ESM: A Plug-in Power Side Channel Shuffling Protection for Scalar Processors

Yao Zhang and Ming Tang

***Abstract*—Hiding is a protection method against power side channel attacks on processor chips. Compared to masking, hiding has less area and time overhead. Existing hiding schemes usually have to identify the instructions which need protection to change their execution moments. This makes these efforts problematic in that 1) these hiding schemes need extension of the ISA and modification of the compiler, 2) large changes to the original microarchitecture, and 3) protection targets limited to cryptographic algorithms and the corresponding arithmetic instructions. In this work, we design a plug-in shuffling protection called External Shuffling Module (ESM), between the front-end and the back-end of the processor based on the instructions which can change the order of execution in program segments. By constructing an instruction dependency table, the order in which instructions are executed each time is changed within the processor, thus avoiding modifications to the compiler. ESM receives all signals from the front-end and forwards them to the back-end through internal processing, without the need for any other components in the microarchitecture. This allows ESM to be started and stopped at any time and applied to different microarchitectures. ESM applies protection to all instructions flowing through it. On the SAKURA-GIII board, we implement ESM on a RISC-V processor. When the processor is able to counter an MLP based power attack of 1 million training traces, ESM requires an additional 1.23% area overhead as well as 12.12% time overhead.**

***Index Terms*—power side channel, protection, scalar processors, shuffling**

I. Introduction

In today's digital era, computers are essential in our daily lives, but this also poses a risk of security threats to computer systems. One of the most dangerous system security threats is the side channel attack, which was first proposed by Kocher [1]. Among the types of side channel attacks, the power side channel attack is particularly concerning as it can extract sensitive information from a device's power consumption (e.g. processor) without physical access. The power consumption of a processor varies depending on the executed instructions [2], making it a target for power side channel attacks on embedded processors, personal computers, and large servers. Adversaries can use the power consumption of a system to infer sensitive information such as encryption keys, passwords, or any other sensitive information that may be present on the device. To counteract this threat, many processor protection schemes have been proposed to reduce the information leakage that an adversary can obtain in power consumption or limit the leakage that is not relevant to sensitive information to make it impossible to launch a successful attack. The main strategies for general-purpose processor power protection are masking and hiding. The masking scheme, originally proposed by Chari et al. [3] and applied to simple devices like cryptographic chips, splits secret information into $share$s and transforms operations into $share$ operations. Masking ensures that not all $share$s of each clock are involved in the operation, preventing adversaries from restoring the secret information through incomplete $share$s. Hannes et al. [4] extends the masking to open source processors by redesigning the processor's execution unit, to comply with masking operation rules. While the masking schemes can achieve provable security in the framework of [5], it imposes significant hardware resource overhead. Additional shared factors require doubling storage and computing resources.

Another way to prevent power side channel attacks is hiding, where sensitive data is periodically moved within the processor. By changing the order of operations or inserting delays before and after sensitive operations, the adversary's ability to extract information from the processor's physical structure is reduced. This makes it more difficult for an adversary to launch a successful side-channel attack against the processor. Compared to masking, hiding protection is simple to implement, has a low resource overhead, and has a low loss of performance (masking requires an additional clock for each additional $share$ to perform the $share$ operation). Although hiding does not provide provable security, the security strength can be assessed by the degradation of signal-to-noise ratio in real power consumption and the ability to successfully reveal the secret information. Many hiding protection strategies have been proposed [6]–[10].

However, most of these studies have focused on the cryptographic chips or software code level shuffling. Fewer studies have been conducted directly on processor hardware. Bayrak et al. [11] propose a generic custom hardware unit to implement random instruction shuffling as an extension to existing processors. The unit operates between the processor and the instruction cache. The author claims that their scheme has around 2% additional area, 1.5% additional power and no performance overhead. However, the reason they achieve such a small overhead is by incorporating the shuffling process in the compiler. Aerabi et al. [12] introduce a new processor architecture that can randomly interleave the execution of two programs. If the processor runs only one algorithm, they develop a preprocessor tool that turns the target algorithm into two (or more) separate queues that can run in alternation. The added circuits cause increased area by about 2% in terms of

Y. Zhang and M. Tang are with School of Cyber Science and Engineering, Wuhan University, Wuhan, China.

M. Tang is with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, Wuhan University, Wuhan, China.

registers and 7% in LUTs and decreased speed by about 4%. The PARADISE proposed by Chen et al. [13] dynamically reorders instructions in an Out-of-Order(O-o-O) processor to provide obfuscated execution with minimal overhead (3.7% on performance, 1.1% on power and 0.7% on area). PARADISE adds a slack unit between the issue unit and the execution unit, which records the time when the instruction enters the issue unit. Based on the time difference between instructions, a suitable execution time delay is inserted to shuffle the power consumption. The limitations of this solution are that it can only be applied to an O-o-O processor and requires modifications to the issue unit in the original design. Pham et al. [14] propose a method called XDIVINSA to implement shuffling in a coprocessor. XDIVINSA extends the instruction set architecture by adding an identification bit to the instructions. Instructions that require protection are tagged and sent to the coprocessor for delayed execution instead of being executed by the original execution unit.

Motivation. We observe that current hardware-based implementations of hiding protection for processors predominantly rely on insertion delays. Conversely, protection of the shuffling type often necessitate compiler support to disorder program segments at the software level. Pure delay guards come at a significant time cost. Moreover, requiring compiler intervention reduces the universality of the protection since software layers need to be designed in parallel. Our inspiration stems from modern O-o-O processors, wherein compiled instruction segments have the potential to swap the execution order of instructions. It means instruction shuffling can be hardware-based implementation. Using the segment's own changeable-order instructions to disrupt the entire instruction sequence introduces no additional time overhead. Shuffling protection goes beyond a simple emulation of the O-o-O processor. The O-o-O processor's objective is to prioritize efficiency and increase pipeline throughput rate. However, when the program context is fixed, the sequence of output instructions remains the same each time. In contrast, shuffling aims to prioritize chaos, ensuring the probability of executing instructions at each moment is as uniform as possible.

Contributions. 1) In this work, we introduce a hardware-based solution named the External Shuffling Module (ESM). ESM offers a versatile and efficient approach to mitigate power side-channel attacks on scalar processors. Here are the key features of ESM:

i. Plug-in Capability: ESM is designed as a plug-in scheme. It takes the signals valid for each instruction cycle of the processor as input and does not require additional signals for activation. ESM stores and forwards the decoded instructions at the appropriate cycle, making it compatible with various processor pipelines without necessitating alterations to the original components. It's a pure hardware implementation, eliminating the need for user-defined ISA or compiler support for protection. This feature is described in Sec.IV-A.

ii. General and Entire Protection: ESM extends protection to all types of programs, not limited to cryptographic algorithms. It safeguards the entire program segment, ensuring that all instructions are protected rather than focusing on specific ones. This feature is described in Sec.IV.

iii. Configurability: ESM is a flexible, configurable solution. It's not a fixed-size design, and its security capability is proportional to the area overhead. Configuration parameters are accessible via the ESM interface, allowing users to adjust the trade-off between security capability and area overhead based on the device's operating environment's signal-to-noise ratio. In Sec.V-C, we demonstrate how to configure ESM.

2) We design and implement specific ESM protection on the SAKURA-GIII board. The protected processor is the CVA6 processor core based on the RISC-V ISA. On this board, we evaluated the resource overhead and security capabilities of the ESM. Compared to existing works, our solution has the following advantages:

i. Metrics. We propose two metrics in Sec.III-B to measure the obfuscation level and security capability of the instruction sequence. The security metric considers the worst-case security scenario of the instruction sequence, and the obfuscation metric considers the overall obfuscation level of the instruction sequence. These two metrics can guide the user to adjust the parameters in ESM in different applications and noise environments, so that the final protection has the appropriate resource overhead and achieves a certain level of security. The metrics are generic and are not limited to evaluating the capabilities of ESM.

ii. Less Overhead: ESM imposes less overhead in terms of area and time compared to existing hiding protections. With the ability to resist a deep learning based SCA attack using one million traces, ESM only incurs about 1.23% of the area overhead of the original design, with a time overhead of approximately 12.12%. Considering ESM provides entire protection, the time overhead further reduces when narrowing down the protection scope to common arithmetic instructions of cryptographic algorithms. The specific result is shown in Sec.V-C.

The remainder of the paper is organized as follows. Section 2 presents relevant background in the scheme design and evaluation. In Section 3, we discuss the main idea and propose the new metrics. The detailed implementation of our scheme is proposed in Section 4. We present the experimental results, including the security and the overhead in Section 5. The total paper is concluded in Section 6.

II. Background

A. The structure of processors

The architecture of modern processors is divided into two parts: the front-end and the back-end. The front-end retrieves instructions from memory, decodes them into micro-instructions, and transfers them to the back-end for execution. The back-end is responsible for executing the micro-instructions received from the front-end. The micro-instructions are renamed in registers and provided with necessary resources, after which they enter the issue unit. The issue unit sends the micro-instructions to corresponding ports for execution. Once executed, the micro-instructions enter the reorder buffer and retire in order. Processor microarchitectures vary in their front-end and back-end implementations. Some processors may include modules to accelerate decoding in

the front-end, such as the Decoded Stream Buffer(DSB). O-o-O processors, in contrast to in-order processors, will also have Reorder Buffers(ROB) in the back-end. However, all processors can be logically divided into the structure of the front and back ends.

B. Power side channel attack model

The operation of a device can produce physical side channel information, such as power consumption and electromagnetic radiation. Power consumption is closely related to the device's operation and data. We can infer the corresponding device state by analyzing the power consumption during device operation. In a power side channel attack, we call the attacked device as the target device, and the data we want to obtain is referred to as the secret information. The data obtained by the target device after processing the input x and the secret information s is called the intermediate values iv. When the target device processes a known set of q inputs $X = \{x_1, x_2, \cdots, x_q\}$, we obtain a set of physical power consumption observations $L = \{l_1, l_2, \cdots, l_q\}$, where each power consumption l_i corresponds to a particular operation of x_i and s. A side channel adversary uses a distinguisher $\mathcal{D}$ [15] to analyze these observations. The distinguisher takes IV(or X), L and the space of possible values for s as the input, and outputs the guess secret information s^* based on the probability that s^* is the correct s. Traditional side channel distinguishers include Pearson correlation coefficient [16], Bayesian classification [17], mutual information [15], and more. In recent years, many deep learning-based distinguishers have also been proposed [18].

C. Side channel evaluation

1) TVLA: Test vector leakage assessment (TVLA) [19] is a leakage assessment method based on the Welch's t-test proposed by Cryptography Research Inc. The Welch's t-test is used to determine if two sets of data are significantly different from each other. The test statistic follows the Student's t-distribution, and is designed to give a probability that indicates whether the means of the two data sets are different. The aim is to determine whether the samples in the two data sets are indistinguishable or not. The t-test is generally considered indicating that the two data sets are indistinguishable when the value of the t-test falls within the interval of [-5, 5], according to the hypothesis testing principle of the t-distribution. In the TVLA method, the first set of data is the power consumption observation L_{fixed} generated by processing the fixed inputs $X_f = \{x, x, \cdots, x\}$ on the device, while the second set of data is the power consumption observation L_{random} generated by processing the random inputs $X_r = \{x_1, x_2, \cdots, x_q\}$. By analyzing the t-score of the data sets, it is possible to determine whether the target device is generating the power consumption associated with the input. When a sufficient amount of data is available, the t-value can be used to make this determination.

2) The rank of the correct guess: For the distinguisher's highest scoring guess, there exists a special case where the output guess is incorrect, but the probability of the correct guess is higher than the majority of other guesses. In this case, although the attack is not directly successful, it is still valid because the number of exhaustive enumerations can be greatly reduced by sorting all guesses according to their probability. Selçuk et al. [20] propose using the *rank* metric to evaluate the effectiveness of different attacks and their bounds. The secret information guesses are ranked according to the distinguisher score. Effective attacks should make the *rank* of the correct guess higher and eventually stabilize in the first place. The effectiveness of different attacks can be compared based on the *rank* value of the convergence of correct guesses and the convergence time.

III. Reordering in Sequential Processors

It's easy to randomize the execution order of instructions, but we need the reordered segment to be able to perform their original function. At the same time, the processor's resources are very limited, so if the 101st instruction wants to be executed first, obviously the processor needs to have a place to store the previous 100 instructions. It also takes extra time to make sure that advancing the instruction has no effect on the final result. In this section we show how to ensure the integrity of the reordered segment and how to minimize the additional time overhead.

A. Main Idea

Based on the *Theorem 1* proposed in [21], the original meaning of a program segment can be preserved as long as the original dependence of each instruction is guaranteed while reordering.

Theorem 1: Any reordering transformation ϕ that preserves every dependence in a program preserves the meaning of that program.

The dependence refers to the data hazards caused by Write-after-Write (WAW), Write-after-Read (WAR) or Read-after-Write (RAW) relationships between instructions and the control hazards caused by the branches. The reordering instructions must not disrupt the established dependencies. In other words, if instruction A relies on the instruction B, then instruction A cannot be executed before instruction B in the actual execution queue. For this purpose, an *instruction dependence table (IDT)* is necessary. This table stores the information about the dependencies of each instruction on preceding ones. To enable the instructions at the bottom of the instruction sequence to be launched with the same priority as the instructions at the top of the sequence, we need an *instruction buffer (IBUF)* to temporarily hold instructions. When the *IBUF* reaches capacity, dependencies are analyzed considering all instructions within the buffer. Then an instruction is selected from the buffer and issued randomly. A given instruction only depends on the instructions that are sorted before it. Therefore, it is not necessary to wait for the buffer to be full before starting the dependence analysis. This will waste a lot of clock cycles. Whenever an instruction enters the buffer, its dependencies can be identified based on the current register occupancy. We hold the occupancy information in the *instruction register table (IRT)*. In addition to initialization,

1. why dependency has to 1 clk cycle lag compared to the entry of the instruction word in IBUF
A) at first it'll fill the IRT, but you need the value from IRT to determine the values in IDT. Hence, the lag
Check Fig 2

the process of reordering instructions can be summarized in five main steps (step II to VI) as below:

I. Initialize each storages, including the *IBUF*, the *IRT* and the *IDT*.

II. Generate a randomized entry index based on the *IDT*. The entry index represents the serial number of the instruction to be issued in the *IBUF*.

III. The *IBUF* issues an instruction according to the entry index.

IV. Store a new instruction into the *IBUF*.

V. Update the *IDT* according to the *IRT*, the input instruction and the entry index.

VI. Update the *IRT* according to the registers occupied by the input and output instructions.

VII. Repeat steps II to VI.

Algorithm 1 Reordering in the In-order Processors

Input: In-order instructions $< I_0, I_1, I_2 \cdots >$
Output: Reordered instructions $< R_0, R_1, R_2 \cdots >$

```
1:  INITIAL IBUF with NOP
2:  INITIAL IDT with 0
3:  INITIAL IRT with 0
4:  r = IIM(idt, 0)
5:  i = 0
6:  while True do
7:      r = IIM(idt, r)
8:      R_i = IBUF(r)
9:      IBUF(r) = I_i
10:     UPDATE_IDT(irt, I_i, r)
11:     UPDATE_IRT(I_i, r)
12:     i = i + 1
13: end while
```

Instruction Issue Module (IIM)

Algorithm 1 delineates the overall framework of the reordering method. This algorithm takes the sequential instructions $< I_0, I_1, \cdots, >$ as the input and generates the corresponding reordering instructions $< R_0, R_1, \cdots >$ as the output. Lines 1 to 4 of Alg.1 correspond to the initialization of step I. The *IBUF* is populated with NOP instructions, so that the buffer can realize its normal function when there are no user instructions. The operations contained within the *while* loop correspond to the five main steps. Line 7 is step II. The IIM (Instruction Issue Module) embodies the procedure for selecting a non-dependent instruction from the *IDT*. Lines 8 and 9 correspond to steps III and IV respectively. r is the entry index of the chosen instruction in the buffer. When the instruction associated with index r is output, the incoming instruction is correspondingly placed at the position denoted by r. This implies that *IDT* and *IRT* only update the data in the r-th entry in their respective tables. UPDATE_IDT and UPDATE_IRT are algorithms for updating the contents of *IDT* and *IRT* respectively, which we describe in detail in Section IV.B. It's important to note that in the hardware implementation of the processor, the operations in the loop are synchronized assignments. As each operation is synchronized, every operation employs 'r' as the value it held in the previous cycle. This means that the generation of the dependence for I_i lags by one cycle compared to its entry into the buffer. Because an instruction which hasn't computed its dependence cannot be certain about its issue status. The IIM operation is responsible for ensuring that r maintains different values before and after its calculation. Since the algorithm uses synchronous assignment, all steps for an instruction to be issued by ESM can be completed within a single clock cycle. This implies that ESM can issue one instruction per clock cycle. The original pipeline also requires at least one clock cycle to issue an instruction. If ESM needs two or more clock cycles to complete steps II to VI, it may potentially reduce the pipeline's speed. So, when we restrict ESM's functionality within a single clock cycle, for a pipeline that has reached steady state, ESM does not affect its throughput. For the whole pipeline, ESM causes latency+1. With the addition of ESM, the pipeline CPI(Cycle Per Instruction) changes from T_n/N to $(T_n+1)/N$, where N represents the number of instructions executed, and T_n represents the cycles for executing N instructions in the original pipeline. When N is very large, ESM causes almost no additional time overhead.

B. Obfuscation and Security Metrics

In the realm of hiding-like power side channel protection, the obfuscation level of the instruction sequence holds a direct influence on the scheme's security. An adversary engaged in the power side channel attack needs to gather a substantial quantity of power traces for dissecting the power characteristics inherent to these traces. The greater the obfuscation within the instruction sequence, the lower the probability that an instruction, at the same moment, occurs in multiple samples. Consequently, heightened obfuscation diminishes the likelihood of any specific instruction appearing concurrently in multiple samples. This directly reduces the probability of each instruction aligning across various samples. In turn, this diminishes the effectiveness of information that an adversary could potentially extract.

Quantifying the obfuscation and security of instruction sequence can predict the protection capability of a protection design in the face of different applications and guide the tuning of the design. However, most of the existing work does not give the corresponding evaluation metrics. For example, in [12], [14], [22], they experimentally show that the protection is able to pass the TVLA test or resist correlation analysis under certain cases, such as the AES algorithm. However, once the protection is applied to other use cases, such as the RSA algorithm, there is no way to illustrate the effectiveness of the protection. [6] proposed *RIJIDindex*, a metric based on the difference in mean values of cross-correlation, to evaluate the 'randomness' provided by the protection compared to the original sequences. *RIJIDindex* is able to evaluate the obfuscation of the sequences, but it is not able to evaluate the security. This is because security requires ensuring that the worst-case leakage is not exploited by an adversary, and the mean value reduces the impact of the worst case.

We propose metrics to evaluate the obfuscation and security of instruction sequences based on the occurrence probability of the instructions. Assuming that the length of all instruction cycles are the same. Denote the length of the instruction

segment is n, then the instructions in this segment have a total of n execution locations, where location refers to the moment when the instruction is executed in the instruction sequence, and the i-th location corresponds to the moment of execution during the i-th instruction cycle. For instruction i in the original sequence, we denote the average occurrence probability as $\overline{p_i} = \sum_{j=1}^{n} p_i(j)/o_i$, where $p_i(j)$ is the probability that instruction i will be executed at the j-th location, and o_i means the number of possible locations for instruction i. Similarly, we denote the peak occurrence probability as $m_i = \max\{p_i(j), j = 1, 2, \cdots, n\}$. The average occurrence probability represents the average probability of an instruction at all its possible locations, and the peak occurrence probability represents the highest probability of an instruction at all its possible locations. When a location of the instruction sequence executes the target instruction of the adversary, it produces an effective signal that the adversary can utilize; otherwise it is noise for the adversary. Thus, the single-point signal-to-noise ratio of an instruction sequence is $SNR = N_{target}/N_{other}$, where N_{target} is a number of times the target instruction appears at this location and N_{other} corresponds to other instructions. So $p_i(j)$ can be expressed in terms of SNR.

$$p_i(j) = \frac{N_{target}}{N_{total}} = \frac{N_{target}}{N_{target} + N_{other}} = \frac{1}{1 + \frac{1}{SNR}}$$

$p_i(j)$ is a monotonically increasing function of SNR. A larger $p_i(j)$ corresponds to a larger SNR, which means that at location j, there will be more leakage associated with instruction i. Therefore, m_i corresponds to the case of maximum leakage for instruction i. If there is no significant leakage at m_i, then at other locations, instruction i would not produce leakage either. To ensure the security of the entire instruction segment, we use the maximum value of m_i among all instructions to represent the security of this segment on the processor as *maximum peak occurrence probability* ($MPOP = \max\{m_i, i = 1, 2, \cdots, n\}$). The threshold of $MPOP$ comes from the actual measurement environment. For a new scenario, it is first necessary to conduct a leakage analysis under possible $MPOP$ values to find the $MPOP$ value without significant leakage, denoted as $MPOP_{nonleak}$. Then, analyze whether the target software will encounter situations where $MPOP > MPOP_{nonleak}$ during runtime. If $MPOP > MPOP_{nonleak}$, it is necessary to enhance protection measures to meet security requirements. The advantage of using $MPOP$ is that once the security level is determined and $MPOP_{nonleak}$ is obtained through actual measurement, as long as the program running on the processor has $MPOP \leq MPOP_{nonleak}$, security can be ensured without the need to test each individual case. $MPOP$ can tell us whether a processor is secure, but $MPOP$ alone is difficult to guide the design and optimization of protection schemes. Because $MPOP$ considers the worst-case scenario, each instruction only incorporates the probability of occurrence at one location into the calculation. If each instruction is likely to occur in 100 locations, then $MPOP$ loses 99% of the shuffling information. Or when an outlier occurs at a location, the $p_i(j)$ of the location is originally higher than others. If the overall protection adjustment is made based on this outlier, it will result in invalid overhead. Because the $p_i(j)$ values of other locations are within the security range, it is only necessary to handle the situation corresponding to the outlier. So, we use *Entire Average Occurrence Probability(EAOP)* to denote the level of obfuscation of a program segment based on the mean value of each position, where $EAOP = \sum_{i=1}^{n} \overline{p_i}/n$. The smaller the $EAOP$, the higher the level of obfuscation of the program segment and the better the scheme's ability to shuffle. $EAOP$ represents the average obfuscation level of the entire instruction segment running on the processor. Its significance lies in guiding protection modifications to eliminate the impact of outliers in $MPOP$. In the experimental section, we describe how to select appropriate ESM parameters using $MPOP$ and $EAOP$.

IV. The Detail of ESM

This section describes in detail our implementation of shuffling in the sequential processor. The design is named as the External Shuffling Module(ESM). ESM is designed to be an independent hardware module that does not rely on the original processor structure and does not require support from an extended instruction set or compiler. ESM can be embedded directly into the original design as an IP.

An IP in the context of hardware design stands for Intellectual Property. It refers to a pre-designed and reusable block of logic or functionality that can be integrated into larger hardware designs

A. Structure of ESM

As depicted in Fig. 1, ESM operates between the front-end and back-end, intercepting signals from the decoding and the issue/execution stages. These signals are categorized into two types: control signals and data signals. Data signals are related to instructions and vary with different instructions. Since the ESM is deployed after the decoder, data signals specifically refer to the information such as instruction PC, registers, immediate numbers, and instruction types (control flow/arithmetic/memory access instructions) parsed after decoding. Control signals, on the other hand, are signals between the original pipeline front-end and back-end, excluding data signals. These signals are unrelated to instructions. They connect various pipeline components and maintain the normal operation of the processor. Taking the CVA6 RISCV processor as an example, it sets up a handshake signal between the decoder and the issuer. When the decoder is ready to decode an instruction, it sends a *valid* signal to the issuer. Upon receiving the *valid* signal, the issuer returns a *ready*

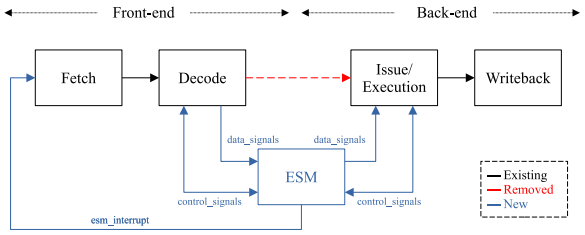


Fig. 1. Processor pipeline with ESM

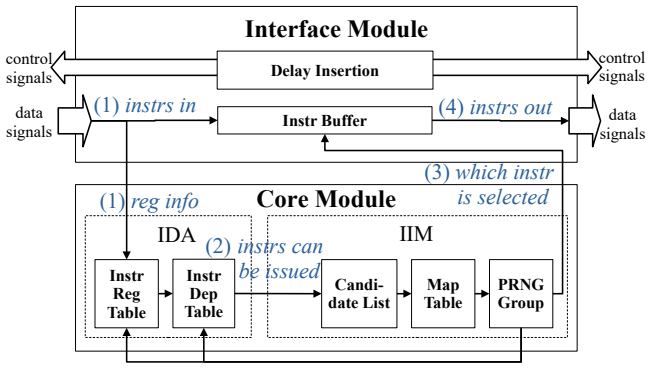


Fig. 2. Structure of ESM

signal indicating that it is ready to receive data normally. Only when the decoder receives the *ready* signal will it proceed with the next decoding, thus ensuring the normal operation of the pipeline. The *valid* and *ready* signals here are examples of data signals. Of course, both data and control signals differ in different microarchitectures. To enable the ESM to be applied across different microarchitectures without modifying the components in the original pipeline, we divide the ESM into the interface module and the core module. The interface module implements three functions. Firstly, it stores and transmits data signals. The interface module has an internal *IBUF* that stores a set of data signals in each instruction cycle and transmits another set of data signals. Secondly, it forwards control signals without changing their values, providing real-time forwarding for subsequent pipeline components to use. If there is a decoding 'valid' signal in control signals as the example above, the ESM drives the core module with this signal; otherwise, it drives the core module with the clock signal. Thirdly, the interface module parses the register information from the data signals and provides it to the core module for use. The core module implements the main functionality of the ESM. It takes the register signals provided by the interface module as input and outputs a random entry index. The core module comprises two primary components: (a) *instruction dependence analysis(IDA)* and (b) *instruction issue module(IIM)*.

Fig. 2 illustrates the structure of ESM and the information flow in ESM. The IDA consists of an *IRT* and an *IDT*. (1) is the first flow of instruction information into the ESM. The data signals passed from the decoder are stored unchanged in *IBUF*, while the register information of the instruction is sent to *IRT* and *IDT*. Based on the newly entered register information, both two tables will update their contents. The updated table data is then passed to the IIM, informing the IIM which instructions can be issued in the current cycle. This corresponds to the data flow marked as (2) in Fig. 2. Then ESM executes the IIM. The IIM contains a *candidate list*, a *mapping table* and a *pseudo-random number generator group (PRNG Group)*. The data passed into the IIM from the *IDT* is stored in the *candidate list*. The index of the launchable instructions are discontinuous, and the IIM uses a *mapping table* to change the indexes to consecutive integers starting from zero. The PRNG Group is used to 'choose' an instruction to be issued in the *IBUF*. (3) identifies which instruction is selected for issuance, and the data transmitted is the entry index mentioned in Section III. Finally, the *IBUF* uses the entry index to issue the instruction contained in the corresponding entry, transmitting it to the next stage of the pipeline, which is marked as (4).

As described in Section 1, the protection mechanism discussed in this paper is applicable to scalar processors. The ESM can be applied to any scalar pipeline for two reasons: 1) The ESM is deployed after the decoder, which is a necessary component in all microarchitectures. From an overall perspective, the ESM forwards signals output by the decoder (both data and control signals) without generating additional signals. For other components of the pipeline, the ESM is transparent and can be considered part of the decoder. This makes the ESM applicable to any microarchitectural structure. 2) The ESM is divided into interface and core modules. The interface module specifically handles differences in control and data signals caused by microarchitectural variations. Since it only forwards signals, the ESM only needs to adjust the input and output signal widths of its own interface module. The register information required by the core module is explicitly present after passing through the decoder, and this is true for any microarchitecture.

B. Instruction Dependence Analysis

Instruction dependence analysis is the most significant process of ESM telling ESM which instructions can be dispatched directly without waiting for the execution result of other instructions. When the order of instruction execution is disrupted, errors in execution results may occur due to data conflicts. To ensure the integrity of processor functionality, ESM analyzes the data dependence relationships of instructions to identify instructions that can be executed out of order. To do this, ESM builds two tables, the *instruction register table* and the *instruction dependence table*, that store the dependence relationships between instructions. The dependence table marks the currently independent instructions and only these instructions can be transmitted to the next stage to avoid data conflicts.

IRT_RD, IRT_TS : no of words = no of possible instruction stored in IBUF, each word = no of registers that instruction can depend on.
IRT_CTRL: 1 word = 1 bit for each instrucion in IBUF

1) Instruction Register Table: The table has three sub-tables. The first one holds the state of destination registers called *IRT_RD*, the second holds the state of source registers called *IRT_RS*, and the last one called *IRT_CTRL* holds the information weather the current instruction is a control flow instruction. If an instruction uses one register, the corresponding element in the table is set to 1. The size of both the *IRT_RD* and the *IRT_RS* is $regnum * bs$, where $regnum$ is the number of the general registers in the target ISA, bs is the size of the *IBUF*. The *IRT_CTRL* is a list with bs elements. Whenever an instruction enters the *IBUF*, the *IRT* updates its content. The process of updating the *IRT* is described in Algorithm 2 as UPDATE_IRT. UPDATE_IRT has two inputs, where I_c represents the instruction entering the *IBUF* this cycle, and r represents the entry index occupied by I_c in the *IBUF*. The index r also corresponds to column r in the *IRT*. Depending on the number of the source and destination registers occupied by I_c, the corresponding element of column r is set to 1 to

Algorithm 2 UPDATE_IRT

Input: I_c, r
Output: *IRT*

```
for i = 0 : regnum − 1 do
  if i == I_c.rs1 or i == I_c.rs2 then
    IRT_RS[i][r] = 1
  else
    IRT_RS[i][r] = 0
  end if
  if i == I_c.rd then
    IRT_RD[i][r] = 1
  else
    IRT_RD[i][r] = 0
  end if
end for
if I_c.ctl == 1 then
  IRT_CTRL[r] = 1
else
  IRT_CTRL[r] = 0
end if
```

indicate the occupation of the registers by the instruction. If the instruction is a control flow instruction, such as jump or branch, the element of the *IRT_CTRL* corresponding to r is set to 1.

IDT: no of words = number of instructions
in each word, bit represents the dependence of the current instruction on the instruction specified by the bit

2) Instruction Dependence Table: The dimensions of *IDT* amount to $bs \times bs$. The element at the (i, j) position within the table signifies whether the i-th instruction in the *IBUF* relies on the j-th instruction. If a dependence is present, the element is assigned the value 1; otherwise, it is designated as 0. Upon the entry of an instruction into ESM, the corresponding row in *IRT* is referenced to determine the register number it employs. ESM subsequently writes the content of the current cycle's *IRT* row into the row aligned with the index of the current instruction in *IDT*. The update process of the *IDT* is described in detail using Algorithm 3, UPDATE_IDT. The inputs I_c and r are consistent with those in UPDATE_IRT. For the resolution of RAW, instructions using source registers necessitate querying IRT_RD according to the specific register number. In the case of WAW, if the current instruction employs the destination register, querying the IRT_RS is essential based on the relevant register number. Similarly,

Algorithm 3 UPDATE_IDT

Input: *IRT*, I_c, r
Output: *IDT*

```
for i = 0 : bs − 1 do
  idt[i][r] = 0
  if i != r then
    raw1 = IRT_RD[I_c.rs1][i]
    raw2 = IRT_RD[I_c.rs2][i]
    war = IRT_RD[I_c.rd][i]
    waw = IRT_RS[I_c.rd][i]
    idt[r][i] = raw1|raw2|war|waw|IRT_CTRL[i]
  end if
end for
```

my correction

```
Example code:

I0: lw  a0, s0
I1: lw  a1, s0
I2: add a2, a0, a1
I3: add a3, a0, imm
I4: ...  ...
```

Fig. 3. Example code

ESM queries IRT_RD according to the current destination register, if there exists WAR. In terms of control hazards, every instruction into the ESM needs to read the value of IRT_CTRL. Ultimately, ESM consolidates the outcomes of the aforementioned queries regarding the four hazards and stores them within the row corresponding to the current instruction's index in the dependence table.

3) A Toy Example: The integration of Algorithm 1 with our proposed *IDT* and *IRT* structure can be understood through a toy example. The Fig. 4 depicts the process of dependence analysis performed by ESM based on *IDT* and *IRT*. The example code is shown in Fig. 3. The code snippet comprises four entries, consisting of two load instructions and two add instructions. I_2 relies on both I_0 and I_1, I_3 depends solely on I_0. For this demonstration, let's set bs to 3. The Fig. 4 delineates the progression of *IDT* and *IRT* states while I_2 and I_3 enter(exit) ESM. Assume that all instructions already existing in the *IBUF* are NOP instructions, and in the first three cycles of the example, this three NOPs are issued in order from entry index 0, 1, and 2 of the *IBUF*. So, I_0, I_1, I_2 correspond to entry index 0,1,2 in the toy example. In cycle 0, I_0 enters the ESM. I_0 is the first instruction that does not depend on other instructions. So, the contents of the *IDT* are not modified. Since the index of I_0 is 0, the column 0 of the *IRT* is updated based on the register occupied by I_0. In cycle 1, I_1 enters the ESM. All the registers of I_1 are different from the destination register of I_0, so I_1 does not depend on I_0. Therefore, there is no need to modify the contents of the *IDT* in cycle 1 either. For the *IRT*, column 1 is modified according to the entry index of I_1. In cycle 2, I_2 enters the ESM. The green rectangle represents the updating process of the *IDT* in cycle 2. This process can be divided into two steps. We have previously assumed that the entry index of I_2 is 2, which means that the index of the instruction selected for issue in cycle 1 is also 2. Therefore, the first step is to assign the elements in the column 2 to 0. The instruction with index 2 in the *IBUF* is waiting for issue and needs to clear its dependence relationships. The second step involves constructing the dependence relationships for the newly entered instruction I_2. The horizontal elements of the *IDT* represent the dependencies of the corresponding instruction on other instructions, so the elements in the row 2 of the *IDT* is updated. I_2 utilizes three registers, a0, a1 and a2. Therefore, the *IDT* needs to access the three corresponding rows in the *IRT_RD*. Additionally, the destination register of I_2 is a2, it is necessary to access the row a2 in the *IRT_RS*. Note that due to synchronized assignments, the *IDT* can only access the *IRT* state of cycle 1. The outcome of this query is subsequently written to row 2 of the *IDT* as shown in Fig. 4. It's worth mentioning that when the *IDT* queries the *IRT*,

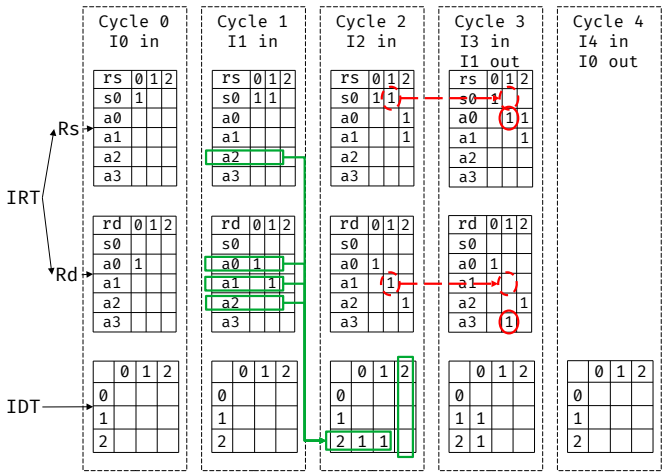


Fig. 4. A toy example

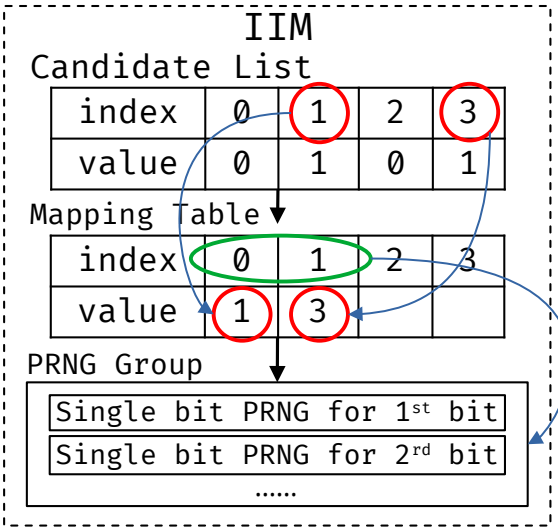


Fig. 5. Structure of IIM

column 2 of the *IRT* isn't included. The rationale behind this is that the *IRT* for cycle 1 encompasses register occupancy information about the impending cycle 1 instruction, which remains unaffected until cycle 2. The registers occupied by forthcoming instructions shouldn't influence subsequent the *IDT* flushes. Consequently, the *IDT* is updated during cycle 2 without considering column 2 of the *IRT* in cycle 1. From cycle 0 to cycle 2, no instructions other than NOP are issued, elements with a value of 1 only increase and do not decrease in the *IRT*. In cycle 3, I_3 enters the ESM. I_0 and I_1 are independent of other instructions, and they can both be issued in cycle 3. In this example, we let I_1 issue in this cycle. The red circle signifies the process of updating the *IRT* in cycle 3. Initially, the column 1 of the *IRT* is set to 0 based on the index of I_1. Subsequently, the registers of I_3 is set to 1 in column 1. The update process for the *IDT* is the same as cycle 2. We set the elements of column 1 to 0, and write the rows corresponding to registers a0 and a3 from the *IRT_RD* (used by instruction I_3), and the row corresponding to a3 from the *IRT_RS* (the destination register of I_3) into the row 1 of the *IDT*.

C. Instruction Issue Module

In this section, we implement the IIM to decide the issue order according to the *IDT*. As shown in the Fig. 5, IIM consists of three parts:*candidate list*, *mapping table* and *PRNG group*. *Candidate List.* This component holds the result of instruction dependence analysis, whose index of each element corresponds to the instruction index in the *instruction buffer*. For any index, if all the elements of the corresponding index row in *IDT* are 0, then the value of the index in the *candidate list* is set to 1, indicating this instruction can be issued. Otherwise, the element is set to 0.

Mapping Table. The index of launchable instructions in the *candidate list* are non-consecutive and randomly distributed. Additional cycles are required when the random number does not hit a valid value in *candidate list*. By using a *mapping table*, ESM maps the non-consecutive index of launchable instructions to the consecutive indices in the *mapping table*. As shown in Fig. 5, the value of *mapping table* is the index of *candidate list*.

PRNG Group. We use a group of PRNG to choose one index of *mapping table*. The *PRNG Group* consists of several single-bit pseudo-random number generators. Each single-bit generator generates a random number of 0 or 1. By concatenating the outputs of multiple single-bit generators, a random number can be obtained. For example, in the Fig. 5, only two indexes are valid in *mapping table*. So, we need just one single-bit generator to choose index from 0 and 1.

Every time the *IDT* has been flushed, IIM records the non-dependence instructions in the *candidate list*, and maps the index of the *candidate list* to *mapping table* immediately. Then *PRNG Group* generates a random number based on the effective length of the *mapping table*. *Mapping table* transfers the value to the buffer according to the random number.

D. Delay Insertion

ESM enables O-o-O execution of instructions without dependencies. However, if there are 'strong' dependencies between instructions in the target program segment, the program may not receive effective protection under ESM framework. Here, 'strong' dependencies refer to long dependency chains between instructions. For example, consider four instructions a, b, c and d, where a is dependent on b, b is dependent on c, and c is dependent on d. We refer to them as having 'strong' dependencies because the length of their dependency chain is equal to the total number of instructions, necessitating execution in the order $< d, c, b, a >$. Conversely, if only a depends on b and c depends on d, then they are considered to have relatively 'weak' dependencies. Segments with 'strong' dependencies have fewer instructions that can be reordered, resulting in a larger $EAOP$. In the most extreme circumstances, it's plausible that all instructions within the *IBUF* possess dependencies, rendering instruction reordering unfeasible. To counterbalance the potential leakage vulnerability of the power

channel under such conditions, we introduce delay into the ESM to reduce the $MPOP$. Previous work has proposed various methods for adding delay. [23] uses a combination of randomized scheduling, randomized instruction insertion and randomized pipeline-delay to resist side-channel attacks. [22], [24] propose hardware solutions to insert dummy instructions in runtime. Since ESM first reordering the original instruction sequence, we do not need delays as complex as those in [23], which would incur significant overhead. The addition of dummy instructions requires altering control signals. 'Strong' dependent program segments do not always exist. Thus, we focus more on addressing outliers in $MPOP$. A *delay parameter* is set in ESM to limit the maximum number of delays to be inserted at a time. Whenever there is only one candidate instruction in the *candidate list*, we insert delays randomly before and after it. When the delay is needed, the ESM holds the control signals low until the delay number of instruction cycles have passed, then forwards the control signal to the next stage of the pipeline. The number of delays varies between 0 and the *delay parameter*, which is generated by the *PRNG Group*. The choice of the delay parameter is related to the dependencies within the current program segment. A larger delay parameter is needed if the segment itself has 'strong' dependencies. The ultimate goal is to achieve a security value for the $MPOP$ of the program segment.

V. Experiments

This section presents the experiments and the analysis of ESM protection scheme. We demonstrate how to find the threshold $MPOP_{nonleak}$ for $MPOP$ through leakage assessment. And after obtaining $MPOP_{nonleak}$, we use $MPOP$ and $EAOP$ to adjust the parameters of ESM in practical applications. With the suitable parameters, we evaluate the performance and area overhead of ESM and conducte an attack on a real AES case.

A. Experiments Setup

The processor core used in the experiments comes from OpenHWGroup's CVA6 [25], which is a 6-stage, single-issue, sequential processor that implements the 64-bit RISC-V instruction set [26]. We implement the CVA6 with ESM protection on a SAKURA-GIII board. The operating frequency of the development board is 12MHz. We evaluate the power leakage and conduct the actual secret information recovery attacks on the power consumption trace of the board. The power consumption is measured using a PicoScope 5244D oscilloscope with the sampling rate of 500MHz. The Vivado EDA is used to analyze the area overhead of ESM and to perform behavioral simulation for evaluating the performance loss.

B. Evaluation

We perform leakage detection of the processor under ESM protection via TVLA and implement two different types of attacks to evaluate the security of ESM under different $MPOP$s.

1) TVLA: This experiment is designed to illustrate the ability of ESM to reduce data-related leakage basing on TVLA [19].

Bench. To conduct leakage analysis for different $MPOP$ values while excluding the influence of outliers, we construct a special set of independent instructions that allow each instruction in the *IBUF* to be executed in any instruction cycle. From a statistical perspective, each location's $p_i(j)$ is equal to $1/bs$. By adjusting the value of bs, we can obtain different running program segments for different $MPOP$ values. The source registers of every instruction are the same. But the destination register of each instruction is different to ensure that there are no dependencies between them. We use this bench to evaluate the operands-related leakage in the program segment for different $MPOP$s. For the two data sets in TVLA, we alternate between acquiring fixed data and random data. The operands used by the processor are generated by the host computer. The total number of observation traces is 2 million.

Results and analysis. Fig.6,7,8 presents the results of TVLA experiments conducted on the processor. In Fig.6, the results for the original processor without ESM protection show a peak t-score at around the 20,000 point and t-score remains around 300 at the 40,000 point. The t-score keeps below 5 until the 20,000 point because the processor is not powered up at this point. So there is no leakage related to the input data. The processor is powered up at 20,000, resulting in a trace difference due to processor initialization, where values are written to the registers. After initialization, the processor starts to fetch and run at 40,000 points. Fig.6 indicate that an unprotected processor generates high and data-related leakage. Fig.7,8 show the results of TVLA with ESM protected processor. The number of points in Fig.6 is greater than that in Fig.7, 8 to visually demonstrate the difference between the processor running and not running. There indeed exists data-related leakage during operation. In the protected processor, we only captured the moments when it was running for analysis. In the experimental setup of TVLA, $MPOP = 1/bs$. When the *IBUF* size is larger, $MPOP$ becomes smaller. From Fig.7 and 8, it can be seen that a smaller $MPOP$ corresponds to a smaller t-score. In our experimental environment, when the $MPOP$ is 0.0164, the t-score is within 5. So the $MPOP_{nonleak}$ is 0.0164 in our experiment. The experimental results demonstrate that ESM can significantly reduce the

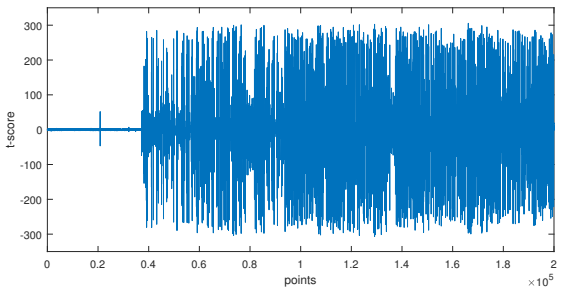


Fig. 6. Leakage assessment of the unprotected processor with TVLA

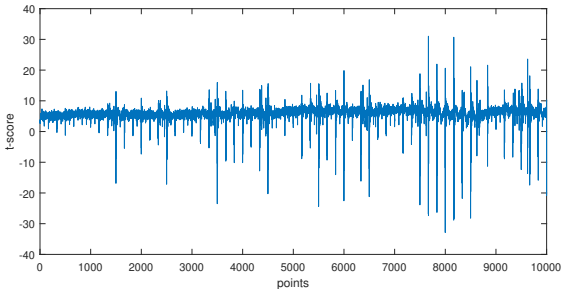


Fig. 7. Leakage assessment of the ESM protected processor with TVLA and the $MPOP$ = 0.0670

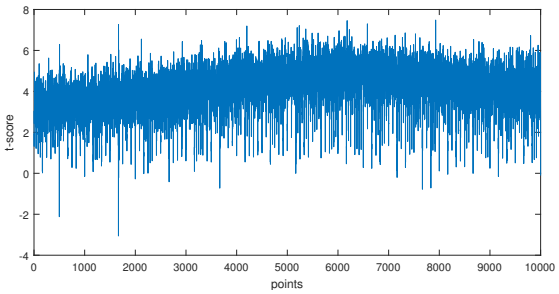


Fig. 8. Leakage assessment of the ESM protected processor with TVLA and the $MPOP$ = 0.0164

amount of data leakage from the processor, making it more challenging for an adversary to successfully attack. And when the $MPOP$ reaches 0.0164, there is no significant leakage from the ESM-protected processor.

2) Deep learning based side channel attack: The experiment in this section verifies whether the ESM protected processor can resist profile-based attacks in the case of no leakage with TVLA. The research of Rastogi et al. [18] shows multi-layered perceptrons (MLP) are the best models for attacking hardware implementation in deep learning based side channel attacks compared to convolutional neural networks (CNN) and recurrent neural networks (RNN). Since our scheme is hardware implemented, we use the MLP model in [18] to perform an attack. The model in the experiment consists of six layers. The 'Input Layer' takes inputs of shape (128, 1). The 'Hidden Layers' consist of two fully connected layers, each with 64 neurons and SELU activation function. The 'Dropout Layer' is after the second fully connected layer with a drop rate of 0.2. The 'Flatten Layer' flattens the multi-dimensional input into a one-dimensional vector. Finally there is an 'Output Layer' which is a fully connected layer with 256 neurons and Softmax activation function for classification tasks. Additionally, the model uses the Adam optimizer with the following parameters: (1) learning rate = 0.0003, (2) beta_1 parameter = 0.9, (3) beta_2 parameter = 0.999 and (4) epsilon = 1e-8. The parameters set for the training process are as follows: (1) the number of training epochs is 30, (2) the validation split ratio is 0.125 and (3) the batch size is 64.

Bench. The testbench used in this experiment is the same as what used in the TVLA. We select one XOR instruction in the segment as the target instruction. Each time the processor runs, a random operand is written to the RS1 register of the XOR instruction and a fixed operand is written to the RS2 register. The fixed operand in the RS2 register is treated as the secret information. For other instructions in the segment, they do not reuse the RS2 register of the target instruction to ensure the power consumption related to the secret information and the power consumption of other data are independent of each other. The operands in this experiment are all randomly generated. The program segment repeatedly runs 1.1 million times.

Data pre-processing. In our early confirmatory experiments, we found that the noise interference generated by the processor is significant. If we use raw data without pre-processing, even an unprotected processor cannot successfully recover the secret information. Therefore, in this experiment, we do the pre-processing on the raw power consumption. The specific pre-processing steps are as follows:

1) Select point-of-interest. The sampling rate is 500MHz, and the operating frequency of the development board is 12MHz. About 42 points are collected per cycle. In order not to lose information, two cycles before and after the XOR original running moment are retained during data analysis. And the length of the traces for analysis has 128 points.

2) Acquire the differential power trace. we divide all traces into two categories based on whether the LSB of XOR RS1 operand is 0 or 1. We take the mean of traces within each category and subtract the means between the two categories. The mean is used to reduce the noise of the differential trace.

3) Convert the differential trace to the frequency domain. Fast Fourier Transform is used to transform the differential trace into frequency field to select the frequency band where actual leakage occurs.

4) Denoise all traces. Based on the filtering results in the frequency domain of the differential trace, we transform all power consumption traces into spectrograms and retain only the data within the leakage frequency band (which is the peak in the differential trace). Data analysis can be performed directly in the frequency domain. However, the data in the time series is more intuitive, the traces are transformed back to the time domain for analysis finally.

Attack. One million pre-processed power consumption traces are used for training the model, while the remaining traces are used for the attack. The target secret information to be attacked is the operand of the RS2 register of the XOR instruction, with the intermediate value (IV) being the result of $rs1 \oplus rs2$, which is related to the physical power consumption. The training data is classified and labeled into 256 categories based on the lower eight bits of the IV. This results in 256 guess values for the lower eight bits of the secret information. For each guess value (gv), we compute $rs1 \oplus gv$ and label the attack data. Finally, the success rate of each gv is ranked based on the matching between the training and attack data.

Results and analysis. Fig.9 and 10 illustrate the FFT transformation results of the power consumption traces after preprocessing. The spectrum plot of the original processor's power consumption traces shows a significant energy signal near the 5.5 MHz band, except for the noise signal at high

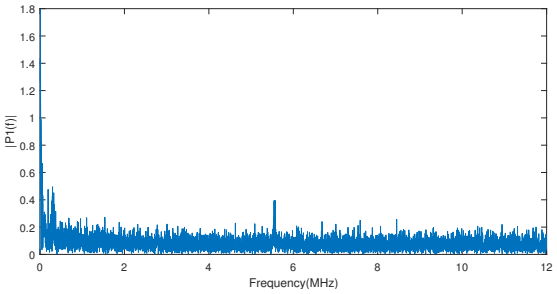


Fig. 9. Single-sided amplitude spectrum of the differential power trace of unprotected processor

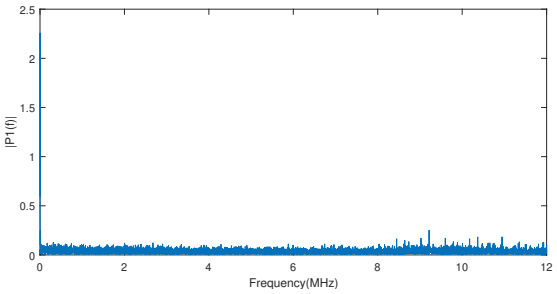


Fig. 10. Single-sided amplitude spectrum of the differential power trace of ESM protected processor and $MPOP$ = 0.0164

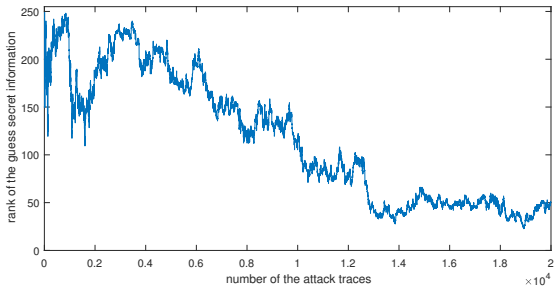


Fig. 11. The guess secret information $rank$ of ESM protected processor based on the MLP attack and $MPOP$ = 0.0670

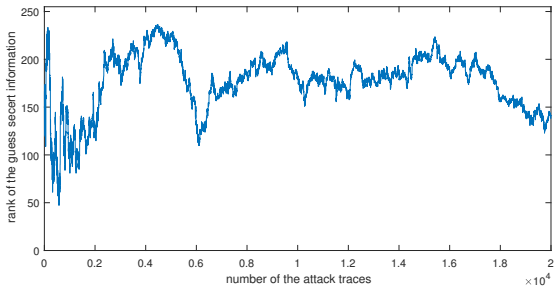


Fig. 12. The guess secret information $rank$ of ESM protected processor based on the MLP attack and $MPOP$ = 0.0164

frequencies. However, after ESM protection, there is no obvious signal peak in the frequency domain. The frequency plot demonstrates the reduction of information in the power consumption due to ESM protection. For the attack, we select the information near the 5.5 MHz band in Fig. 9. Fig.11 and 12 display the *rank* of the correct secret information guess in the space of all secret information guesses for ESM protected processor. For the unprotected raw processor power consumption, we use only 10K traces for training. As the number of attack traces increases, the *rank* of the correct guess gradually decreases. When the attack trace reaches 6000, the correct guess has ascended to the top *rank*. For the processor with ESM protection, the number of training traces reaches 1 million. When the $MPOP$ is 0.0670, the *rank* of the correct guess also decreases as the number of attack traces increases. When the number of traces reaches 13K, the *rank* of the correct guess starts to converge and stabilizes around 50. However, when the $MPOP$ is 0.0164, the *rank* of the correct guess does not show a significant decreasing trend. The results of Fig.9, 10 demonstrate ESM's ability to "hide" the power leakage of the secret information, while the results of Fig.11, 12 show ESM's capability to effectively combat deep learning side channel attacks, providing effective protection against secret information leakage under 1 million training traces. When the $MPOP$ drops below 0.0167, it can be guaranteed that the program will not be attacked by DL attack with MLP model.

3) Correlation power analysis.: The experiment in this section verifies whether the ESM protected processor can resist non-profile attacks in the case of no leakage with TVLA. CPA is a traditional non-profile attack that uses the Pearson correlation coefficient as the distinguisher. In this experiment, we use the same bench and attack flow as the DL attack. The difference is that CPA does not require a modeling process. We calculate the correlation coefficient between the guess intermediate value $rs1 \oplus gv$ and the actual power consumption based on each guess gv.

Regardless of whether the $MPOP$ is 0.0670 or 0.0164, the CPA is not able to effectively distinguish the correct secret information when the trace number reaches one million, as shown in the Fig.13 and 14. The experiment shows that ESM protection can be effective against traditional non-profiling attacks such as CPA.

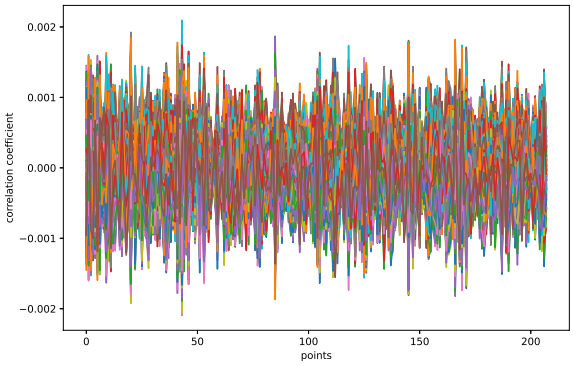


Fig. 13. Pearson correlation coefficient of ESM protected processor and $MPOP$ = 0.0670

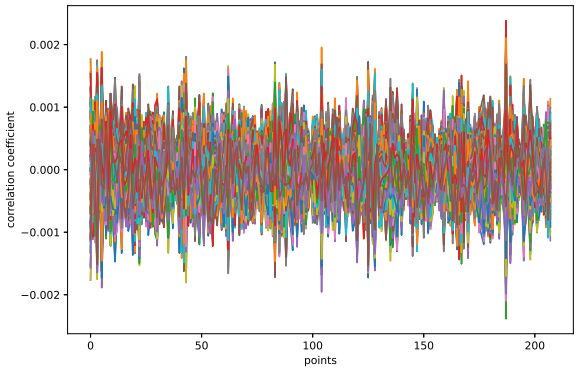


Fig. 14. Pearson correlation coefficient of ESM protected processor and $MPOP$ = 0.0164

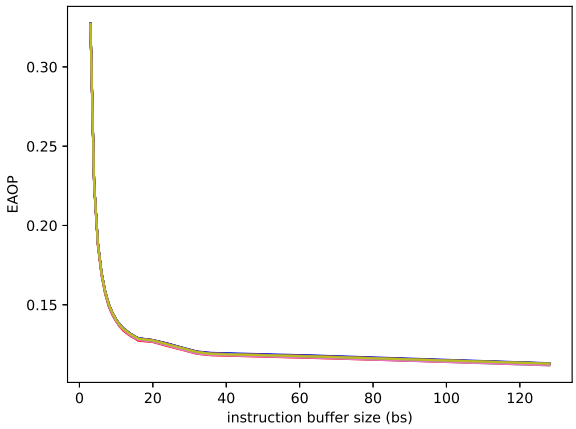


Fig. 15. $EAOP$ of SPEC2017 benches

C. Overheads

The strength of ESM protection can be varied by adjusting the size of the *IBUF* and the number of delays. The larger the *IBUF*, the greater the area overhead; the more delays, the greater the time overhead, and consequently, the higher the protection strength. Each user's usage scenario is different. Two factors that affect users' protection needs are the signal-to-noise ratio of the processor's operating environment and the programs running on the processor. If the operating environment has a low signal-to-noise ratio, users may only require lower protection strength to achieve the necessary security. If the processor is used in scenarios with programs that have low instruction dependency, the required security can be achieved with a smaller *IBUF* and fewer delays. In summary, configurability allows users to minimize area and time overhead as much as possible according to their usage scenarios and security requirements. In this section, we show how to choose the suitable bs and number of insertion delays, and evaluate the area and time overhead when oriented towards real program segments. The benches used in the experiment are taken from SPEC2017.

To pick an appropriate bs, we set the delay number in ESM to 0 and observe the $EAOP$ values that each bench can achieve after shuffling. Fig.15 shows the change in the $EAOP$ values of each bench when bs goes from 3 to 128. As can be seen from the figure, the $EAOP$ values are very close across the benches. The lack of difference between benchmarks does not affect the choice of bs. Because Fig.15 aims to demonstrate the overall trend of their changes rather than the differences. When bs is less than 16, the $EAOP$ has a significant decrease with the increase of bs. When bs is greater than 16, the buffer size required increases exponentially, although the $EAOP$ continues to get smaller. Therefore, for the SPEC2017 benches, we set bs to 16. This not only makes full use of the program's disorderable instructions, but also keeps the area overhead within a reasonable range.

On the basis of $bs = 16$, the delay number setting is performed to reduce the final $MPOP$ value so that the $MPOP$ can reach the $MPOP_{nonleak}$. We test the effect of different delay parameters on $MPOP$. Taking *500.perlbench_r* as an example, when the delay parameter is set to 3, the $MPOP$ drops to 0.0344 (less than 0.0670), which is already able to withstand the traditional non-profile attack. When the delay parameter is set to 4, the $MPOP$ is 0.0125 (less than 0.0167), which is able to counter the deep learning attack. In fact, if the secret information has numerous bits, such as an 128-bit key, setting the delay parameter to 3 is sufficient. When the $MPOP$ is 0.0670, the MLP attack reduces the rank of the correct secret information to about 50. This means that the adversary still needs about 50 guesses to recognize the correct one. Recovering 8 bits of the secret information at a time would require 16 attacks to restore the complete information. Ultimately, 50^{16} (greater than 2^{80}) guesses are required. Fig. 16 shows the extra time overhead of different SPEC2017 benches with ESM protection. When $bs = 16$ and the delay parameter sets to 3, the average additional time overhead for SPEC2017 is approximately 12.12%. This overhead is based on all instruction protection. If, like related work, protection is limited to arithmetic instructions, the time overhead can significantly decrease, with an average additional increase of 2%.

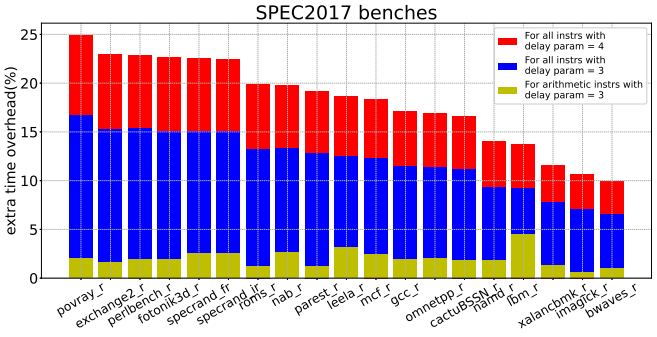


Fig. 16. Time overhead for SPEC2017

With $bs = 16$ and delay parameter is 3, the resource of ESM is 403 LUTs and 77 registers. The original cva6 processor uses 32,635 LUTs and 20,988 registers. The area overhead of ESM is about 1.23% of the original design. We show the

TABLE I
AREA OVERHEAD

work	target	modifications	area	time
This work	entire program segment	pipeline	1.23%	12.12%
XDIVINSA [14]	specific instructions	compiler, ISA, coprocessor	4%	10%-20%
ERIST [22]	cryptographic programs	pipeline, register files	8%	21.09%
Conjoin [12]	cryptographic programs	compiler, pipeline, register files	7%	4%

area and time overhead of ESM and compare it with existing related work in Table I. Besides area and time overhead, we also summarized two differences between this work and other similar works in Table I. ”Modifications” refers to the changes needed to add related protections to an existing processor. This work aims to implement a plug-in style protection. It separates hardware protection from software and minimizes modifications to the microarchitecture as much as possible. The advantage of this is that the application scope of the protection is broader. ”Target” refers to the target of the protection. The protection of this work covers all instructions running on the processor, not just cryptographic algorithms or arithmetic instructions. If the target is some specific goals, then the software user needs to modify their own program, through the compiler or operating system to tell the processor about their sensitive assets. This brings more ”modifications” and also requires more professionals to apply the protection.

D. Attack on AES

Finally, we conduct an actual attack on AES to verify the practical significance of ESM. The ESM is configured with the *IBUF* size of 16 and the delay parameter of 3. The target AES is implemented using T-tables, and we attack the output of the first round. Fig.17 and Fig.18 respectively present the results of TVLA and CPA. It can be seen that when running AES, the processor protected by ESM does not exhibit significant leakage, and it is unable to distinguish between 256 guessed keys.

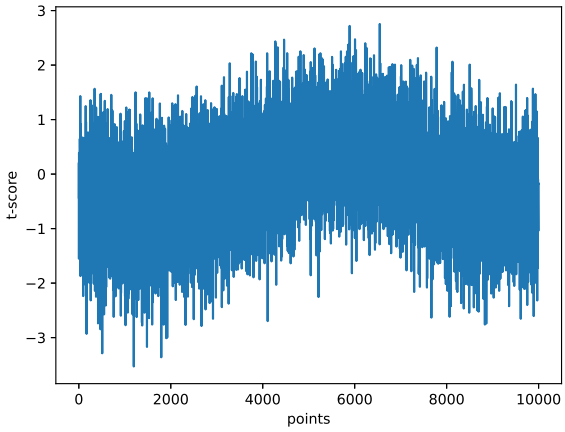


Fig. 17. TVLA of ESM protected processor running AES

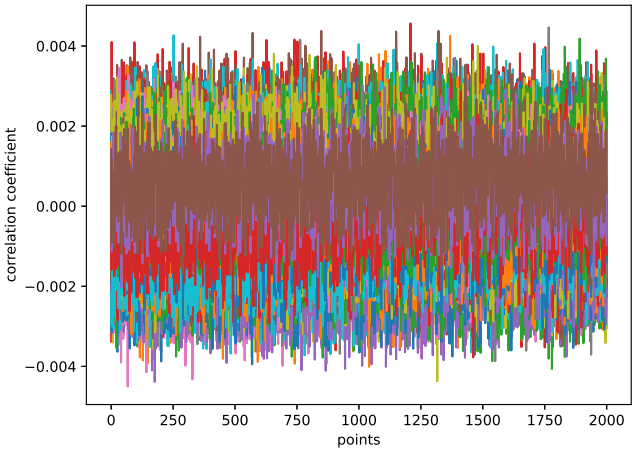


Fig. 18. Pearson correlation coefficient of 256 key guess of AES running on the ESM protected processor

VI. CONCLUSION

In this article, we propose ESM, a plug-in power protection based on the hiding strategy against power side-channel leakage hazards on sequential processors. We utilize changeable-order instructions in program segments to reorder the entire instruction sequence to reduce the protection overhead. ESM is designed to be placed between the front and back ends of the pipeline, and can be started and stopped at any time without modifying the rest of the microarchitecture. We implement the full protection in hardware without requiring the user to define additional instruction set and compiler functions to enable ESM. The protection capability of ESM is not limited to cryptographic algorithms, and full-instruction protection can be implemented for a variety of programs. In addition, we propose metrics for instruction sequence obfuscation and security capabilities to quantify the differences in protection when oriented to different applications. ESM is flexible and can be used to adjust the realization parameters according to changes in metrics. In the experiments, we perform an actual implementation of ESM protection on a SAKURA-GIII board for a RISC-V processor. Our scheme consumes less resource overhead compared to the same type of work, with an area of about 1.23% of the original processor and a time of about 12.12%. It is able to successfully resist correlation analysis and MLP-based deep learning attacks at 1 million traces.

Future work. Presently, ESM is exclusively implemented in scalar processors and cannot be disabled which make all programs suffer additional timing overhead. Future work aims to extend ESM to O-o-O processors, further enhancing their resistance against power leakage, and explore more effective activation methods to make ESM more flexible. It’s worth noting that although ESM’s size is configurable, it requires determination prior to implementation. Future efforts will explore dynamically configurable processor protection schemes at runtime.

ACKNOWLEDGMENT

This work was supported in part by the National Key R&D Program of China (No.2022YFB3103800).

REFERENCES

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 1999, pp. 388–397.

[2] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008, vol. 31.

[3] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 1999, pp. 398–412.

[4] H. Groß, M. Jelinek, S. Mangard, T. Unterluggauer, and M. Werner, "Concealing secrets in embedded processors designs," in *Smart Card Research and Advanced Applications: 15th International Conference, CARDIS 2016, Cannes, France, November 7–9, 2016, Revised Selected Papers 15*. Springer, 2017, pp. 89–104.

[5] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*. Springer, 2003, pp. 463–481.

[6] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "Randomized instruction injection to counter power analysis attacks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 3, pp. 1–28, 2012.

[7] ——, "Rijid: Random code injection to mask power analysis based side channel attacks," in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 489–492.

[8] S. Patranabis, D. B. Roy, P. K. Vadnala, D. Mukhopadhyay, and S. Ghosh, "Shuffling across rounds: A lightweight strategy to counter side-channel attacks," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 440–443.

[9] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 431–446.

[10] S. N. Dhanuskodi and D. Holcomb, "Enabling microarchitectural randomization in serialized aes implementations to mitigate side channel susceptibility," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 314–319.

[11] A. G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson, "An architecture-independent instruction shuffler to protect against side-channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–19, 2012.

[12] E. Aerabi, A. E. Amirouche, H. Ferradi, R. Géraud, D. Naccache, and J. Vuillemin, "The conjoined microprocessor," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 67–70.

[13] Y. Chen, A. Hajiabadi, R. Poussier, A. Diavastos, S. Bhasin, and T. E. Carlson, "Mitigating power attacks through fine-grained instruction reordering," *arXiv preprint arXiv:2107.11336*, 2021.

[14] T. H. Pham, B. Marshall, A. Fell, S.-K. Lam, and D. Page, "Xdivinsa: extended diversifying instruction agent to mitigate power side-channel leakage," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2021, pp. 179–186.

[15] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel, "Mutual information analysis: A generic side-channel distinguisher," in *Cryptographic Hardware and Embedded Systems–CHES 2008: 10th International Workshop, Washington, DC, USA, August 10-13, 2008. Proceedings 10*. Springer, 2008, pp. 426–442.

[16] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems-CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings 6*. Springer, 2004, pp. 16–29.

[17] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*. Springer, 2003, pp. 13–28.

[18] T. S. Rastogi and E. B. Kavun, "Deep learning techniques for side-channel analysis on aes datasets collected from hardware and software platforms," in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 21st International Conference, SAMOS 2021, Virtual Event, July 4–8, 2021, Proceedings*. Springer, 2022, pp. 300–316.

[19] B. J. Gilbert Goodwill, J. Jaffe, P. Rohatgi *et al.*, "A testing methodology for side-channel resistance validation," in *NIST non-invasive attack testing workshop*, vol. 7, 2011, pp. 115–136.

[20] A. A. Selçuk, "On probability of success in linear and differential cryptanalysis," *Journal of Cryptology*, vol. 21, pp. 131–147, 2008.

[21] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[22] Z. He, T. Ao, M. Wan, K. Dai, and X. Zou, "Erist: An efficient randomized instruction insertion technique to counter side-channel attacks," *IAENG International Journal of Computer Science*, vol. 43, no. 1, 2016.

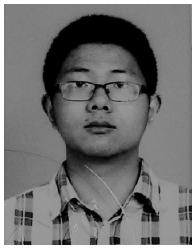
[23] Z. He, X. Deng, B. Yang, K. Dai, and X. Zou, "A sca-resistant processor architecture based on random delay insertion," in *2015 International Conference on Computing and Communications Technologies (ICCCT)*. IEEE, 2015, pp. 278–281.

[24] G. Leplus, O. Savry, and L. Bossuet, "Insertion of random delay with context-aware dummy instructions generator in a risc-v processor," in *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2022, pp. 81–84.

[25] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[26] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," *Volume I: User-Level ISA', version*, vol. 2, 2014.

drawbacks:
1) 1 clk cycle delay from between instruction entering the buffer and ida updating
2) No encryption
3) no means to switch off the ESM



Yao Zhang (1994-), is a PhD candidate at Wuhan University. His research interests include hardware security, cryptographic chip side channel security. E-mail: zy_max@foxmail.com

Ming Tang (1976-), Ph.D., Professor, Wuhan University. Her research interests include cryptography, secure design of cryptographic chips, lightweight protection against side-channel analysis, and systematic research on side-channel analysis. E-mail: m.tang@126.com