggered to generate the output signal OUT. As indicated in the figure, a stable output is attained 3 ns after the trigger activation.

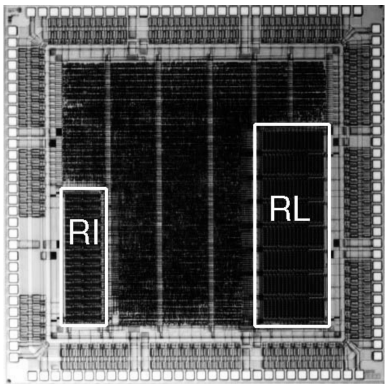


**Fig. 10** Micrograph of the Chameleon cipher chip.

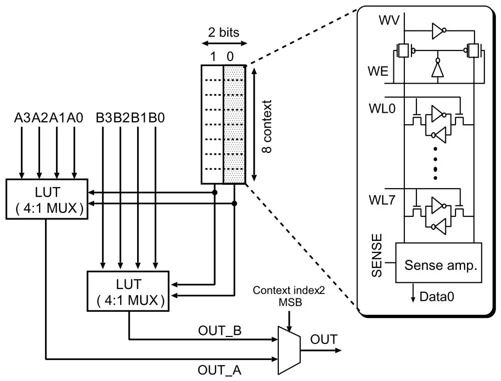


**Fig. 13** Basic cell architecture of *RI*.

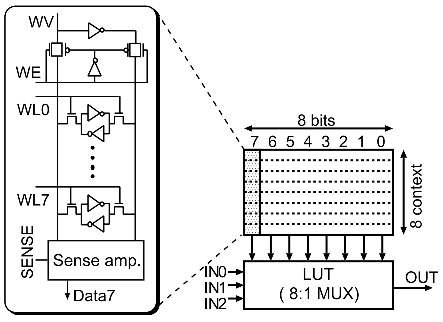


**Fig. 11** Basic cell architecture of *RL*.

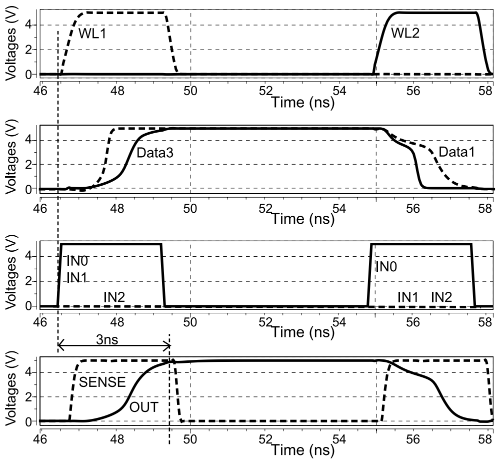


**Fig. 12** Simulation waveforms of the basic cell in *RL* (subkey generation).

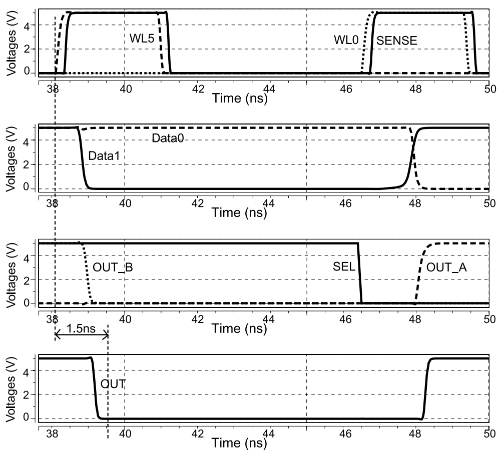


**Fig. 14** Simulation waveforms of the basic cell in *RI* (subkey generation).

On the other hand, as can be seen from the example of Fig. 14, when the context selector signal WL5 and the SENSE signal both rise to a certain level, the generation of Data1 and Data0 taken from "bit 1" and "bit 0", respectively, of the context memory is triggered to generate OUT_B and then the output signal OUT. As shown in the figure, a stable output is attained 1.5 ns after the trigger activation.

## 4. Distinctive Features

### 4.1 Multiple Paths

The security of Chameleon lies mainly in the difficulty of regenerating the data-dependent subkeys at each iteration through the use of multiple contexts in the algorithm. Unlike cryptosystems which employ only one set of circuits to be used repeatedly, Chameleon uses 8 possible circuits (contexts). By means of the rotary dial-based algorithm for the *Context Index* generation, circuits can change at each iteration, thereby increasing the complexity of the task for the attacker/decoder. The possible number of paths for the subkey generation is $8^{16}$; i.e., $2^{48}$ or $2.8 \times 10^{14}$. In contrast, existing algorithms use only 1 path—that is, one fixed circuit—through all iterations.

### 4.2 Reprogrammable Context Data

In addition to being dynamically reconfig-

**Table 2** Comparison of Chameleon and MISTY (hardware)[15].

| Characteristic | Chameleon | MISTY1_43 | MISTY1_3 |
|---|---|---|---|
| Block size | 64 bits | 64 bits | 64 bits |
| Key size | 128 bits | 128 bits | 128 bits |
| Iterations | 16 | 8 | 8 |
| Technology | $0.6\,\mu$m CMOS | $0.8\,\mu$m CMOS | $0.8\,\mu$m CMOS |
| Gates | 16.4 K | 6 K | 25 K |
| Clock frequency | 178.6 MHz | 55 MHz | 16 MHz |
| Encryption speed | 317.5 Mbps | 100 Mbps | 512 Mbps |
| Mbps/Kgates | 19.4 | 16.7 | 20.5 |

urable, the context configuration data of *RL* and *RI* are also reprogrammable, and this property enhances Chameleon's flexibility of use. In contrast to a secret key, which is kept private by a user, the context data may be adopted for common use within an organization and changed according to the users, applications, periods, and so on. Although the security does not lie in keeping the context configuration data secret, this user-modifiable data can be used as an extra 2.5 Kbits of secret-key information. Thus a user can have the option of keeping this information secret within the domain of operation as an additional security measure. The trade-offs for this flexibility are an increase in the area for the subkey generation in proportion to the number of added contexts, as well as the necessity for initial context data loading. To minimize this loading process, the context data can be kept unchanged for a certain period of operation. On the other hand, a frequent change in the context data can prevent cryptanalytic attacks from being attempted against the context data. The frequency of context data change is arbitrarily fixed, depending on the user's security requirement.

### 4.3 Randomness of Cipher Text

We have investigated the degree of randomness of the cipher text by means of a collision test[13] on $2^{12}$ (= 4,096) samples of corresponding 16-bit segments of cipher text blocks. This test evaluates 16-dimensional randomness for $2^{16}$ (= 65,536) possible collisions. The cipher text for this test is generated under the probable condition that the secret key is set to all zeros and the sequence of input plain texts are incremented through a counter. A test on eight sets of 4,096 samples attained an average of 122 collisions. The average of the total number of collisions of a well randomized data set should be between 119 and 134. Noting that the ideal average is $4{,}096^2/(2 \cdot 65{,}536) = 128$, we can see that Chameleon can achieve an almost ideal randomized cipher text.

## 5. Applications and Comparative Evaluation

Chameleon aims mainly at embedded hardware applications in communication and multimedia systems and devices, data networks, and so on, where the chip size and speed are of primary importance.

Major popular encryption algorithms, such as 128-bit block Advanced Encryption Standard (AES) candidates[14], target the whole spectrum of applications for high-end software, smartcards, FPGAs, and LSIs, and hence an increase in chip area is inevitable. For example, a pipelined 128-bit key implementation of Serpent achieves the fastest throughput at 8.03 Gbps with a large transistor count of around 5.7 M. An iterated 128-bit key implementation of Twofish attaining 105 Mbps has a large transistor count of 264 K. Thus AES candidates with a large block size (128 bits) and thereby a larger transistor count, are less suited to embedded applications.

On the other hand, a rough comparison can be performed between Chameleon and MISTY cores[15],[16], commercial ciphers with 64-bit data block size and 128-bit key, which are intended for software and embedded hardware applications. **Table 2** shows the major characteristics of Chameleon and two MISTY cores. It can be observed that although Chameleon is comparable to MISTY1_43 and MISTY1_3 in terms of Mbps/Kgates, the comparison may have some discrepancies. For example a MISTY core uses only $0.8\,\mu$m CMOS technology, while Chameleon uses $0.6\,\mu$m CMOS technology, and, as regards complexity, Chameleon performs 16 iterations, while MISTY performs only 8 iterations. It should also be remarked that the speed advantage of MISTY1_3 can be accounted for by the presence of a full repetition structure of the 8 iterations, which is not included in the current Chameleon prototype chip.

Thus, we can verify that Chameleon, like

MISTY, is suitable for current embedded applications.

Unlike a general-purpose reconfigurable FPGA-like device[1]), the proposed Chameleon cipher chip is a special-purpose implementation, whose size can be reduced. Since Chameleon requires two types of circuits, dynamically reconfigurable and fixed, a hybrid architecture can be employed so as to optimize the silicon area by assigning a relatively large area to the dynamically reconfigurable part of $RL$ and $RI$, but a fairly small area to the remaining fixed part.

In contrast to a special-purpose implementation, a general-purpose implementation can be enhanced in such a way that communication facilities such as channel codec and protocol stack control can be added to the cipher scheme. Consequently, there is a trade-off between chip size and programmability for multiple applications.

## 6. Conclusion

This paper has described the architecture and VLSI implementation of Chameleon, a novel dynamically reconfigurable hardware-based secret-key cryptosystem. Through the use of multi-context units, the number of possible paths for data-dependent subkey generation is increased to $8^{16}$. Furthermore, flexibility is attained by exploiting the reprogrammable feature of reconfigurable units. According to the implementation results, Chameleon realizes a maximum data throughput of 317.5 Mbps with 65.6 K transistors, and hence this cipher can be of practical use for embedded hardware applications in communication and multimedia systems, data networks, and so forth.

Development work is continuing on a sophisticated low-power system-on-a-chip integration of the second-generation Chameleon equipped with communication facilities such as channel codec, protocol stack control, etc., dedicated to mobile applications.

## 7. Acknowledgments

The authors would like to thank Professor Toru Fujiwara for his constructive comments and suggestions, and Makoto Furuie and Morgan Hirosuke Miki for their invaluable assistance in the chip implementation.

The proposed cipher has been fabricated in the fabrication program of VDEC (VLSI Design and Education Center), the University of Tokyo, with the collaboration of Rohm Corporation and Toppan Printing Corporation.

## References

1) Fujii, T., Furuta, K., Motomura, M., Nomura, M., Mizuno, M., Anjo, K., Wakabayashi, K., Hirota, Y., Nakazawa, Y., Ito, H. and Yamashina, M.: A Dynamically Reconfigurable Logic Engine with a Multi-context/Multi-mode Unified Cell Architecture, *ISSCC Digest of Technical Papers*, pp.364–365 (1999).

2) Hauser, J.R. and Wawrzynek, J.: Garp: A MIPS Processor with a Reconfigurable Coprocessor, *Proc. Symposium on FCCM*, pp.12–21 (1997).

3) Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M. and Taylor, R.R.: PipeRench: A Reconfigurable Architecture and Compiler, *IEEE Computer*, Vol.33, No.4, pp.70–77 (2000).

4) Hauck, S.: The Roles of FPGA's in Reprogrammable Systems, *Proc. IEEE*, Vol.86, No.4, pp.615–638 (1998).

5) Higuchi, S. and Kajihara, N.: Evolvable Hardware Chips for Industrial Applications, *Commun. ACM*, Vol.42, No.4, pp.60–69 (1999).

6) DeHon, A.: Reconfigurable Architectures for General-Purpose Computing, AI Technical Report 1586, MIT Artificial Intelligence Laboratory, Cambridge (1996).
ftp://publications.ai.mit.edu/ai-publications/1500-1999/AITR-1586.ps

7) Tau, A., Chen, D., Brown, J. and DeHon, A.: A First Generation DPGA Implementation, *Proc. 3rd Canadian Workshop of Field Programmable Devices*, pp.138–143 (1995).

8) Villasenor, J. and Hutchings, B.: The Flexibility of Configurable Computing, *IEEE Signal Processing Magazine*, Vol.15, No.9, pp.67–84 (1998).

9) National Bureau of Standards, Data Encryption Standard, U.S. Department of Commerce, FIPS pub. 46 (1977).

10) Lai, X. and Massey, J.: A Proposal for a New Block Encryption Standard, *Advances in Cryptology – EUROCRYPT '90 Proceedings*, pp.389–404, Springer-Verlag (1991).

11) Feistel, H.: Cryptography and Computer Privacy, *Scientific American*, Vol.228, No.5, pp.15–23 (1973).

12) Schneier, B.: *Applied Cryptography*, 2nd ed., John Wiley and Sons, NY (1996).

13) Knuth, D.: *Seminumerical Algorithms*, Art of Computer Programming, Vol.2, 2nd ed., pp.68–70, Addison-Wesley, MA (1981).

14) Weeks, B., Bean, M., Rozylowicz, T. and Ficke, C.: Hardware Performance Simulations

of Round 2 Advanced Encryption Standard Algorithms, National Security Agency, pp.37–38 (2000).
http://csrc.nist.gov/encryption/aes/round2/NSA-AESfinalreport.pdf

15) Mitsubishi Electric Corp.: MISTY1 megafunction (1998).
http://www.mitsubishi.com/ghp_japan/misty/misty1megafunc.htm

16) Matsui, M.: New Block Encryption Algorithm MISTY, *Proc. 4th Fast Software Encryption Workshop* (1997).

(Received September 18, 2000)
(Accepted February 1, 2001)

**Zaldy Andales** received the B.S. degree in Applied Physics from the University of the Philippines (UP) at Los Banos in 1991. After graduation, he joined the Institute of Mathematical Sciences and Physics, UP Los Banos. He later obtained the M.S. degree in Electrical Engineering from UP Diliman in 1996. Currently, he is pursuing a doctorate degree in Information Systems Engineering at Osaka University, Japan under the Japanese Government (Monbukagakusho) scholarship. His research interests are in VLSI design and information security.

**Yukio Mitsuyama** received the B.E. and M.E. degrees in Information Systems Engineering from Osaka University, Japan, in 1998 and 2000, respectively. He is currently pursuing a doctorate degree in the same field also at Osaka University. His research interests include VLSI design and implementation of multimedia and security systems.

**Takao Onoye** received B.E. and M.E. degrees in Electronic Engineering, and Dr.Eng. degree in Information Systems Engineering all from Osaka University, Japan, in 1991, 1993, and 1997, respectively. He joined the Department of Information Systems Engineering, Osaka University in 1993 as a research associate, where he was promoted to a lecturer in 1998. Meanwhile, he was with the ICS Department, University of California, Irvine, as a visiting associate researcher in 1997–1998. Presently, he is an Associate Professor in the Department of Communications and Computer Engineering, Kyoto University. He has also served a principal research scientist of Arnis Sound Technologies, Co., Ltd. His research interests include media-centric low-power system architecture and its VLSI implementation.

**Isao Shirakawa** received the B.E., M.E., and Ph.D. degrees in Electronic Engineering from Osaka University, Japan, in 1963, 1965, and 1968, respectively. He joined the Department of Electronic Engineering at Osaka University in 1968 as a Research Assistant. He became an Associate Professor in 1973 and a Professor in 1987 and is now a Professor in the Department of Information Systems Engineering. During 1974–1975, he was with the Electronic Research Laboratory, University of California, Berkeley, as a Visiting Scholar. In 1996 and 1997, he was a Director of the editorial board of IEICE of Japan. He has been engaged in education and research mainly on the basic circuit theory, applied graph theory, VLSI CAD, and VLSI implementation. Prof. Shirakawa was a Vice President of the IEEE CAS Society in 1995 and 1996. He is a Fellow of the IEEE and a Fellow of the IEICE.