An Architecture-Independent Instruction Shuffler to Protect against Side-Channel Attacks

ALI GALIP BAYRAK, NIKOLA VELICKOVIC, and PAOLO IENNE,
Ecole Polytechnique Fédérale de Lausanne (EPFL)
WAYNE BURLESON, University of Massachusetts

Embedded cryptographic systems, such as smart cards, require secure implementations that are robust to a variety of low-level attacks. *Side-Channel Attacks (SCA)* exploit the information such as power consumption, electromagnetic radiation and acoustic leaking through the device to uncover the secret information. Attackers can mount successful attacks with very modest resources in a short time period. Therefore, many methods have been proposed to increase the security against SCA. Randomizing the execution order of the instructions that are independent, i.e., *random shuffling*, is one of the most popular among them. Implementing instruction shuffling in software is either implementation specific or has a significant performance or code size overhead. To overcome these problems, we propose in this work a generic custom hardware unit to implement random instruction shuffling as an extension to existing processors. The unit operates between the CPU and the instruction cache (or memory, if no cache exists), without any modification to these components. Both true and pseudo random number generators are used to dynamically and locally provide the shuffling sequence. The unit is mainly designed for in-order processors, since the embedded devices subject to these kind of attacks use simple in-order processors. More advanced processors (e.g., superscalar, VLIW or EPIC processors) are already more resistant to these attacks because of their built-in ILP and wide word size. Our experiments on two different soft in-order processor cores, i.e., OpenRISC and MicroBlaze, implemented on FPGA show that the proposed unit could increase the security drastically with very modest resource overhead. With around 2% area, 1.5% power and no performance overhead, the shuffler increases the effort to mount a successful power analysis attack on AES software implementation over 360 times.

Categories and Subject Descriptors: C.3 [**Real-time and Embedded Systems**]

General Terms: Design, Security, Performance

Additional Key Words and Phrases: Side-channel attacks, instruction shuffler, random permutation generation

ACM Reference Format:
Bayrak, A. G., Velickovic, N., Ienne, P., and Burleson, W. 2012. An architecture-independent instruction shuffler to protect against side-channel attacks. ACM Trans. Architec. Code Optim. 8, 4, Article 20 (January 2012), 19 pages.
DOI = 10.1145/2086696.2086699 http://doi.acm.org/10.1145/2086696.2086699

1. INTRODUCTION

In the last decade, embedded systems have increasingly become a critical part of daily life. Almost everyone carries their critical personal information in embedded devices such as mobile phones, smart cards and other portable electronic equipment. Ensuring

Authors' addresses: A. G. Bayrak, N. Velickovic, and P. Ienne, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland; W. Burleson, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA. Correspondence; email: aligalip.bayrak@epfl.ch.

DOI 10.1145/2086696.2086699 http://doi.acm.org/10.1145/2086696.2086699

security is crucial for these devices in order not to expose the secret information such as keys and passwords to adversaries. Side-channel attacks (SCA), which exploit various types of leakage emitted from a device, are shown to be an important security threat against the embedded devices. For example, power analysis attacks [Kocher et al. 1999], which use the real time power consumption information of the device during the encryption process of a cryptographic algorithm, are theoretically and empirically proved to be very successful in recovering the secret key used in the encryption. Other SCA, such as electromagnetic [Gandolfi et al. 2001], acoustic [Shamir and Tromer 2004] and timing [Kocher 1996] have also been studied and shown to be effective.

With the invention of SCA, researchers started to develop countermeasures to increase the security against these attacks, especially for the power and EM based ones, since these attacks are easy to mount, efficient and effective. One of the most popular countermeasures against these attacks is to randomly change the order of the instructions that could be performed independently, in each different run of the implementation. This method is known as *shuffling*. Different approaches in hardware and software have been proposed for shuffling in the literature. Software approaches mostly focus on a specific application of the method on a chosen algorithm and lack generality. They also have the problem that in order to implement the feature, we either lose performance or increase the code size, often drastically. An efficient way of designing a generic shuffler which does not rely on the underlying software implementation is to implement a hardware shuffler which randomly selects among the instructions that could be run at a given time step. May et al. [2001] proposed such a design where they used the idea of superscalar computers for randomization of the instructions rather than for parallelism. Although their approach is generic, the work lacks important implementation details on hardware since they did not implement the idea on a real system. When implemented, their system will occupy a significant area. This paper exploits the fact that there has been significant research on compilers for parallel architectures where instructions that can be executed in parallel (which are the instructions that will be shuffled in our case) could be identified at compile time. This effectively allows for a simple and efficient shuffler convenient to use in embedded applications.

In this work, we propose an alternative hardware shuffler design and verified it on two different soft processor cores, i.e., Open-RISC and MicroBlaze. Note that, the proposed unit is mainly designed for in-order processors since they are the main targets of SCA [Mangard et al. 2007]. Embedded devices subject to SCA, such as smart cards (used in many applications such as pay TV, banking, health care, public transit, etc.) [Mangard et al. 2007] and car keys [Paar et al. 2009], use simple in-order processors [Gemalto 2008; ARM 2011; MicroChip 2011; STMicroelectronics 2004] because of the constraints such as cost, size and energy. More advanced processors such as superscalar, VLIW or EPIC, are structurally more resistant to SCA because parallelism, dynamic scheduling and wide word size are the factors which drastically increase the effort to mount a SCA [Mangard et al. 2007]. However, we still discuss the effect of our shuffler unit on a superscalar processor in Section 4.5 and show that it might further increase the security of such a processor.

In our experiments, we used SASEBO (Side-channel Attack Standard Evaluation BOard) G-II [SASEBO 2009] board to evaluate the security of the implemented system. We used the open source Open-RISC implementation with cache and closed source MicroBlaze implementation without cache. The overall architecture is given in Figure 1. We have a custom hardware shuffling unit which intercepts the signals between the processor and instruction cache (or memory, if cache does not exist) and provides the processor one of the instructions that could run at the current clock cycle. The dependency analysis, which determines the instructions that could be shuffled, is

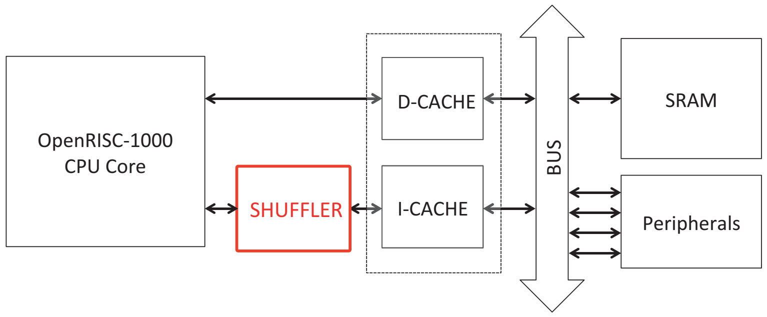


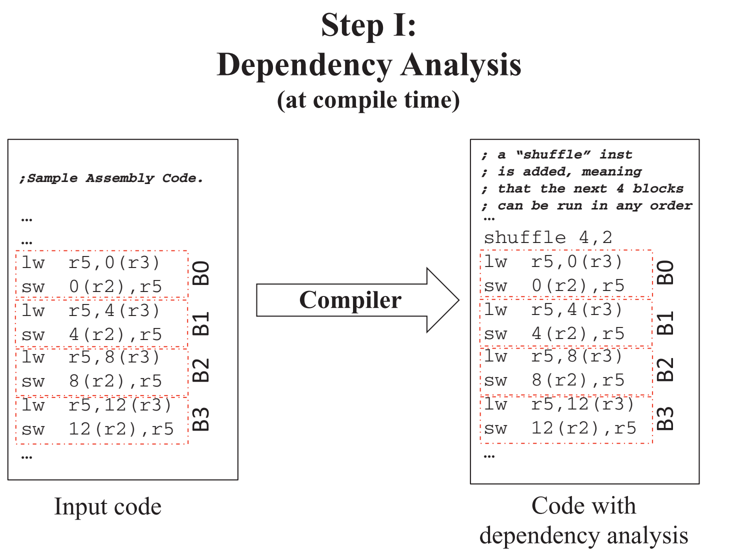
Fig. 1. The overall architecture is shown. The shuffler is placed between the CPU core and instruction cache (or memory, if cache does not exist) and it provides the CPU core random instructions selected from pool of instructions that could be run in any order.

done at compile time as opposed to the approach of May et al. This eliminates the need of complex hardware design that determines the dependencies and thus minimizes the impact on area and energy. An important fact that is considered at this point is that, for cryptographic systems, most of the dependency analysis can be done statically at compile time because the cryptographic algorithms are structurally deterministic and execution flow is generally input independent. They are designed intentionally to be input independent because of efficiency reasons and also input dependent executions are subject to other serious attacks like timing attacks.

Figure 2 shows a basic flow of how the system works. The idea of the proposed system is similar to that of VLIW, except that the dependency analysis is used for ordering in the former while it is used for parallelism in the latter. In other words, the instructions that run in parallel in a VLIW processor would run sequentially, but in random order, in our system. We support instruction and block level dependencies. Being able to shuffle the blocks of instructions brings flexibility and eliminates a huge effort on the compiler level by eliminating the need of tricks such as resource allocation and register renaming. Also, the user can manually specify the dependencies easily, if he/she knows the semantics of the given code. For example, for the AES algorithm, in each round, there are 16 SubBytes operations that could be run in any order. The user could simply insert the custom instruction which represents the independent blocks just at the beginning of those operations, after identifying how many instructions a SubBytes operation consists of. Similar tricks could also be done for the other operations (i.e., AddRoundKey, MixColumns, ShiftRows). Having identified the dependencies, the processor uses this information not for parallelism as in VLIW, but for ordering at run time. The shuffler provides the determined instructions to the processor in the run-time generated random order.

The overall method is shown to be very efficient, i.e., with very low resource overhead we can gain a significant security increase. The experiments on the SASEBO-GII board showed that with only about 2% area, 1.5% power and almost no performance overhead, the security of a cryptographic implementation could be increased 360 times. We used the number of necessary traces to be able to mount a successful power analysis attack, as a security metric.

The hardware could also support other countermeasure, i.e., random insertion of dummy operations. In this countermeasure, we insert dummy operations at random moments during the run time. When combined with shuffling, the effect of both countermeasures are superimposed, thus generating even more resistant devices [Mangard et al. 2007]. This can be handled by simply adding dummy blocks in addition to the

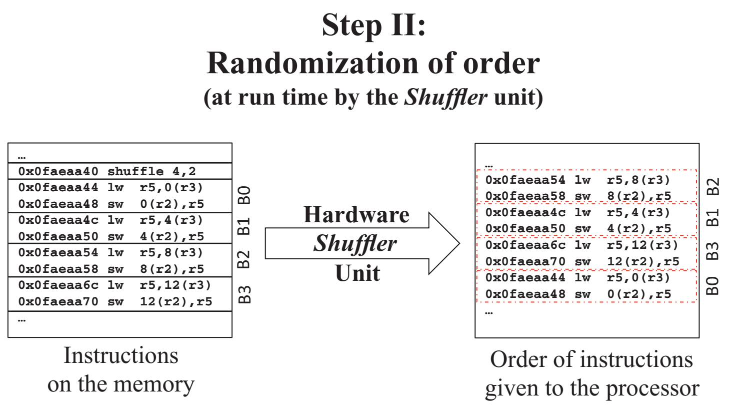


```
;Sample Assembly Code.

…
…
lw   r5,0(r3)
sw   0(r2),r5
lw   r5,4(r3)
sw   4(r2),r5
lw   r5,8(r3)
sw   8(r2),r5
lw   r5,12(r3)
sw   12(r2),r5
…
```

```
; a "shuffle" inst
; is added, meaning
; that the next 4 blocks
; can be run in any order
…
shuffle 4,2
lw   r5,0(r3)
sw   0(r2),r5
lw   r5,4(r3)
sw   4(r2),r5
lw   r5,8(r3)
sw   8(r2),r5
lw   r5,12(r3)
sw   12(r2),r5
…
```

(a) The dependency analysis is done at compile time. The custom instruction, "`shuffle n,m`", tells the shuffler unit that the next n block could be run in any order and each block consists of m instructions.


Step II:
Randomization of order
(at run time by the *Shuffler* unit)
Hardware *Shuffler* Unit
B0
B1
B2
B3
Instructions on the memory
Order of instructions given to the processor

```
…
0x0faeaa40 shuffle 4,2
0x0faeaa44 lw   r5,0(r3)
0x0faeaa48 sw   0(r2),r5
0x0faeaa4c lw   r5,4(r3)
0x0faeaa50 sw   4(r2),r5
0x0faeaa54 lw   r5,8(r3)
0x0faeaa58 sw   8(r2),r5
0x0faeaa6c lw   r5,12(r3)
0x0faeaa70 sw   12(r2),r5
…
```

```
…
0x0faeaa54 lw   r5,8(r3)
0x0faeaa58 sw   8(r2),r5
0x0faeaa4c lw   r5,4(r3)
0x0faeaa50 sw   4(r2),r5
0x0faeaa6c lw   r5,12(r3)
0x0faeaa70 sw   12(r2),r5
0x0faeaa44 lw   r5,0(r3)
0x0faeaa48 sw   0(r2),r5
…
```

(b) When the shuffler unit recognizes a "shuffle" instruction, it generates a random permutation of the blocks (e.g., (2,1,3,0) in this case) and provides the instructions to the processor in the generated order one by one, each time it is asked for next instruction.

Fig. 2. The *shuffler unit* randomizes the execution order of the independent instruction blocks, which are determined at compile time.

existing independent blocks at compile time and let the shuffler randomize them at run time.

2. SIDE-CHANNEL ATTACKS AND RELATED WORK

To understand the concept of random shuffling, we first give some brief background on side-channel attacks and in particular, power analysis attacks. When the cryptographic implementation is running on a physical device, the attacker observes the leakage emitted through it and uses this information to recover the secret information (e.g., key). This leakage could be power consumption, electromagnetic radiation, acoustic, etc. The first step of a power analysis attack, which uses the former, is to collect the real time power consumption values from the device while it is encrypting a set of given plaintexts. This can be done with very modest resources, i.e., an oscilloscope with probes and a resistor connected to the power pin of the device. After collecting the data, the power traces are analyzed off-line using some statistical methods. The basic assumption is that the power consumption while executing an instruction is correlated with the data it is processing. Given the fact that we know the cryptographic algorithm running on the device, since it is generally public information; and the plaintexts that have to be encrypted, since we provide them to the device as input, we make assumptions on an intermediate operation of the plaintexts and the secret key. For example, the first operation of the AES algorithm is an exclusive-or of the key and the plaintext. Thus, we know that at some time during the encryption process, the first byte (word) of the key will be exclusively-or'ed with the first byte (word) of the plaintext. Then, if we are trying to recover the secret key, for each possible key guess, we correlate the measured power consumption with the hypothetical power consumption values (based on a simple power consumption model as a function of chosen intermediate operation) and find the key guess which gives the highest correlation. Experiments have shown that only a few traces could be enough to recover the secret key [Mangard et al. 2007].

To prevent these attacks, many methods have been proposed in the past. One of the most common ideas is to randomize the execution, so that the power traces collected by the attacker would be misaligned. Random insertion of dummy instructions and random shuffling are such methods. Other countermeasures include randomization of power consumption [Shamir 2000; Benini et al. 2003], randomly changing clock frequency [Zafar et al. 2010], boolean and arithmetic masking [Coron and Goubin 2000; Akkar and Giraud 2001; Blömer et al. 2004; Oswald et al. 2005], protected logic styles [Tiri et al. 2002; Tiri and Verbauwhede 2004; Toprak and Leblebici 2005], etc. All of these countermeasures increase the effort to mount a successful attack but do not guarantee a perfect protection against side-channel attacks.

Shuffling, which is the focus of this work, can be implemented both in hardware and software. Most of the previous works focus on specific algorithms [Tillich et al. 2007; Kamal and Youssef 2009; Madlener et al. 2009]. May et al. [2001] proposed a generic shuffler which is an extension to the processor core, similar to ours. They used the idea of superscalar computers for randomization of the instructions and determined the dependencies at run time using tables. Although their approach is generic, their idea has not been implemented on a real system. Considering the structure of cryptographic algorithms, which are generally deterministic, determining the dependencies at run time is wasteful and brings a huge area and energy overhead. Instead, as is done in VLIW processors, these dependencies could be determined at compile time easily, which eliminates the complicated dependency analysis circuitry. This results in a simple, efficient and effective unit which could be used on existing embedded systems easily. In addition to this, our processor supports shuffling blocks of instructions (instead of single instructions) which gives the user flexibility and eliminates complicated compiler tricks such as resource allocation and register renaming.

In this work, we propose an efficient hardware implementation of random shuffling method. Our hardware could also support the random insertion of dummy instructions.

3. RANDOM SHUFFLING

In this section, we explain the random shuffling method in detail and discuss the proposed hardware shuffler unit.

3.1. Random Shuffling Method

The basic idea of random shuffling is to execute a set of instructions that do not have dependencies in a random order. For example, if we have two `load` instructions and then an `add` instruction that adds the two loaded values, the order of `load` instructions is not important and could be randomly determined at run time. We also can have independent instruction blocks. For example, in Figure 2(a), we show code for copying the content of one array to the other and the order in which the copy operation is performed is not important; so, the blocks (i.e., B[0..3]) could be run in any order, as in the example in Figure 2(b). Since a block could consist of one or more instructions, we use the term "*block*" in the rest of the paper to describe both the independent instructions or instruction blocks.

Random shuffling could be implemented either in software or hardware or both. Software implementations of shuffling are generally algorithm-specific and lack generality. If the designers want to implement a generic software shuffling method, they should either keep track of the executed blocks by complicated conditional branching structures in the code, or embed all possible random orders to the code. The former adds a reasonable amount of performance (and therefore, energy) overhead, while the latter increases the code size drastically (in the order of $n!$, where n is the number of blocks to be shuffled). Doing the shuffling in hardware is better in these perspectives. However, we pay some area overhead. Fortunately, our experiments showed that our custom hardware shuffler unit occupies a small amount of the overall area of the processor (2% of OpenRISC) and has a small power (1.5%) overhead with almost no performance overhead.

The random shuffling method basically consists of two main steps, which is shown in Figure 2. Firstly, we analyze the dependencies and determine which blocks could be shuffled, at compile time. Then, at run time, we randomly shuffle the order of blocks that are determined in the previous step. In each shuffling operation, a random permutation is randomly generated and the instructions are executed in this order.

The dependency analysis is handled at compile time and blocks that could be executed in random order are specified with "`shuffle`" custom instruction (or any reserved instruction for the same purpose) as can be seen in Figure 2(a). This analysis could be done in different ways. For example, we can use a VLIW compiler, since the idea of the proposed system is similar to the that of VLIW, except that the dependency is used for ordering in the former while it is used for parallelism in the latter. The instructions that can be run in parallel in a VLIW processor will run sequentially but in random order in our system. As an alternative, the intermediate representations generated by off-she-shelf compilers, e.g., data and control flow information, could be exploited to modify the compilers to support our custom instruction. Another alternative is to manually insert the custom instructions, which represent the independent blocks, to the generated assembly code. This is, as opposed to many manual processes, an easy task if the user knows the semantics of the underlying implementation. For example, the 16 SubBytes operations within a round of AES could be run in any order. Similar considerations could also be done easily for all the other operations, i.e., AddRoundKey, ShiftRows, MixColumns. Since we are able to shuffle the blocks, the user only needs to know the number of instructions for one iteration of the high level operation (e.g., SubBytes), which is obvious from the generated assembly code.

Random shuffling of the determined blocks at run time is handled by our hardware shuffler unit that is situated between the instruction cache (or memory) and the processor core. The unit only intercepts the signals between these units and modifies the data; so we do not need to modify neither these units, nor the protocols or signals. The shuffling of the blocks is done in three main steps.

In the first step, the unit recognizes the custom `shuffle` instruction. This is done when the processor asks for the next instruction from the instruction cache (or memory). The shuffler simply checks the opcode of the instructions before sending them to the processor and determines whether it is a `shuffle` instruction or not. If not, it sends the instruction to the processor as is. If it is a `shuffle` instruction, the unit checks the number of blocks to be shuffled, N_b, and the number of instructions in each block, N_i. For example, $N_b = 4$ and $N_i = 2$ in Figure 2(b). If the blocks are of varying size, `nop`s could be inserted at compile time to make them of equal size. For simplicity, N_b is assumed to be a power of 2 and has an upper limit determined by the size of the unit. In our experiments, we supported up to 32 blocks but the proposed methods are all generic and this number could be increased easily. For n blocks, the area is on the order of $O(n * log_2 n)$. Number of instructions in a block, i.e., N_i, could be any positive integer.

In the second step, a random permutation of numbers between $[0, N_b - 1]$ is generated. This permutation is created by the *random permutation generator* described in Section 3.2 and determines the execution order the blocks. This step could be handled at the same clock cycle with the first step. In this clock cycle, the unit could either send the processor a `nop` instruction or a wait signal instead of the custom instruction. More `nop`s (or wait signals) could be sent if these steps could not fit in a clock cycle in a very fast processor (which is not the case in embedded systems or FPGAs). In fact, this extra one clock cycle could be eliminated as explained in Section 4.2.

As a last step, until all of the instructions that have to be shuffled are sent to the processor, the unit provides the instructions one by one in the generated random order. The details of how this is done is provided in Section 3.3. Note that the blocks are not interleaved, i.e., all the instructions in a block are sent consecutively before the instructions of the next block are sent. See Figure 2(b) for an example.

After all these steps are completed, the random shuffler again starts to forward all the instructions without change until it detects a `shuffle` instruction.

3.2. Random Permutation Generator

An important part of the shuffler unit is the *random permutation generator*, which permutes the numbers between $[0, N_b - 1]$, where N_b is the number of blocks to be shuffled. For example, it produces (2, 1, 3, 0) in Figure 2(b). There are many classic works in the literature which propose solutions to this problem, especially for software. The “Knuth shuffle” algorithm is one of the most popular and efficient software solutions and has a $O(n)$ time complexity, where n is the number of elements to be shuffled. However, these algorithms generally do not consider parallel execution. Using parallelism, shuffling can be done in $O(log_2 n)$. The idea of multistage interconnection networks (MINs) could be used in order to implement efficient permutation generators using parallelism. A detailed analysis of MINs is given in the thesis of Rani [2011]. Lee et al. [2001] proposed efficient permutation instructions, based on MINs, for programmable processors. In order to permute n bits, we need to use $O(log_2 n)$ of these instructions. Later, Shi et al. [2003] proposed alternative approaches for application specific instruction processors to achieve 64-bit permutations in one or two cycles and Lee et al. [2005] proposed MOMR (Multiple Operands Multiple Results) implementations to achieve n-bit permutations in one or two cycles. In contrast, in this work we implemented an architecture-independent permutation generator fully in hardware which can run within a single clock cycle. We used the idea of a *permutation network* by Waksman [1968], which describes an efficient switching network to permute a set of signals. This idea could be realized in hardware very efficiently by a combinational circuit of depth $O(log_2 n)$ and size $O(n \cdot log_2 n)$, which has a similar cost as the barrel shifter.

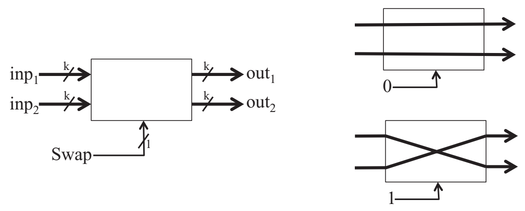


Fig. 3. The *swapper* swaps the two k bit inputs if the select bit is 1.

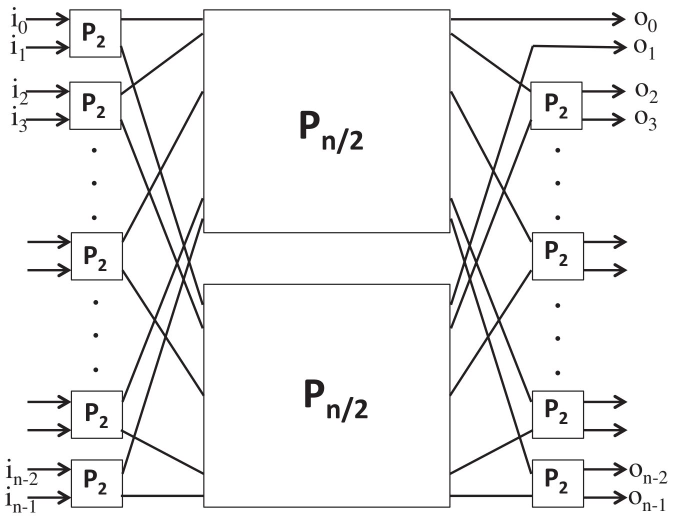


Fig. 4. P_n, which permutes the given n inputs is shown in the figure. P_2 is the swapper shown in Figure 3. The circuit is defined recursively with two $P_{n/2}$ and $n-1$ times P_2. The P_n circuit, in total, consists of $n \cdot log_2 n - n + 1$ swappers, i.e., P_2, when we solve the recurrence relation. The critical path has $2 \cdot log_n - 1$ swappers.

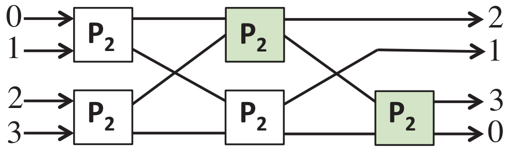


Fig. 5. Example permutation generator circuit for $n = 4$, i.e., P_4. The swappers, i.e., P_2, that are shown with green have select signal with value 1.

The permutation generator circuit consists of *swappers* shown in Figure 3. The select signal decides to swap two k bit inputs or not, where $k = log_2 n$. The permutation circuit is defined recursively as follows. The basic swapper shown in the figure is represented as P_2. Then, P_n, which permutes the given n inputs (the numbers between $[0, N_b - 1]$ in our case) is designed as shown in Figure 4. It consists of two $P_{n/2}$ circuits and $n - 1$ swappers, i.e., P_2. When we solve the recurrence relation, we can find that the P_n circuit consists of $n \cdot log_2 n - n + 1$ swappers in total. The depth of the circuit, which is the number of swappers on the critical path, is $2 \cdot log_n - 1$.

To show how the implementation works, we give an example for $n = 4$ in Figure 5. This circuit can generate any permutation of numbers between $[0, 3]$ by assigning the necessary select bits as proved in the original work of Waksman. In the

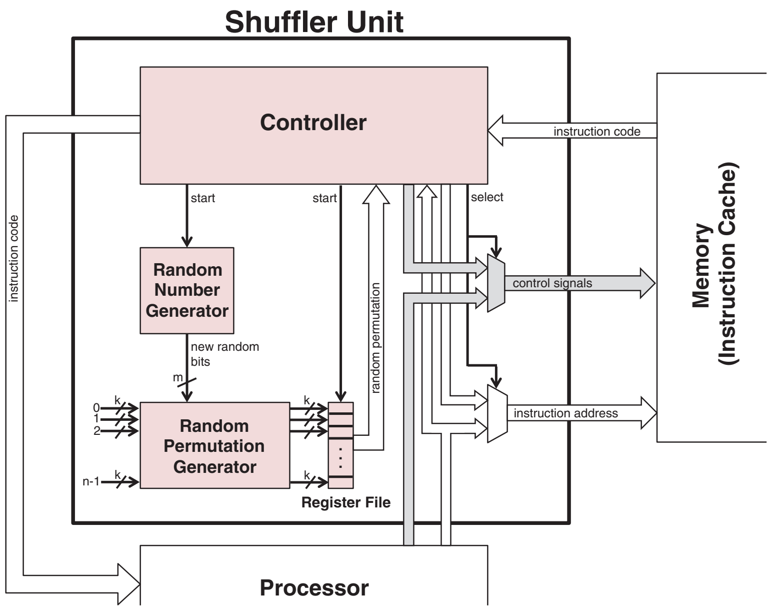


Fig. 6. The overall *shuffler* unit is shown. There are four main components. *Controller* is the component which starts the execution of the shuffler and controls the other components. *Random number generator* produces $m = n \cdot log_2 n - n + 1$ random bits, where n is the number of blocks to be shuffled and then *random permutation generator* permutes the numbers between $[0, n - 1]$ using these random bits, as described in Section 3.2. At the same clock cycle, the permutation is stored in the *register file* and the controller gives the instruction blocks in the generated order until it sends all of them.

figure, we see how it can generate the permutation $(2, 1, 3, 0)$. The swappers that are shown with green have select signal with value 1 and the others have the value 0. The select bits are determined at run time randomly. Since we need one random bit per swapper, we need $n \cdot log_2 n - n + 1$ random bits to generate a random permutation. These random bits could be generated by a TRNG or a PRNG or could be taken from outside through some input device. We discuss the details of random number generation in Section 3.4.

We have shown in the experimental section that the critical path for the shuffler with $n = 32$ is less than a clock cycle of the processors we used, which means that it will not add any delay to the execution of the program. Because of the structure of the random permutation generation circuit, it also can be divided into smaller units (e.g., according to the depth of the swappers) in case of much faster devices, to allow multi-cycle execution. Any other random permutation generation circuit that satisfies the same condition could be used instead of this. We have selected this algorithm, since it is simple and efficient. The efficiency and correctness proofs of this permutation generator are given in Waksman [1968].

3.3. Execution of Shuffler

The overall shuffler unit is shown in Figure 6. Shuffler intercepts the signals and the data between the processor and the memory (or the instruction cache, if exists). Controller checks the opcode of the instruction sent from memory to the processor and starts the shuffling if it is a `shuffle` custom instruction. If processor does not have a custom instruction support, any of the existing instructions could be used for the

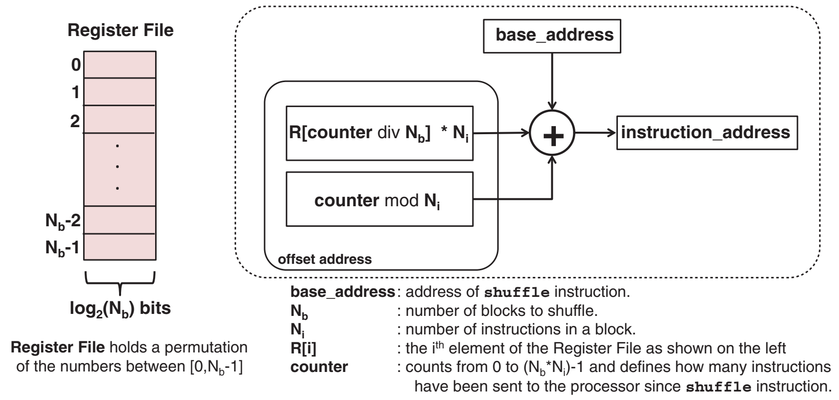

Register File
0
1
2
N_b-2
N_b-1
$\log_2(N_b)$ bits
Register File holds a permutation of the numbers between [0,N_b-1]
base_address
R[counter div N_b] * N_i
counter mod N_i
offset address
instruction_address
base_address: address of shuffle instruction.
N_b : number of blocks to shuffle.
N_i : number of instructions in a block.
R[i] : the i^{th} element of the Register File as shown on the left
counter : counts from 0 to (N_b*N_i)-1 and defines how many instructions have been sent to the processor since shuffle instruction.

Fig. 7. In this figure, we show how the addresses of the instructions in the shuffled order are generated. As discussed in Section 3.2, first, a random permutation of the numbers between $[0, N_b - 1]$ is generated and stored to the register file. After that, at each new instruction request, the offset is calculated using the shuffled numbers in the register file and the *counter*. The *counter* is initialized to 0 at the beginning of the shuffling and incremented by one after each instruction request. Note that *base_address* is the address of the `shuffle` instruction and all the instructions that will be shuffled are contiguous in the memory.

same purpose. When the *random number generator* gets the start signal, it generates $n{\cdot}log_2n{-}n{+}1$ random bits to be used by the *random permutation generator* as described in Section 3.2. *Random permutation generator* produces a permutation of the numbers between $[0, n - 1]$ and this permutation defines the execution order of the blocks to be shuffled. This operation is handled in a single clock cycle and the result is stored in the *register file*. After that, during the next $N_i \cdot N_b$ instruction requests from the processor, at each step, the *controller* requests the next instruction by sending the shuffled address to the memory and sending the fetched instruction from the memory to the processor. Note that, the shuffler unit does not firstly fetch all the instructions to be shuffled and then shuffle them (i.e., opcodes); instead, it shuffles the offsets (which are used in calculating the instruction addresses) and then sends to the memory the addresses in the shuffled order while delivering the received opcodes from the memory to the processor without changing them. How the addresses are generated in the shuffled order is shown in Figure 7. After all the instructions are sent, the controller sends the data and the signals between the processor and the memory without any change until it detects another custom `shuffle` instruction.

3.4. Random Number Generation

Random number generation is an important part of the shuffler unit. It provides the random bits that will be used by the random permutation generator unit. The random numbers could be received from an external device or generated in the unit. There are two different kinds of random number generators (RNG), True Random Number Generator (TRNG) and Pseudo Random Number Generator (PRNG). The former has an unpredictable behavior, even for the designers, and is generally based on some physical inputs such as noise. The latter produces a sequence of random numbers using a deterministic approach from a given number, called a seed. Some of today's embedded systems come with TRNG or PRNG in them. In this case, we can use this built-in RNG for the random number generation. Otherwise, we can build one ourselves. There are many works which propose PRNG or TRNG implementation for both ASIC [Cui et al. 2002; Tokunaga et al. 2008; Zhun and Hongyi 2001] and FPGA [Danger et al. 2007; Klein et al. 2008; Kwok and Lam 2007; Schellekens et al. 2006]. We used FPGA implementations in this work. We have implemented both PRNG and TRNG and

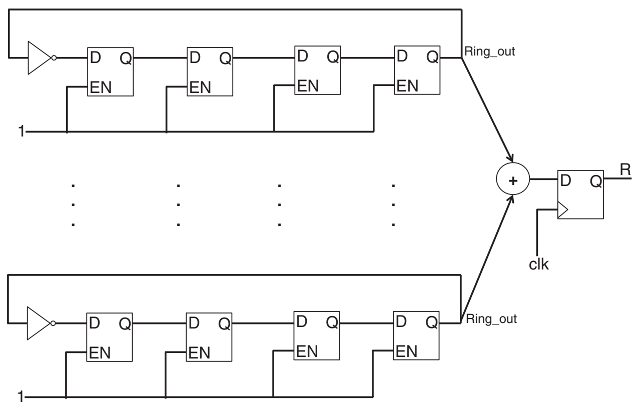


Fig. 8. The TRNG architecture is shown. Ring oscillators, which are composed of an inverter and delay elements, are exclusively-or'ed and the result is sampled with system clock.

discussed the area and security tradeoff in the experimental section. Our implementations are entirely on chip in order not to expose them to the attackers.

For the PRNG, a simple linear feedback shift register (LFSR) could be used. LFSRs have a period of $2^n - 1$ where n is the number of registers in it. Xilinx Virtex devices have SRL (Shift Register LUT) macro, implementing efficient shift registers varying from one to sixteen bits and an LFSRs could be implemented using these SRLs [George and Alfke 2007]. The selection of the primitive polynomial determines the maximum length pseudo random sequence and appropriate primitive polynomials are described in [Alfke 1996]. Although the PRNG could produce uniform random numbers, the output is deterministic and so could be exploited by the attackers. Even for the tamper resistant devices where the attacker does not have access to the PRNG, the characteristics of it, such as period, could be determined by applying input patterns and analyzing the output sequences.

For the TRNG, the proposed methods in the literature either use jitter [Klein et al. 2008; Kwok and Lam 2007; Schellekens et al. 2006] or metastability [Danger et al. 2007] as a noise source. We implemented the method proposed by Klein et al. [2008], which is based on sampling jitter. Basically a high frequency clock generated by ring oscillators is sampled with the system clock. The TRNG circuit is shown in Figure 8 to generate one single random bit. It is composed of multiple ring oscillators each of which is made from an inverter and delay elements (open latches) as noise source. To obtain better random numbers, it is proposed to exclusive-or the outputs of multiple ring oscillators. The output is sampled with system clock and the single output bit, R, is produced. In order to generate multiple bits, a straightforward way is to produce them independently to reduce the correlation between the bits. This approach requires a considerable area, but has a high throughput, is less predictable and more resistant against attacks. To avoid the high area requirement, as suggested by Klein et al. [2008], we can redesign the TRNG so that it produces just one bit at a time, and stores it to a register which has a width of necessary number of random bits. The bits can be shifted at each cycle to allow the next bit stored into the LSB. This approach would need only one unit that is shown in Figure 8, and thus will occupy less area, but has a low throughput. Another approach would be to combine these two techniques, and produce r bits at a time by u units, where $r \cdot u = b$ and b represents the number of necessary bits for the random permutation generator. Remember that $b = n \cdot log_2 n - n + 1$ where n is the number of blocks to be shuffled.

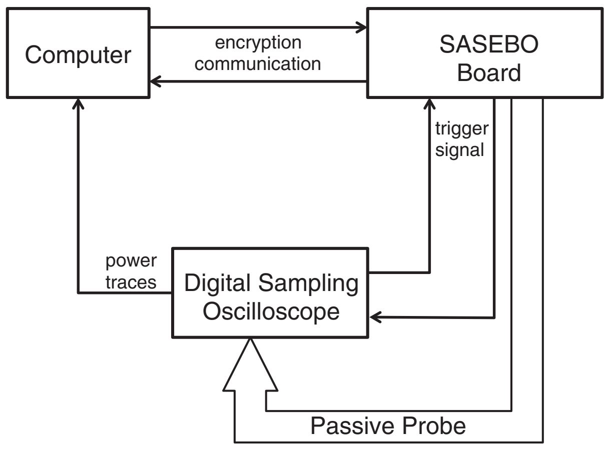


Fig. 9. The power measurement setup is shown. Power consumed by the FPGA board is measured using a passive probe connected to a digital sampling oscilloscope. The board and oscilloscope communicates to inform each other about the start and end of measurement. Computer and board communicates for loading the code and checking the results. Power traces are analyzed and attack is mounted at the computer.

4. EXPERIMENTAL RESULTS

4.1. Experimental Setup

The proposed solution has been implemented and tested on two different systems. The first system has a Xilinx FPGA with Virtex II (XC2V8000) chip. The second system is SASEBO (Side-channel Attack Standard Evaluation BOard) G-II board [SASEBO 2009], which is becoming popular in side-channel evaluation experiments. It has two Xilinx FPGAs, Virtex-5 (XC5VLX30) and Spartan-3A (XC3S400A-4FTG256). The board is designed to make the side-channel measurements easy, e.g., we can connect a passive probe to one of the connectors on the board and measure the consumed power.

The setup for power measurements is shown in the Figure 9. It consists of four main components. The *SASEBO board* runs the encryption algorithm on the soft processor core which has the proposed hardware extension. When it starts encryption, the *oscilloscope* is triggered to start storing the power consumption trace. The *passive probe* connected to the oscilloscope is used to measure the power consumption. The oscilloscope samples the data at $4GHz$ but samples it down (we used $10 * f$ where f is the frequency of the clock of the processor) using peak detect mode, which only saves the min and max peaks. This method is used to save space and reduce the saving time. It has been shown in Mangard et al. [2007] that the peaks reflect the overall behavior of the consumption successfully. The number of samples to be stored is determined according to the number of clock cycles of the encryption. When the saving of data is finished, oscilloscope sends a signal to the board to let it run the encryption for the next input. These triggering tricks are used to align the power traces. If the attacker does not have such an access, the traces could be collected without trigger and then aligned off-line before the attack, using the alignment techniques discussed in Mangard et al. [2007]. The *computer* communicates with the board to send the program to the board and verify the results of the encryption on the board. It is also used for collecting the traces from the oscilloscope and mounting the attack.

4.2. Performance and Power Results

Our proposed extension has a very low performance overhead and even that can be eliminated by a small modification. If there is no shuffling, the system simply works

Table I. Power consumption of the shuffler with $N_b = 16$ and $N_b = 32$, where N_b is the maximum number of blocks that could be shuffled

	Base System	Shuffler with $N_b = 16$	Shuffler with $N_b = 32$
Total Power (mW)	258.1	261.8	262.5
Percentage	100%	101.4%	101.7%

Table II. Virtex-2 Area Results. The shuffler is implemented as an extension to OpenRISC soft processor core. The numbers represent the area of proposed shuffler, TRNG, PRNG and processor with the units, respectively. Area is defined in terms of number of occupied sequential and combinational elements. N_b represents the maximum number of blocks that could be shuffled.

		Shuffler Unit	TRNG	PRNG	Total System
$N_b = 4$	Regs	75 (0.9%)	152	5	8446
	LUTs	237 (1.3%)	991	1	18346
$N_b = 8$	Regs	91 (1.1%)	164	17	8474
	LUTs	280 (1.5%)	991	1	18423
$N_b = 16$	Regs	131 (1.5%)	196	49	8548
	LUTs	469 (2.5%)	991	1	18602
$N_b = 32$	Regs	226 (2.6%)	276	129	8722
	LUTs	1112 (5.8%)	991	1	19267

as usual. If there is a shuffling, the one clock cycle for `shuffle` instruction might be considered as overhead. But s/c ratio, where s represents the number of `shuffle` instructions and c represents the number of clock cycles for the whole encryption is generally very low. For example, the unprotected AES software implementation we used in our experiments takes 5701 clock cycles, while we need only 40 shuffle instructions even if want to protect all high level operations in all rounds, which will add approximately 0.7% performance overhead. In fact, this one clock cycle because of `shuffle` instruction could also be avoided. In the architecture, we told that the random permutation is generated when the unit detects a `shuffle` instruction and the one clock cycle delay is caused by this random permutation generation operation. We can also prepare these numbers at the beginning of the run and at the last clock cycle of each shuffle instruction, so that if the unit detects a `shuffle` instruction, it can use previously generated random permutation and simply send the next instruction to the processor.

The second important criteria is the clock rate at which the unit can operate. Our experiments have shown that the critical path for the shuffler unit for 32 blocks is less than that of soft processor cores we used. This means that our shuffler can generate the random permutation of 32 numbers, i.e., determine in which order the blocks will run, in less than one clock cycle of the maximum clock rate that both the OpenRISC and MicroBlaze can operate at. We have tried up to 32 blocks since it is enough parallelism for most cryptographic algorithms (e.g., 16 is enough for AES).

For the power consumption of the shuffler, our shuffler unit for 32 blocks consumes only 1.7% more power compared to the base system (without shuffling unit). This number is 1.4% for the shuffler for 16 blocks. We can see the absolute numbers in Table I.

4.3. Area Results

We give the area usage of the units (shuffler, PRNG and TRNG) in the Tables II and III. The numbers of PRNG and TRNG are given only for reference and are not a part of the shuffler core. If there is an existing RNG in the system, we can easily use that one. The first system is implemented on Virtex2 FPGA with an OpenRISC soft processor core, while second is implemented on Virtex5 with a MicroBlaze soft core on it. OpenRISC is an open source project and code is accessible, while the MicroBlaze is closed source

Table III. Virtex-5 Area Results. The shuffler is implemented as an extension to MicroBlaze soft processor core. The numbers represent the area of proposed shuffler, TRNG, PRNG and processor with the units, respectively. Area is defined in terms of number of occupied slices, sequential and combinational elements. N_b represents the maximum number of blocks that could be shuffled.

		Shuffler Unit	TRNG	PRNG	Total System
$N_b = 4$	Slices	66 (3.5%)	477	2	1869
	Regs	71 (4.0%)	152	5	1757
	LUTs	102 (3.9%)	746	1	2571
$N_b = 8$	Slices	93 (4.8%)	495	5	1951
	Regs	91 (5.0%)	164	17	1789
	LUTs	153 (5.8%)	746	1	2618
$N_b = 16$	Slices	136 (6.9%)	486	13	1973
	Regs	131 (7.0%)	192	49	1862
	LUTs	278 (10.1%)	748	1	2745
$N_b = 32$	Slices	335 (15.4%)	491	33	2176
	Regs	227 (11.1%)	259	129	2040
	LUTs	751 (23.3%)	752	1	3222

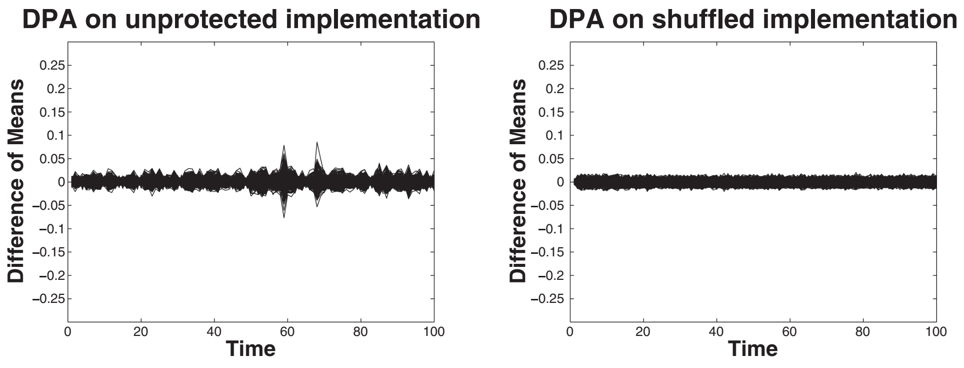
and we used only the provided signals. We provide the numbers of occupied slices, sequential and combinational elements, respectively. We do not have slice numbers for the first implementation because the tools used did not provide the information. As can be seen from the results, if we want to support shuffling blocks of size up to 16, which is enough for most cryptographic applications, such as AES, we only have 1.5% and 2.5% area overhead in terms of occupied sequential and combinational elements, respectively, compared to the OpenRISC processor core. This number goes high in MicroBlaze processor, because we used the minimal possible configuration of the core where it supports only the necessary operations.

The experiments are repeated on two different processor cores to show these following features. First, we have shown that our method is independent of the processor used and could be ported easily to another core only if the signals between the processor core and memory (or cache) are known. We also showed that our shuffler could work both in the presence of cache (in OpenRISC) or not (in MicroBlaze). Last but not least, by using a closed source implementation, i.e., MicroBlaze, we showed that the system does not modify the existing components, but only intercepts the signals.

4.4. Security Results

In this section we provide security results. We collected power traces during the encryption process of the AES-128 implementation on MicroBlaze processor with and without the shuffler. The experimental setup mentioned in Section 4.1 is used for the measurements. We ran each system (w/ and w/o shuffler) 100000 times giving the same set of inputs. We have fixed N_b, the maximum number of blocks to be shuffled, to 16, which is generally the chosen value for AES algorithm since it has 16 bytes *state* and the operations (e.g., SubBytes) for each byte could be handled in random orders. After collecting the traces, we mounted two different attacks, differential power analysis attacks (DPA) [Kocher et al. 1999] and correlation-based differential power analysis attack (CPA) [Brier et al. 2004]. We attacked the SubBytes operation, which is a non-linear operation, because, the non-linear operations are the weakest points where the attackers generally target [Mangard et al. 2007]. As a power model, which is used for calculation of hypothetical power values, we used Hamming weight.

The results of the mentioned attacks are three-dimensional matrices, *time* vs. *value* vs. *key guess*. *Time* dimension represents the sampling points since we collect power traces during a time interval, not only for single points at time. The *value* is “difference-of-means” for DPA and “correlation coefficient” for CPA. If the *value* is high, the *key*



(a) Unprotected implementation. (b) Shuffled implementation.

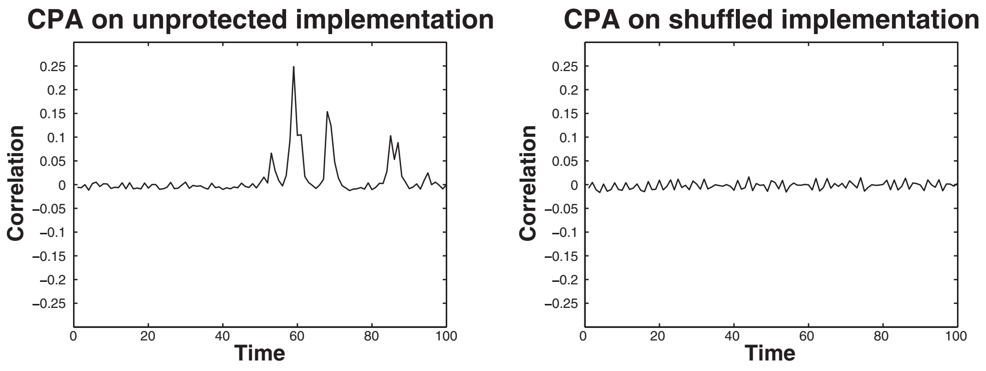
Fig. 10. The results of the DPA attack is shown. The x-axis represent the time and y-axis represent the difference of means, which shows the recoverability. All key guesses are shown in the figure and the key guess which shows a peak in the first figure is the correct key and the time where we can see the peak is the clock cycle where the attacked operation is performed (or the result of it is processed in later cycles). In the second figure, there is no visible peak, which means that the shuffled implementation is not attackable using difference-of-means based DPA.

guess that gives this *value* has more chance of being the correct key and the *time* where we see the peak has more chance of being the moment where the attacked operation (i.e., SubBytes in our case) has been performed. What we basically do is to get *time* vs *value* graphs for each possible *key guess* and if there is a *key guess* who has a distinguishably high *value* at any *time*, we are successful in recovering the key. As a *key guess*, we do not focus on whole key at once (e.g., 128 bits in AES-128), but we try to recover it byte-by-byte at a time, since the operations are performed in bytes (or words) on the processors.

In Figure 10, we can see the results of the difference-of-means based DPA; the correct key is distinguishable in the original version without shuffler, while the shuffled implementation does not give any significant peaks. This means that our shuffler is successful in protecting the implementation, i.e., the attack is unsuccessful on the shuffled implementation.

CPA is generally more successful in recovering the key. As mentioned in [Mangard et al. 2007], theoretically, the correlation coefficient, which increases with the recoverability, is reduced by N_b times for random shuffling of N_b instructions. They also show that the number of necessary traces to be able to mount a successful attack increases by N_b^2 times in case of N_b times decrease in correlation. Our experiments, as can be seen in Figure 11, shows that the correlation coefficient is decreased by more than 19 times. This is mainly because of the shuffling (16 times) and the rest is because of the noise added by the shuffler unit. We can use the formula given in [Mangard et al. 2007] and conclude that the number of necessary traces to mount a successful attack increases by 366 times for the shuffled implementation.

We have used the TRNG for the security evaluation because of its unpredictable behavior, which is important for security. Although PRNG could give us random numbers, the deterministic structure of it could be exploited by the attacker. For example, if the period, P, of it is not high enough, the attacker can simply get one of the P consecutive encryptions and ignore the rest, which will end up with having the same random number. This does not apply in our case, since the period of the PRNG that can be used with our 16-block shuffler has a period of $2^{49} - 1 \approx 5 \cdot 10^{14}$, but other attacks, which exploit the deterministic nature of the PRNG, especially if the attacker knows the structure of the PRNG, are also possible. Some of the examples are *backtracking*,



(a) Unprotected implementation. (b) Shuffled implementation.

Fig. 11. The results of the CPA attack is shown. The x-axis represent the time and y-axis represent the correlation coefficient, which shows the recoverability. Only the correlation for the correct key guess is shown in the figure because we wanted to show the values of the correlation coefficient clearly; otherwise, the figures are similar to ones in Figure 10, except the peaks of CPA on the unprotected implementation is higher. The correlation for the correct key decreased from 0.2486 to 0.013 from unprotected implementation to shuffled implementation, which means 366 times increase in the number of necessary traces to mount a successful attack.

permanent compromise, *iterative guessing* attacks elaborated in [Kelsey et al. 1998]. The consequences are that a good quality PRNG which is resistant to these kinds of attacks is more difficult to design than a TRNG, which is why, when possible, a TRNG is a better solution for the source of randomness.

4.5. Shuffler Unit with Out-of-Order Processors

As we discussed in Section 1, the shuffler unit is mainly designed for in-order processors because of two main reasons. Firstly, the devices that are the targets of the mentioned attacks have simple in-order processors because of constraints such as cost, size and energy. Secondly, more advanced processors have some features which make them already more resistant to those attacks, such as parallelism, dynamic scheduling and wide word size. However, in this section, we discuss whether the proposed unit could further increase the security of such processors.

The shuffler unit changes the order of the instructions not for efficiency, but for security reasons. When the shuffler unit is used with an out-of-order processor (e.g., superscalar), the scheduling of the processor might undo the effects of the shuffling for performance reasons. This mainly depends on two parameters; the size of the blocks to be shuffled and the size of the instruction window of the scheduler. If the former is big enough compared to the latter, then the scheduler might change the order of the instructions in a block, but it won't be able to undo the whole effect of the shuffler. In this case, the security of the processor will be further increased. On the other hand, if the latter is big enough compared to the former, then the scheduler might undo most of the effects of the shuffler, resulting in no significant security improvement. The dependencies between the instructions also effect the overall behavior. In order to see what happens in a real life situation, we simulated our AES implementation with shuffling on an out-of-order superscalar processor using *SimpleScalar* tool version 3.0e [SimpleScalar Tools 2011]. We used the *default* configuration file of the tool to define the processor. Some important configuration settings are listed in Table IV.

To simulate the effect of the shuffler, we took the AES implementation used in our experiments and generated the shuffled versions of this base implementation off-line, by placing the instruction blocks in shuffled orders and thus generating one unique

Table IV. Some important *default* configuration settings of the *out-of-order* superscalar processor in *SimpleScalar*

Inst fetch queue size	4
Branch predictor type	bimod
Instruction decode B/W (insts/cycle)	4
Instruction issue B/W (insts/cycle)	4
Register update unit (RUU) size	16
load/store queue (LSQ) size	8
Total number of integer ALU's available	4

code for each permutation. Since the attackers' main target is the SubBytes operation in AES as discussed in Section 4.4, we simulated the shuffled versions of this operation. Note that, the SubBytes is basically a non-linear operation which updates the state matrix by performing "`state[i][j] = S[state[i][j]]`", where $0 \leq i, j < 4$ and `S` is a non-linear look-up table of size 256. Since generating all the possible permutations is computationally infeasible ($16! \approx 2.1 \cdot 10^{13}$), we used the possible permutations of only the first 4 blocks ($P(16, 4) \approx 4.4 \cdot 10^4$) assuming that the behavior should be similar for the remaining blocks. Note that, a block consists of the instructions that perform one step of the SubBytes operation; thus, there are 16 blocks each of which perform the look-up operation for a given (i, j) pair. Using *SimpleScalar*, we simulated each of these codes on the given processor and examined the clock cycles spent in the functional unit for each instruction. Then, we analyzed whether the instruction blocks are processed in the given order or the processor has further changed the order of the instructions and hence the blocks are intermixed. We examined this behavior by observing different instructions in a block since the blocks consist of the same instructions (e.g., the *store* instructions which store the result of the operation to the `state` matrix or the *load* instructions which load the `state[i][j]`). After analyzing the results, we observed that a chosen instruction is always processed in the order given in the shuffled code and no change is done by the processor. For example, if the "`state[1][2] = S[state[1][2]]`" operation comes before the "`state[1][0] = S[state[1][0]]`" operation in the shuffled code, then the *store* operation which stores the result of the former is processed before the *store* of the latter. This means that the order in which the "blocks" are executed is mainly determined by the random permutation generated by the shuffler rather than the built-in scheduler of the processor. We are talking about inter-block ordering and note that the instructions in a block could be rescheduled by the scheduler. Since the mentioned attacks make use of the alignment of the "instructions" in time dimension (at which clock cycle an instruction is performed) and the shuffler is shown to be able to determine this order, we can conclude that the shuffling will further increase the security of this out-of-order processor as mentioned in Section 4.4. Note that, this is just an example of the applicability of the shuffler to an out-of-order processor and as discussed before in this section, this conclusion can not be guaranteed under all situations. The size of the blocks and the dependencies in the code caused this behavior in this example.

5. CONCLUSIONS

In this work, we proposed a generic hardware shuffler unit as an extension to the processor, which randomly shuffles the execution order of the independent instructions, to protect against side-channel attacks. To show that the proposed unit could be used with different architectures, we ported the unit to two different processors, i.e., MicroBlaze and OpenRISC. We used the closed-source system, i.e., MicroBlaze, to show that the unit does not need any modification on the existing system, but is just an addition which only intercepts some signals. In the experimental section, we showed that the security gain is drastic with a very modest resource overhead. The unit can especially be used for lightweight embedded systems because of its cost-efficient and security-effective structure.

REFERENCES

Akkar, M.-L. and Giraud, C. 2001. An implementation of DES and AES, secure against some attacks. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*. 309–318.

Alfke, P. 1996. Efficient shift registers, LFSR counters, and long pseudo random sequence generators. http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf.

ARM. Downloaded on October 21st, 2011. SecurCore processors. http://www.arm.com/products/processors/securcore/index.php.

Benini, L., Macii, A., Macii, E., Omerbegovic, E., Pro, F., and Poncino, M. 2003. Energy-aware design techniques for differential power analysis protection. In *Proceedings of the Design Automation Conference (DAC)*. 36–41.

Blömer, J., Guajardo, J., and Krummel, V. 2004. Provably secure masking of AES. In *Proceedings of Selected Areas in Cryptography (SAC)*. 69–83.

Brier, E., Clavier, C., and Olivier, F. 2004. Correlation power analysis with a leakage model. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*. 16–29.

Coron, J.-S. and Goubin, L. 2000. On boolean and arithmetic masking against differential power analysis. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*. Vol. 1965, 231–237.

Cui, W., Chen, H., and Han, Y. 2002. VLSI implementation of universal random number generator. In *Proceedings of Asia-Pacific Conference on Circuits and Systems (APCCAS)*. 465–470.

Danger, J.-L., Guilley, S., and Hoogvorst, P. 2007. Fast true random generator in FPGAs. In *Proceedings of Circuits and Systems (NEWCAS)*. 506–509.

Gandolfi, K., Mourtel, C., and Olivier, F. 2001. Electromagnetic analysis: Concrete results. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*. Vol. 2162, 251–261.

Gemalto. 2008. Gemalto.NET user guide. http://www.swisstech.it/store/pdf/Gemalto.NET_User_Guide.pdf.

George, M. and Alfke, P. 2007. Linear feedback shift registers in virtex devices. http://www.xilinx.com/support/documentation/application_notes/xapp210.pdf.

Kamal, A. A. and Youssef, A. M. 2009. An area-optimized implementation for AES with hybrid countermeasures against power analysis. In *Proceedings of the International Symposium on Signals, Circuits and Systems (ISSCS)*. 1–4.

Kelsey, J., Schneier, B., Wagner, D., and Hall, C. 1998. Cryptanalytic attacks on pseudorandom number generators. In *Proceedings of Fast Software Encryption (FSE)*. 168–188.

Klein, C., Cret, O., and Suciu, A. 2008. Design and implementation of a high quality TRNG in FPGA. In *Proceedings of Intelligent Computer Communication and Processing (ICCP)*. 311–314.

Kocher, P. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Proceedings of Advances in Cryptology (CRYPTO)*. Vol. 1109, 104–113.

Kocher, P., Jaffe, J., and Jun, B. 1999. Differential power analysis. In *Proceedings of Advances in Cryptology (CRYPTO)*. Vol. 1666, 398–412.

Kwok, S. H. M. and Lam, E. Y. 2007. FPGA-based high-speed true random number generator for cryptographic applications. In *Proceedings of TENCON*. 1–4.

Lee, R., Shi, Z., and Yang, X. 2001. Efficient permutation instructions for fast software cryptography. *IEEE Micro 21,* 6, 56–69.

Lee, R. B., Yang, X., and Shi, Z. J. 2005. Single-cycle bit permutations with MOMR execution. *J. Comput. Sci. Technol. 20,* 5, 577–585.

Madlener, F., Stoettinger, M., and Huss, S. A. 2009. Novel hardening techniques against differential power analysis for multiplication in $GF(2^n)$. In *Proceedings of the International Conference on Field-Programmable Technology –ICFPT 2009*. 328–334.

Mangard, S., Oswald, E., and Popp, T. 2007. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer.

May, D., Muller, H. L., and Smart, N. P. 2001. Non-deterministic processors. In *Proceedings of Information Security and Privacy (ACISP)*. Vol. 2119, 115–129.

MicroChip. Downloaded on October 21st, 2011. MicroChip Keeloq. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2075&param=en001022#P156_7747.

OSWALD, E., MANGARD, S., PRAMSTALLER, N., AND RIJMEN, V. 2005. A side-channel analysis resistant description of the AES S-box. In *Proceedings of Fast Software Encryption (FSE)*. 413–423.

PAAR, C., EISENBARTH, T., KASPER, M., KASPER, T., AND MORADI, A. 2009. KeeLoq and side-channel analysis - Evolution of an attack. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 65–69.

RANI, R. 2011. Design and performance evaluation of multistage interconnection networks. Ph.D. thesis, Punjabi University, India.

SASEBO 2009. Side-channel Attack Standard Evaluation Board SASEBO-GII Specification. http://staff.aist.go.jp/akashi.satoh/SASEBO/pdf/SASEBO-GII_Spec_Ver1.01_English.pdf.

SCHELLEKENS, D., PRENEEL, B., AND VERBAUWHEDE, I. 2006. FPGA vendor agnostic true random number generator. In *Proceedings of Field Programmable Logic and Applications (FPL)*. 1–6.

SHAMIR, A. 2000. Protecting smart cards from passive power analysis with detached power supplies. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*. 71–77.

SHAMIR, A. AND TROMER, E. 2004. Acoustic cryptanalysis: On nosy people and noisy machines. http://www.cs.tau.ac.il/~tromer/acoustic/.

SHI, Z., YANG, X., AND LEE, R. B. 2003. Arbitrary bit permutations in one or two cycles. In *Proceedings of Application-Specific Systems, Architectures and Processors (ASAP)*. 237–247.

SIMPLESCALAR TOOLS. Downloaded on October 21st, 2011. Simplescalar simulator software. http://www.simplescalar.com/.

STMICROELECTRONICS. 2004. Smartcard 32-bit MCU datasheet. http://pdf1.alldatasheet.com/datasheet-pdf/view/107490/STMICROELECTRONICS/ST22N256.html.

TILLICH, S., HERBST, C., AND MANGARD, S. 2007. Protecting AES software implementations on 32-bit processors against power analysis. In *Proceedings of Applied Cryptography and Network Security (ACNS)*. 141–157.

TIRI, K., AKMAL, M., AND VERBAUWHEDE, I. 2002. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the European Solid-State Circuits Conference (ESSCIRC)*. 403–406.

TIRI, K. AND VERBAUWHEDE, I. 2004. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proceedings of Design, Automation and Test in Europe (DATE)*. 246–251.

TOKUNAGA, C., BLAAUW, D., AND MUDGE, T. 2008. True random number generator with a metastability-based quality control. *IEEE J. Solid-State Circuits 43,* 1, 78–85.

TOPRAK, Z. AND LEBLEBICI, Y. 2005. Low-power current mode logic for improved DPA-resistance in embedded systems. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*. 1059–1062.

WAKSMAN, A. 1968. A permutation network. *J. ACM 15,* 1, 159–163.

ZAFAR, Y., PARK, J., AND HAR, D. 2010. Random clocking induced DPA attack immunity in FPGAs. In *Proceedings of the International Conference on Industrial Technology (ICIT)*. 1068–1070.

ZHUN, H. AND HONGYI, C. 2001. A truly random number generator based on thermal noise. In *Proceedings of the International Conference on ASIC (ICASIC)*. 862–864.

Received July 2011; revised October 2011 and December 2011; accepted December 2011