J Hardw Syst Secur (2017) 1:68–84
DOI 10.1007/s41635-017-0004-3

Trace Buffer Attack on the AES Cipher

Yuanwen Huang[1] · **Prabhat Mishra**[1]

Received: 23 August 2016 / Accepted: 27 March 2017 / Published online: 20 April 2017
© Springer 2017

Abstract Since the standardization of AES/Rijndael symmetric-key cipher by NIST in 2001, it gained widespread acceptance in various protocols and withstood intense scrutiny from the theoretical cryptanalysts. From the physical implementation point of view, however, AES remained vulnerable. Practical attacks on AES via fault injection, differential power analysis, scan-chain and cache-access timing have been demonstrated so far. In this paper, we propose a novel and effective attack, termed *Trace Buffer Attack*. Trace buffers are extensively used for post-silicon debug of integrated circuits. We identify the trace buffer as a source of information leakage. We first report the detailed process of trace buffer attack assuming that the register-transfer level (RTL) implementation is available. We further analyze the AES encryption algorithm and Rijndael's key expansion algorithm, and illustrate that trace buffer attack is feasible without implementation (RTL) knowledge. Our experimental results show that trace buffer attack is capable of partially recovering the secret keys of different AES implementations.

Keywords Post-silicon debug · Trace buffer · AES · Cryptography · Cryptanalysis

✉ Yuanwen Huang
yuanwenhuang@ufl.edu

[1] Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611, USA

1 Introduction

As the human civilization is collectively progressing towards an ubiquitous information age, the corresponding stakes on ensuring confidentiality, integrity and authenticity are also rising higher. Advanced Encryption Standard (AES) algorithm with various key lengths (128, 192 and 256) is widely used. The fact that AES stood the intense scrutiny from attackers over the last 15 years itself makes it an important benchmark for cryptography and cryptanalysis. So far, the best-known attempt against full AES-128, by *algebraic cryptanalysis*, has a computational complexity of $2^{126.1}$, which is slightly better than the brute-force attack and practically infeasible [11]. However, the perspective of *physical cryptanalysis* changes this scenario completely.

In practice, one routinely faces a situation where the cryptographic schemes are deployed in different adversarial setting, where keys are compromised, and the internal memory is not fully opaque. This situation leads to a set of physical cryptanalysis techniques, commonly known as *side channel attacks*. Side channel attacks exploit the physical implementation of cryptographic algorithms. The physical implementation might enable *leakage*, i.e., observations and measurements on the implementation details, as well as tampering with them. Such attacks have broken systems with mathematical security proof. In this scenario, secure implementation is rapidly becoming as important as the mathematical security proofs. For example, an AES implementation with protection against a first-order side-channel attack is presented in [24]. The protected design is still vulnerable to more sophisticated attacks and even then,

incurs 4.6× area- and 3.6× power-overhead, respectively, compared to the unprotected implementation.

In light of these developments, it is of utmost importance to remain fully aware of the design vulnerabilities, in the form of precise information leakage. In this paper, we introduce **Trace Buffer Attack** (TBA), a novel attack that can be mounted with the help of post-silicon debug facilities present in a chip. System-on-Chip (SoC) designs have in-built trace buffer (described in Section 2) that traces a small set of internal signals during execution, and the traced signal values are used during post-silicon (off-line) debug. There is an inherent conflict between security and observability. While debug engineers would like to have better observability, the security experts would like to enforce limited or no visibility with respect to the security modules in a SoC design. A trade-off is typically made where trace signals are carefully selected to maintain security while providing reasonable debug capability. To the best of our knowledge, the vulnerability of trace buffers in cryptographic implementation has not been studied in the literature. We conclusively show that to achieve a certain quantifiable level of debugging ability, security is compromised. We consider AES as the benchmark algorithm for demonstrating the efficacy of this attack though, the attack can be mounted on other ciphers following the same principles outlined in this work. Our experimental results demonstrate that we can fully recover the secret key for AES-128 (iterative) implementation whereas we can partially recover the secret key for various pipelined AES implementations.

The rest of this paper is organized as follows. Background on AES and trace buffer is covered in Section 2. Section 3 surveys related work on AES attack and trace buffer. Section 4 describes our trace buffer attack with knowledge of the RTL implementation. Section 5 analyzes the process to attack without any knowledge of the RTL implementation. Section 6 presents the experimental studies followed by proposed countermeasures in Section 7. The paper is concluded in Section 8.

2 Background

2.1 AES Specification

AES works on a block size of 128 bits and a key size of 128, 192 or 256 bits, which are referred to as AES-128, AES-192 and AES-256, respectively[1]. We briefly review AES-128 here, for further details readers can refer to [3].

The encryption flow of AES is shown in the Fig. 1. AES accepts a 128-bit plaintext, 128-bit user key and generates 128-bit ciphertext. The encryption proceeds through an initial round and subsequent 10 round repetition of 4 steps. These steps are *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. In the final round, MixColumns step is skipped. For each of these rounds, separate 128-bit round subkeys are needed. The round subkeys are generated from the initial user key via a key expansion step. The key expansion uses Rijndael's key schedule.

The plaintext is organized as a 4×4 column-major order matrix, which is operated through the AES rounds. The SubBytes step uses a non-linear transformation on every element of the matrix. The non-linear transformation is defined by an 8-bit substitution box, also known as Rijndael S-box. The ShiftRows step cyclically shifts the bytes in each row by a certain offset. In the MixColumns step, each column is multiplied by a fixed matrix. In the AddRoundKey step, each byte of the matrix is exclusive-OR-ed with each byte of the current round subkey. This is shown graphically in the Fig. 1.

2.2 Trace Buffer

One of the major challenges in post-silicon validation and debug is the limited controllability and observability of the fabricated integrated circuit. Trace buffer is widely used to improve the observability of circuit and thus assist post-silicon debug and analysis [16, 29–31]. It is a buffer that traces (records) some of the internal signals in a silicon chip during runtime. If an error is encountered, the content of trace buffer would be dumped out through JTAG interface for off-line debug and error analysis. Due to design overhead constraints, the number of trace signals is only a small fraction of all internal signals in the design. The size of the trace buffer directly affects the observability that we can get from the trace buffer.

Figure 2 illustrates how the trace buffer is used during post-silicon validation and debug. Signal selection is done during the design time (pre-silicon phase). Let us assume that S_1, S_2, ..., S_n are the selected trace signals. Figure 2 shows a trace buffer with a total size of $n \times m$ bits, which traces n signals (buffer width) for m cycles (buffer depth). For example, the ARM ETB [1] trace buffer provides buffer sizes ranging from 16Kb to 4Mb. In this case, a 16Kb buffer can trace 32 signals for 512 cycles (i.e., n=32 and m=512). Once the trace signals are selected, they need to be routed to the trace buffer. A trigger unit is also needed that decides when to start and stop recording the trace signals based on specific (error) events. The trace buffer records the states of the traced signals during runtime. During debug time, the states of traced signals will be dumped out through the standard JTAG interface. Signal restoration is performed to restore as many states as possible, which is to maximize the observability of the

[1] For the rest of the paper, unless explicitly specified, we will use AES-128 and AES interchangeably.

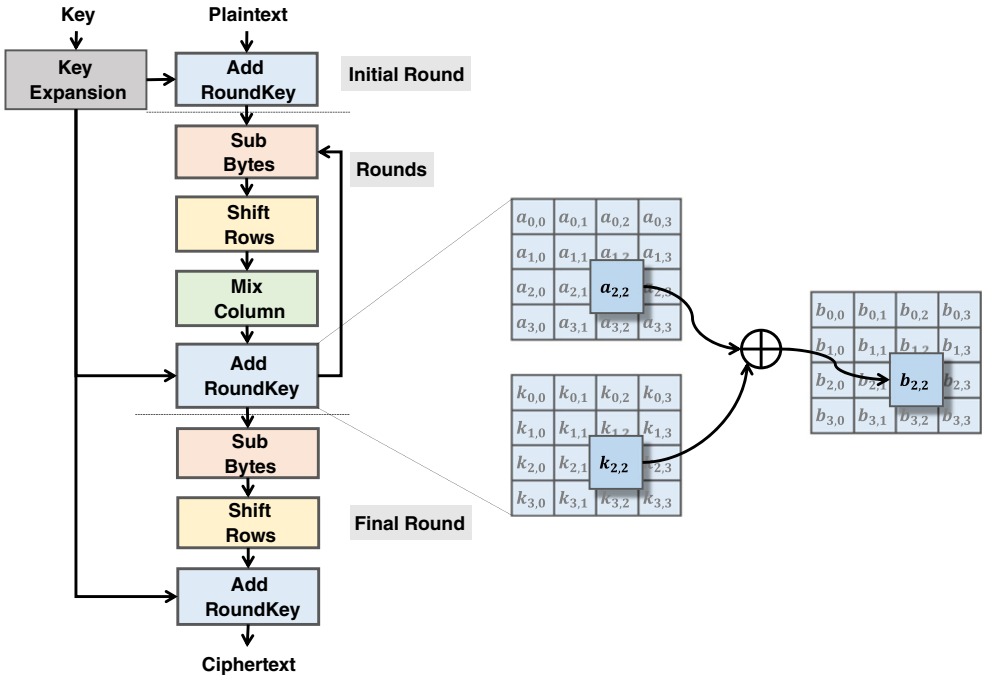


Fig. 1 AES encryption flow

internal signals in the chip. The off-line debug and analysis would be based on the traced signals and the restored signals.

3 Related Work and Motivation

In this section, we first describe the prior works related to AES attack. Next, we discuss the inherent conflict on debug observability versus security. Finally, we present our adversary model for trace buffer attack.

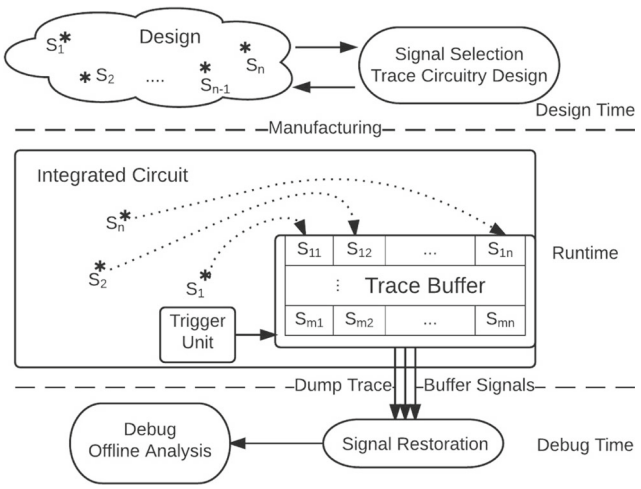


Fig. 2 Overview of trace buffer in system validation and debug

3.1 AES Attack

Since the pioneering works on differential power analysis [22], numerous side-channel attacks have been developed. Side-channel attacks can be classified into *passive*, *semi-invasive* and *invasive* attacks depending on the level of intrusion necessary for the attacker. The side-channels are of varied forms ranging from the software execution pattern such as cache timing [27] to more detailed hardware-oriented information leakages such as electromagnetic waves [17], acoustic waves [18] and optical fault injections [34]. Recent surveys on timing channels and invasive fault attacks are available in [13] and [8], respectively. Another approach of constructing an invasive attack originates from a malicious hardware, secretly inserted into a chip. These are commonly known as hardware Trojans [15, 20].

Considering the impact that AES has on our everyday communications, many of the attack techniques report their efficacy by demonstrating an attack on AES, which is also the target cipher for the current work. Among the hardware side-channel attacks reported against AES, attacks based on scan-chain [5] and external fault injections [25] are most prominent. For all these attacks, effective countermeasures are proposed and the inherent resilience of various design points [4] is studied. It is also shown that there exists an interplay between the countermeasures of one attack and the consequently increased vulnerability against another attack [32].

3.2 Debug Observability Versus Security

Trace buffer provides observability into the circuit so as to assist post-silicon debug and test. The quality of selected trace signals will directly affect the observability that we can get from the circuit. The goal of trace signal selection is to obtain a set of signals, which can restore the maximum number of internal states in the chip. Basu et al. [9] proposed a metric based algorithm that employs total restorability for selecting the most profitable signals. Chatterjee et al. [12] proposed a simulation based algorithm which is shown to be more promising than metric based approaches. Li and Davoodi [23] proposed a hybrid approach which combines the advantages of metric and simulation based approaches. A simulation based approach using augmentation and ILP techniques by Rahmani et al. [29] demonstrated very high restoration capability and thus high observability of the internal signals.

It is accepted in the research community that there is a strong link between observability/testability and security [19] for Design for Testability (DfT) facilities. Scan chain based DfT has been studied for attacks on block ciphers, including Data Encryption Standard (DES) [35] and Advanced Encryption Standard (AES) [36], and stream ciphers [26]. However, it is surprising that the vulnerability of trace buffers in cryptographic implementation is not studied so far. This forms the core motivation of our work. We show that an effective security attack is possible by analyzing the trace buffer content.

3.3 Attack Model

The proposed trace buffer attack has the following assumptions:

1) The primary key is stored in secure memory and properly maintained by key management.
2) The attacker knows the AES encryption algorithm as it is open to public.
3) High level timing information, as well as the RTL implementation of the AES, is known to the attacker.
4) The attacker has access to trigger the trace buffer recording at any time and dump out the traced content via the JTAG port after designated clock cycles.
5) The attacker does not know which signals are recorded in the trace buffer.

The assumptions that we have made for trace buffer attack are similar to the ones made in the literature for scan-chain attacks [26, 35, 36]. The primary key is assumed to be properly maintained by key management. The attacker knows the algorithmic details and the high level timing information of the cryptosystem being implemented in the device. In Assumption (3), we assume that the attacker knows the RTL implementation of AES for the attack proposed in Section 4. We realize that this is a very strong assumption. We remove this assumption to make it a more realistic attack in Section 5. Assumption (4) is similar to scan chain attacks concerning the debugging JTAG port. The attacker has the ability to run the device under test mode, i.e. trigger the recording of trace buffer and dump out the buffer content. He can feed the circuit with designated inputs (plaintexts and fake keys) for cryptanalysis. For Assumption (5), the attacker does not know which signals are recorded in the trace buffer. It is the first challenge to be resolved if the attacker wants to launch an attack.

Compared with the scan chain attacks, the trace buffer attack introduces the following additional challenges. (1) The first step for trace buffer attack is similar to scan-based attack, which is to identify signals. For scan-based attack, the attacker knows what signals are in the scan flip-flops. The attacker's problem is to identify the structure (order) of the scan chain. However, for trace buffer attack, the attacker does not even know which signals are selected to be recorded in the buffer. (2) The number of signals traced is usually much smaller compared to the length of scan chains (especially if it is full-scan). The number of traced signals is limited, which makes it more challenging for the second step of signal restoration for trace buffer attack. (3) The trace buffer can record values over a continuous interval. Trace buffer attack has the advantage to analyze the signal values between clock cycles (between encryption rounds, while the scan chain can only scan out the signal values at one clock cycle. (4) Protection against scan-based attacks mostly focuses on scrambling the structure of scan chain. For trace buffer attack, the countermeasures have to focus on scrambling or direct encryption of the recorded signals.

Preliminary version of this work appeared in conference proceeding [21] with the discussion of trace buffer attack when RTL implementation is available. This paper extends the attack technique, in particular by relaxing the assumption of RTL knowledge and analysing the AES algorithm for a more realistic attack.

4 Trace Buffer Attack with RTL Knowledge

In this section, we launch the trace buffer attack assuming that the register-transfer level (RTL) implementation is available. The proposed attack proceeds in two phases. In the first phase, we attempt to establish the correspondence between the signal values in trace buffer and variables in the AES design. In the second phase, depending on the trace buffer size and the number of cycles for which each signal is dumped, the signal values are fed to the restoration algorithm. The restoration algorithm attempts to restore internal signals and eventually recover bits in the

user-specified primary key. Details of each step are elaborated in the following sections.

4.1 Attack Step 1: Determine Trace Buffer Signals

If an attacker wants to steal the primary key, signal values in the trace buffer are the starting point of hacking. Unless the traced data is encrypted or debugging is authentication based, the attacker can easily dump traced data through JTAG interface. The challenge for *Trace Buffer Attack* is that the attacker does not know what signals are recorded in the trace buffer. In this section, we assume that the attacker has access to a few test chips and the RTL description of the AES design. The one-to-one mapping between the traced signals and the registers in RTL description can be established by running some test chips and matching with RTL simulation.

Algorithm 1 MAP SIGNALS TO REGISTERS IN RTL

```
Input: AES RTL implementation, AES test chip
Output: Identified signals in trace buffer
while true do
    Select a random plaintext T_itr, a random key K_itr
    Run RTL simulation with T_itr and K_itr for c cycles
    Run the test chip with T_itr and K_itr for c cycles
    for Each traced signal S_i in trace buffer do
        Represent S_i as a vector of c values
        for Each register R_j in RTL do
            Represent R_j as a vector of c values
            if the vectors of S_i and R_j are the same then
                (S_i, R_j) is a possible match
            end
        end
        if S_i has a unique match R_j then
            (S_i, R_j) is a verified match
        end
    end
    if Every signal in S has a unique match then
        Break
    end
end
return Identified signals in trace buffer
```

Algorithm 1 shows the process to match signals in trace buffer with registers in the RTL implementation. For each iteration, we select a random input plaintext T_{itr} and a random key K_{itr}. We run the test chip and the RTL simulation with the same key and input text for c cycles. Each traced signal will have a vector of c values stored in the trace buffer. For each traced signal, we compare its vector with vectors of all the registers from RTL simulation. If a unique match is found in the RTL simulation, this traced signal is identified in the RTL description. We repeat the process until all the traced signals are uniquely identified.

4.2 Attack Step 2: Signal Restoration

Let us assume that the attacker has finished the preparation in the previous step and successfully identified the signals in the trace buffer. The next step is to run the chip in the working mode with the secret primary key and take advantage of the trace buffer to initialize the attack. The attacker dumps out the signal states recorded in the buffer during online encryption, and tries to analyze the design so as to recover as many other signals as possible, and eventually obtain the primary key. In post-silicon debug, restoration of unknown signals based on trace buffer data is a crucial step in debugging. This section describes signal restoration based on trace buffer.

The signals can be reconstructed from the traced signals in two directions: forward and backward restoration. Forward restoration pushes the restoration of signals from input to output, which is the process of inferring output values if some inputs are known. Backward restoration infers input values if some outputs are known. Figure 3 illustrates forward and backward restoration with a simple example of AND gate. Figure 3a shows forward restoration: if one of the inputs is 0, the output can be inferred to be 0; if both of the inputs are 1, the output can be inferred to be 1. Figure 3b and c shows backward restoration: if the output is 1, both of the inputs can be inferred to be 1. However, if the output is 0, backward restoration might not be successful as shown in (c). The restoration process for other logic components is similar to AND gate. The restoration for registers (flip-flops) is that the state at current cycle is related to the state at previous cycle as specified by their truth tables.

Algorithm 2 outlines the major steps in a typical restoration algorithm. We first read in the AES circuit and form

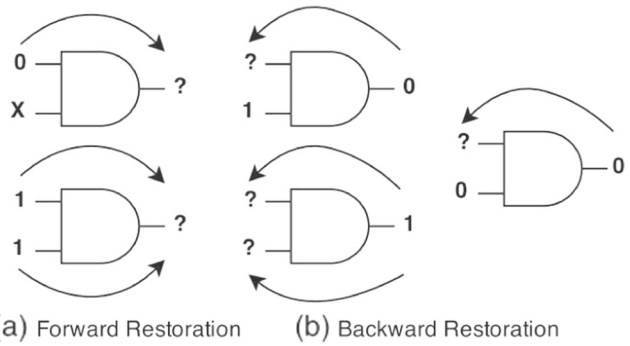


Fig. 3 Illustration of signal restoration for an AND gate

a hypergraph. Based on the trace buffer content, we perform forward and backward restorations to construct value assignments for un-traced nodes. We use a queue *UnderProcess* to keep track of nodes which have new values been restored. The queue is initialized with nodes from the trace buffer. Each node in the queue is processed by backward and forward restoration and nodes with newly assigned values will be put to the end of the queue. This process continues until no new assignments are created, i.e., the queue becomes empty. Although this algorithm has exponential complexity, in reality, it completes the process very fast (as demonstrated in Section 6) since the number of new values created decreases significantly after each iteration.

Algorithm 2 SIGNAL RESTORATION ALGORITHM

Input: Trace buffer content, AES netlist
Output: Restored signal (node) values
Read in the AES circuit and form a hypergraph
Put all traced nodes into the $UnderProcess$ queue
Update the traced nodes with their known values (0/1)
Update all other nodes with unknown values (x)
while $UnderProcess$ *is not empty* **do**
  Take a node N from the $UnderProcess$ queue
  for *each node in* N*'s* $BackwardNeighbors$ **do**
    Backward Restoration for this neighbor node
    if *value at any cycle is restored* **then**
      Add this neighbor to $UnderProcess$
    end
  end
  for *each node in* N*'s* $ForwardNeighbors$ **do**
    Forward Restoration for this neighor node
    if *value at any cycle is restored* **then**
      Add this neighbor to $UnderProcess$
    end
  end
end
return

5 Trace Buffer Attack Without RTL Knowledge

In this section, we aim to attack the AES cipher without the knowledge of its RTL implementation. We consider iterative AES-128 with a trace buffer (32×512) of width 32 and depth 512. We first identify as many signals as we can in the trace buffer, by referring to the variables in the AES encryption algorithm. We then show that the primary key can be retrieved by taking advantage of Rijndael's key scheduling,

5.1 Mapping Signals to Algorithm Variables

Suppose we have the AES-128 chip as in Case Study 1 (Section 6), we don't have the implementation at RTL level, which means that we cannot run the RTL simulation and the chip side-by-side to compare which signals are recorded in the trace buffer. What we know from the chip datasheet is that it takes 13 cycles to complete one encryption. The trace buffer contains values of 32 internal signals. The intermediate encrypted text and the round key are most beneficial for signal restoration to recover the primary key bits. The code snippet of C implementation of AES-128 is shown in Fig. 4. The variables $\{state[0], state[1], state[2], state[3]\}$ represent the intermediate encrypted text, and the variables $[w0\ w1\ w2\ w3]$ represent the 128-bit round key. The *for* loop of 10 iterations represents the 10 encryption rounds.

We would like to find out whether any signals (bits) in the trace buffer are from the intermediate encrypted text or the round key. It takes 13 cycles for the AES chip to complete one encryption operation, which is one initial round and 10 subsequent rounds in the C program of AES. If one bit is from the round key $[w0\ w1\ w2\ w3]$, we can represent the values of this bit over 10 rounds as a 10-bit binary string by running the C program. The resulting 10-bit string would be a substring of the same bit/signal in the trace buffer. If we apply a fixed key and a fixed plaintext when we run the test chip, a matching algorithm variable bit from the C program would actually be a 10-bit substring repeating with a period of 13 in the string of the same signal in the trace buffer.

Algorithm 3 shows the details about how we matched the signals from trace buffer bits to algorithm variable bits. We run the C program and the test chip with a same random

```
void AES128(word state[], word key[]){
    /* the initial round */
    state[0] ^= key[0];
    state[1] ^= key[1];
    state[2] ^= key[2];
    state[3] ^= key[3];
    word y, p0, p1, p2, p3;
    byte rcon = 1;
    /* ten encryption rounds */
    word w0 = key[0];
    word w1 = key[1];
    word w2 = key[2];
    word w3 = key[3];
    for(int i=1; i<=10; i++) {
        // round-key generation
        ...
        // four-step encryption
        ...
    }
}
```

Fig. 4 C Code Snippet for AES-128

plaintext T_{itr}, and a same random key K_{itr}. Each signal S_i in trace buffer is represented as a 512-bit binary string and each variable bit $V_{j,k}$ as a 10-bit binary string. We decide that (S_i, $V_{j,k}$) is a possible match if variable bit $V_{j,k}$ is a repeating pattern of S_i. The algorithm tries to identify as many signals of S as possible, and it will terminate when matched signals are uniquely identified and no more unique match can be found. The complexity of the matching algorithm is $O(W * \sum_j B_j)$, where W is the buffer width, B_j is the number of bits in variable V_j, $\sum_j B_j$ is the total number of candidate variable bits.

Algorithm 3 MAP SIGNALS TO BITS IN AES VARIABLES

```
Input: AES C implementaion, AES test chip
Output: Identified signals in trace buffer
while true do
    Select a random plaintext T_itr, a random key K_itr
    Run the C program of AES-128 with T_itr and K_itr
    Run the test chip with T_itr and K_itr
    for Each traced signal S_i in trace buffer do
        Represent S_i as a 512-bit binary string
        for Each variable V_j in AES algorithm do
            Extract V_j across the 10 encryption rounds
            for Each bit V_(j,k) in V_j do
                Represent V_(j,k) as 10-bit binary string
                if V_(j,k) is a repeating pattern in S_i then
                    (S_i, V_(j,k)) is a possible match
                end
            end
        end
        if S_i has a unique match V_(j,k) then
            (S_i, V_(j,k)) is a verified match
        end
    end
    if Every signal in S have either unique or no match then
        Break
    end
end
return Identified signals in trace buffer
```

By applying the above method, we can identify 30 out 32 signals in the trace buffer. These 30 signals include two bits from the intermediate register, and another 28 bits from the round key register can be matched. The 28 bits from the 128-bit round key register include 1 bit from the first word, 2 bits from the third word, and 25 bits from the fourth word. More details about these signals are presented in Section 6.1. Note that we apply the state-of-the art signal selection algorithm to select the trace signals. In other words, we did not choose signals that would help us in trace buffer attack. This research also points to the need for having security-aware signal selection.

5.2 Attack by Taking Advantage of Rijndael's Key Expansion

As shown in Fig. 1, the last step of each round is XOR with a round key. The initial round takes the primary key, and each of the following 10 rounds uses a different round key. The round key generation follows the Rijndael's key expansion algorithm to generate the next 4-word round key $[RK_{i+1,1}, RK_{i+1,2}, RK_{i+1,3}, RK_{i+1,4}]$ based on the current 4-word round key $[RK_{i,1}, RK_{i,2}, RK_{i,3}, RK_{i,4}]$.

$$\begin{aligned} RK_{(i+1,1)} &= RK_{(i,1)} \oplus sbox(lcs(RK_{(i,4)})) \oplus RC_i \\ RK_{(i+1,2)} &= RK_{(i,2)} \oplus RK_{(i+1,1)} \\ RK_{(i+1,3)} &= RK_{(i,3)} \oplus RK_{(i+1,2)} \\ RK_{(i+1,4)} &= RK_{(i,4)} \oplus RK_{(i+1,3)} \end{aligned} \tag{1}$$

Equation 1 shows the Rijndael's key expansion algorithm. For the $(i+1)^{th}$ round, the first word $RK_{(i+1,1)}$ is the XOR of three items: the first word of i^{th} round, the substituted word by applying a one-byte *lcs* (*Left Circular Shift*) operation and a byte-wise *sbox* substitution on the fourth word of i^{th} round, and the round constant RC_i. The *sbox* function is byte-to-byte substitution according to a 16×16 lookup table as shown in Fig. 5. For the other three words $RK_{(i+1,2)}$, $RK_{(i+1,3)}$, $RK_{(i+1,4)}$, they follow the same pattern: the XOR of the word itself at i^{th} round and the previous word at $(i+1)^{th}$ round. Based on the above observation, we generalize two rules as shown in Eq. 2, which will be useful for signal values restoration between cycles (rounds).

$$\begin{aligned} &\text{Rule 1: } sbox(lcs(RK_{(i,4)})) = RK_{(i,1)} \oplus RK_{(i+1,1)} \oplus RC_i \\ &\text{Rule 2: } RK_{(i+1,j-1)} = RK_{(i,j)} \oplus RK_{(i+1,j)},\ j = 2, 3, 4 \end{aligned} \tag{2}$$

In Rijndael's round key expansion, the fourth word of current round key is the seed word for generating the next round key, which is shown in Eqs. 1 and 2. The *lcs* and *sbox* operations on the fourth word are the sources to introduce unpredictable randomness to round keys. We have figured out that the trace buffer contains bits from the fourth word

in the round key register in Section 5.1, which would be critical for us to retrieve the full key.

(1) Analysis: Assume the Fourth Word Known

Table 1 shows that if all the 32 bits of the fourth word of the round keys are known, the 128-bit primary key, which is all 0's for this example, can be recovered. Round keys are represented as hexadecimal digits and 'X' means 'unknown'. Assume we know the fourth word of the round keys as shown in Table 1(A). We first apply Rule 2 on $RK_{(1\sim10,4)}$ (the fourth column of the Table 1(B)), we can retrieve the third word of all round keys except the first round, which is $RK_{(2\sim10,3)}$ (the third column). Similarly, we can retrieve the second column and the first column ($RK_{(3\sim10,2)}$ and $RK_{(4\sim10,1)}$). Now we have successfully retrieved the full round key RK_4. The Rijndael's key expansion defines the relation between two consecutive round keys, which means we can use Eq. 1 to get the previous round key if we have the current round key. With RK_4 already retrieved, we can then get RK_3, RK_2, RK_1 and eventually RK_0, which is the primary key.

Table 1 shows that we would be able to retrieve RK_4 if all the 32 bits of the fourth word of the round key register are known. In fact, only the first four rows in Table 1 are needed to retrieve RK_4. With Rule 2, any four consecutive rounds with the fourth word known will be able to retrieve a full round key, i.e. the value of $RK_{(i\sim i+3),4}$ will lead to the recovery of full round key RK_{i+3}.

Table 1 Assume we have the fourth word of all rounds known, we can apply Rule 2 in a cascaded way and recover all bits in RK_4. From RK_4, we can use Eq. 1 to get RK_3, RK_2, RK_1, and RK_0, which is the primary key

(A) Assume the fourth word of all rounds known				
RK_1	XXXXXXXX	XXXXXXXX	XXXXXXXX	62636363
RK_2	XXXXXXXX	XXXXXXXX	XXXXXXXX	F9FBFBAA
RK_3	XXXXXXXX	XXXXXXXX	XXXXXXXX	0B0FAC99
RK_4	XXXXXXXX	XXXXXXXX	XXXXXXXX	7E91EE2B
RK_5	XXXXXXXX	XXXXXXXX	XXXXXXXX	F34B9290
RK_6	XXXXXXXX	XXXXXXXX	XXXXXXXX	6AB49BA7
RK_7	XXXXXXXX	XXXXXXXX	XXXXXXXX	C61BF09B
RK_8	XXXXXXXX	XXXXXXXX	XXXXXXXX	511DFA9F
RK_9	XXXXXXXX	XXXXXXXX	XXXXXXXX	4C664941
RK_{10}	XXXXXXXX	XXXXXXXX	XXXXXXXX	6F8F188E
(B) Apply Rule 2 to recover RK_4				
RK_1	XXXXXXXX	XXXXXXXX	XXXXXXXX	62636363
RK_2	XXXXXXXX	XXXXXXXX	9B9898C9	F9FBFBAA
RK_3	XXXXXXXX	696CCFFA	F2F45733	0B0FAC99
RK_4	EE06DA7B	876A1581	759E42B2	7E91EE2B
RK_5	7F2E2B88	F8443E09	8DDA7CBB	F34B9290
RK_6	EC614B85	1425758C	99FF0937	6AB49BA7
RK_7	21751787	3550620B	ACAF6B3C	C61BF09B
RK_8	0EF90333	3BA96138	97060A04	511DFA9F
RK_9	B1D4D8E2	8A7DB9DA	1D7BB3DE	4C664941
RK_{10}	B4EF5BCB	3E92E211	23E951CF	6F8F188E
(C) Use Eq. 1 to get $RK_3 \sim RK_1$, and RK_0 (the primary key)				
RK_0	00000000	00000000	00000000	00000000
RK_1	62636363	62636363	62636363	62636363
RK_2	9B9898C9	F9FBFBAA	9B9898C9	F9FBFBAA
RK_3	90973450	696CCFFA	F2F45733	0B0FAC99
RK_4	EE06DA7B	876A1581	759E42B2	7E91EE2B

(2) Restoration from Partial Information in Trace Buffer

However, the trace buffer contains only 25 bits of the fourth word as shown in Section 6.1. The above approach needs four consecutive rounds with the fourth word known, while we have 7 bits missing for the fourth word of each round. If we try to brute-force all possibilities, the time complexity is $2^{7*4} = 2^{28}$. While this brute-force attack is within reasonable computation limit, we show that even that is not required if we put the *sbox* bijection property into use. The *sbox* lookup table as shown in Fig. 5 is the core of Rijndael's key expansion. It is a bijective mapping between the bytes before and after *sbox* substitution. The bijection property of this byte-to-byte mapping makes it possible to recover missing bits in round keys.

Rule 1 shows the relationship between $RK_{(i,1)}$, $RK_{(i+1,1)}$, $RK_{(i,4)}$ and RC_i. Given that $RK_{(i,1)}$, $RK_{(i+1,1)}$ and $RK_{(i,4)}$ are partially obtained from the trace buffer content and RC_i is a known constant. We use the example in Fig. 6 to illustrate how Rule 1 can help recover missing bits in $RK_{(i,4)}$. In this example, $RK_{(4,1)}$ and $RK_{(5,1)}$ have 4 bits missing and $RK_{(4,4)}$ has 5 bits missing. We apply Rule 1 as shown in line 7 and derive with the partially known $RK_{(4,1)}$, $RK_{(5,1)}$ and $RK_{(4,4)}$. After some bit-manipulation,

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Fig. 5 AES *sbox* lookup table (the numbers are in hexadecimal format)

1: Given partial round key bits:
2: $RK_{(4,1)} = (111x11100000x1101101101001xx1011)_b$
3: $RK_{(5,1)} = (011x11110010x1100010101110xx1000)_b$
4: $RK_{(4,4)} = (01xx11101001x0x11x10111000xx1011)_b$
5: $RC_4 = (00010000000000000000000000000000)_b$

6: Apply Rule 1:
7: $\mathbf{sbox}(\mathbf{lcs}(RK_{(4,4)})) = RK_{(4,1)} \oplus RK_{(5,1)} \oplus RC_4$
8: $\longrightarrow$
9: $\mathbf{sbox}(\mathbf{lcs}(01xx1110\,1001x0x1\,1x101110\,00xx1011))$
10: $= 100x00010010x0001111000111xx0011$
11: $\longrightarrow$
12: $\mathbf{sbox}(1001x0x1\ 1x101110\ 00xx1011\ 01xx1110)$
13: $= 100x0001\ 0010x000\ 11110001\ 11xx0011$

14: For each byte, check the *sbox* lookup table
15: $sbox([1001x0x1]) = [100x0001]$
16: $\rightarrow sbox([10010001] = [10000001]$
17: $sbox([1x101110]) = [0010x000]$
18: $\rightarrow sbox([11101110] = [00101000]$
19: $sbox([00xx1011]) = [11110001]$
20: $\rightarrow sbox([00101011] = [11110001]$
21: $sbox([01xx1110]) = [11xx0011]$
22: $\rightarrow sbox([01111110] = [11110011]$

23: Missing bits of $RK_{(4,4)}$ recovered by *sbox* lookup table:
24: $RK_{(4,4)} = (01111110100100011110111000101011)_b$

Fig. 6 An example showing the recovery of missing bits in $RK_{(4,4)}$ by using *sbox* lookup table (Rule 1)

we get *sbox* mapping from a 32-bit word to another 32-bit word in line 12∼13. In line 15, the first byte is $1001x0x1$ and we would like to figure out two unknown bits. If we decompose the byte in half, the left part is 9 and the right part could be four choices: 1, 3, 9 or B. This means we need to consider sbox(9,1), sbox(9,3), sbox(9,9) or sbox (9,B) as potential matches. However, the right hand side of line 15 indicates that the expected value $100x0001$ can be either 81 or 91. Among the four possible choices, only sbox(9,1) fits the requirement. Therefore we get the unique mapping for the first byte as sbox(9,1) = 81, i.e., $sbox([10010001]) = [10000001]$ as shown by the yellow circle (Row 9 and Column 1) in the lookup table. Using similar lookup, we can identify the other three bytes, i.e., $sbox([11101110]) = [00101000]$, $sbox([00101011]) = [11110001]$, and $sbox([01111110]) = [11110011]$.

Algorithm 4 shows the steps to restore the primary key from the available trace buffer content. In Step 1, we first apply Rule 2 in a cascaded way to get partial bits of the first word in round keys. In Step 2, we then apply Rule 1 and the unique mapping property of *sbox* to further recover missing bits of the fourth word in round keys. After using *sbox*, the following steps will be very similar to the example shown in Table 1. In Step 3, we re-apply Rule 2 in a cascaded way to get a full round key. Finally in Step 4, we use Equation 1 to push back from that round key and eventually get the primary key. Section 6-A shows detailed experimental results after each step.

The most critical part in Algorithm 4 is Step 2, i.e., applying Rule 1 to recover missing bits in $RK_{(i,4)}$. Note, it is also possible that multiple candidate mappings are available in the lookup table for the partially known bytes. In that case, we have to evaluate all possible mappings. Our experiments with different random numbers suggest that the chances of multiple candidates are very rare.

Algorithm 4 RESTORE MISSING BITS IN ROUND KEYS

Input: Identifid signals in round keys from trace buffer
Output: Restored round key bits
Update identified bits with values (1/0) from trace buffer
Update all other bits with unknown values (x)
/***Step 1**: *Apply Rule 2* */
for $j \leftarrow 4$ ***to*** 2 **do**
  for $i \leftarrow 1$ ***to*** 9 **do**
    $RK_{(i+1,j-1)} = RK_{(i,j)} \oplus RK_{(i+1,j)}$
  end
end

/* **Step 2**: *Apply Rule 1* */
for $i \leftarrow 4$ ***to*** 9 **do**
  Use the bijection property of *sbox* to recover missing bits in $RK_{(i,4)}$
end

/* **Step 3**: *Apply Rule 2 one more time* */
for $j \leftarrow 4$ ***to*** 2 **do**
  for $i \leftarrow 1$ ***to*** 9 **do**
    $RK_{(i+1,j-1)} = RK_{(i,j)} \oplus RK_{(i+1,j)}$
  end
end

/***Step 4**: *Use Eq. 1 to get the primary key* */
for $i \leftarrow 9$ ***to*** 1 **do**
  $RK_{(i-1,2)} = RK_{(i,2)} \oplus RK_{(i,1)}$
  $RK_{(i-1,3)} = RK_{(i,3)} \oplus RK_{(i,2)}$
  $RK_{(i-1,4)} = RK_{(i,4)} \oplus RK_{(i,3)}$
  $RK_{(i-1,1)} =$
  $RK_{(i,1)} \oplus sbox(lcs(RK_{(i-1,4)})) \oplus RC_{i-1}$
end
return $PrimaryKey = RK_0$

6 Experimental Results

We applied our trace buffer attack on the AES Verilog implementations (the iterative AES-128, and the pipelined AES-128, AES-192 and AES-256 [2]) from the OpenCores website. The Synopsys Design Compiler is used to synthesize the RTL implementation into a gate-level netlist. We developed C++ code to simulate the gate-level circuits and implement the signal restoration algorithms. The experiments were conducted on a computer with AMD Opteron 2.4GHz core and 32GB memory. We used signal selection algorithm in [29] to select trace signals for the trace buffers since it produces signals that can maximize observability compared to the other signal selection techniques.

6.1 Case Study 1: Iterative AES-128

The iterative AES-128 design has 530 flip-flops and about 25,000 basic logic gates. The 530 flip-flops (registers) include:

- *ld_r*, *done*, which are one-bit control signals.
- *dcnt[0..3]*, which is a 4-bit register keeping track of the encryption rounds.
- *text_in_r[0..127]*, which is a 128-bit register holding the plaintext.
- *w0[0..31]*, *w1[0..31]*, *w2[0..31]*, and *w3[0..31]*, which are 32-bit each, holding the round keys.
- *sa00[0..7]*, *sa01[0..7]*, *sa02[0..7]*, *sa03[0..7]*, *sa10[0..7]*, *sa11[0..7]*, *sa12[0..7]*, *sa13[0..7]*, *sa20[0..7]*, *sa21[0..7]*, *sa22[0..7]*, *sa23[0..7]*, *sa30[0..7]*, *sa31[0..7]*, *sa32[0..7]*, *sa33[0..7]*, which are 8-bit registers holding intermediate encrypted text in bytes.
- *u0.rcon[24..31]* and *u0.r0.rcnt[0..3]*, which are 12 temporary registers in the key expansion unit.
- *text_out[0..127]*, which is a 128-bit register holding the ciphertext.

(1) Attack Without RTL Implementation

As described in Section 5, we assume that we don't have the RTL implementation. We first use Algorithm 3 to guess which bits of the round keys are recorded in the (32×512) trace buffer. We are able to recognize 28 bits from the round key, including 1 bit from the first word (w0[14]), 2 bits from the third word (w2[17] and w2[29]), and 25 bits from the fourth word (w3[0-3, 6-13, 15-16, 18, 20-27, 30-31]). We conduct the restoration process according to Algorithm 4. Table 2 shows intermediate results after each step of the restoration process. First, we apply Rule 2 (the relation between different words) to restore missing bits in the fourth word, which results in Table 2(B). We then apply Rule 1 (the unique mapping property of *sbox* lookup table) to get Table 2(C). We apply Rule 2 again to get a full round key RK_7, which results in Table 2(D). From RK_7, we can use Equation 1 to get $RK_6 \sim RK_1$ and eventually get the primary key RK_0, which is all 0's in this case.

The attack can successfully retrieve the full key without RTL knowledge. There are two major reasons why this attack works so well. Firstly, 28 bits of the round key registers are recorded in the trace buffer. The signal selection algorithm in [29] only greedily choose signals that are best for observability. The signals in the round key registers happen to be of highest restoration capability for observing other internal signals. The blindness of selection of these round key signals contributes to information leakage, as well as high observability. Secondly, the bijection property of *sbox* function plays a critical role in recovering the missing bits in the fourth word of the round keys. However, if too many bits from the round key were not recorded in the buffer, we might need a lot more brute-force effort in Step 2 of Algorithm 4 to verify missing bits when using the *sbox* lookup table. In the next section, we will see that the attack with RTL knowledge is more powerful in recovering primary key bits.

(2) Attack with RTL Implementation

We explore different trace buffer sizes with buffer widths of 8, 16, and 32, buffer depth (traced cycles) of 64, 128, 256 and 512 in our experiments. The signals recorded in the trace buffer are identified by using methods detailed in Section 4.1 with the help of RTL implementation. The identified signals for each buffer width is as follows:

- BufferWidth=8: {dcnt[2], ld_r, w3[2], w3[1], w3[30], w3[27], w3[17], w3[13]}
- BufferWidth=16: {dcnt[2], ld_r, w3[4], w3[29], w3[27], w3[23], w3[22], w3[18], w3[16], w3[15], w3[14], w3[13], w3[12], w3[10], w1[9], w3[8]}
- BufferWidth=32: {dcnt[2], ld_r, sa03[7], sa13[7], w3[7], w3[6], w3[3], w3[2], w3[1], w3[31], w3[30], w2[29], w3[27], w3[26], w3[25], w3[24], w3[23], w3[22], w3[21], w3[20], w3[18], w2[17], w3[16], w3[15], w0[14], w3[13], w3[12], w3[11], w3[10], w3[9], w3[8], w3[0]}

Table 3 shows our results of trace buffer attack on the iterative AES-128 cipher. The trace buffers with a buffer width of 32 and a buffer depth no less than 128 are able to recover the full primary key in a few minutes.

Figure 7a shows the number of bits in the user key leaked with different buffer sizes. Figure 7b shows the total number of internal states restored (debug observability) during restoration. The number of restored primary key bits increases with bigger buffer width. For the same buffer width, the number of restored key bits increases slightly as the trace cycles increase, and it will be saturated after buffer

Table 2 The above shows the steps to restore the primary key from the available trace buffer content. There are 28 bits available in the trace buffer, as shown in Table 2(A). We first apply Rule 2 to get Table 2(B). We then apply Rule 1 to get Table 2(C), and apply Rule 2 again to get a full round key RK_7 shown in Table 2(D). From RK_7, we can use Equation 1 to get RK_6, ..., RK_1 and eventually get the primary key (RK_0), which is all 0's in this case

(A) Step 0: Bits from the trace buffer

RK_1	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxx0xxxxxxxxxxxxxxxx	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxx0xxxxxxxxxxxxxxxx	00xx10110000x1x11x10110010xx1001
RK_4	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx11101001x0x11x10111000xx1011
RK_5	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxx1xxxxxxxxxxxxxxxx	11xx00110100x0x11x01001010xx0000
RK_6	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx10101011x1x01x01101110xx0111
RK_7	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxx1xxxxxxxxxxxxxxxx	11xx01100001x0x11x11000010xx1011
RK_8	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx00010001x1x11x11101010xx1111
RK_9	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx11000110x1x00x00100101xx0001
RK_{10}	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxx0xxxxxxxxxxxxxxxx	01xx11111000x1x10x01100010xx1110

(B) Step 1: Apply Rule 2 in a cascaded way

RK_1	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	100x10111001x0001x01100011xx1001	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	011x10010110x1001x00111111xx1010	111x00101111x1000x01011100xx0011	00xx10110000x1x11x10110010xx1001
RK_4	111x11100000x1101101101001xx1011	100x01110110x0100x01010110xx0001	011x01011001x1100x00001010xx0010	01xx11101001x0x11x10111000xx1011
RK_5	011x11110010x1100010101110xx1000	111x10000100x1000x11111000xx1001	100x11011101x0100x11110010xx1011	11xx00110100x0x11x01001010xx0000
RK_6	111x11000110x0010100101110xx0101	000x01000010x1010x11010110xx1100	100x10011111x1110x00100100xx0111	01xx10101011x1x01x01101110xx0111
RK_7	001x00010111x1010001011110xx0111	001x01010101x0000x10001000xx1011	101x11001010x1110x10101100xx1100	11xx01100001x0x11x11000010xx1011
RK_8	000x11101111x0010000001100xx0011	001x10111010x0010x10000100xx1000	100x01110000x1100x00101000xx0100	01xx00010001x1x11x11101010xx1111
RK_9	101x00011101x1001101100011xx0010	100x10100111x1011x11100111xx1010	000x11010111x0111x11001111xx1110	01xx11000110x1x00x00100101xx0001
RK_{10}	101x01001110x1110101101111xx1011	001x11101001x0101x10001000xx0001	001x00111110x0010x01000111xx1111	01xx11111000x1x10x01100010xx1110

(C) Step 2: Apply Rule 1 and sbox to recover

RK_1	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	100x10111001x0001x01100011xx1001	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	011x10010110x1001x00111111xx1010	111x00101111x1000x01011100xx0011	00xx10110000x1x11x10110010xx1001
RK_4	111x11100000x1101101101001xx1011	100x01110110x0100x01010110xx0001	011x01011001x1100x00001010xx0010	01111110100100011110111000101011
RK_5	011x11110010x1100010101110xx1000	111x10000100x1000x11111000xx1001	100x11011101x0100x11110010xx1011	11110011010010111001001010010000
RK_6	111x11000110x0010100101110xx0101	000x01000010x1010x11010110xx1100	100x10011111x1110x00100100xx0111	01101010101101001001101110100111
RK_7	001x00010111x1010001011110xx0111	001x01010101x0000x10001000xx1011	101x11001010x1110x10101100xx1100	11000110000110111111000010011011
RK_8	000x11101111x0010000001100xx0011	001x10111010x0010x10000100xx1000	100x01110000x1100x00101000xx0100	01010001000111011111101010011111
RK_9	101x00011101x1001101100011xx0010	100x10100111x1011x11100111xx1010	000x11010111x0111x11001111xx1110	01001100011001100100100101000001
RK_{10}	101x01001110x1110101101111xx1011	001x11101001x0101x10001000xx0001	001x00111110x0010x01000111xx1111	01xx11111000x1x10x01100010xx1110

Table 2 (continued)

(D) Step 3: Apply Rule 2 again to recover				
RK_1	xxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxx1xxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	100x10111001x0001x01100011xx1001	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxx0xxxxxxxxxxxxx	011x10010110x1001x00111111xx1010	111x00101111x1000x01011100xx0011	00xx10110000x1x11x10110010xx1001
RK_4	111x11100000x1101101101001xx1011	100x01110110x0100x01010110xx0001	011x01011001x1100x00001010xx0010	01111110100100011110111000101011
RK_5	011x11110010x1100010101110xx1000	111x10000100x1000x11111000xx1001	100011011101101001111100101111011	11110011010010111001001010010000
RK_6	111x11000110x0010100101110xx0101	000101000010010101110101100011100	100110011111111100001001001101111	01101010101101001001101110100111
RK$_7$	001000010111010100010111100000111	001101010101000001100010000001011	101011001010111101101011001111100	11000110000110111111000010011011
RK_8	000011101111100100000001100110011	001110111010100101110000100111000	100101110000011000001010000000100	01010001000111011111101010011111
RK_9	101100011101010011011000011100010	100010100111110110111001110111010	000111010111101110110011111011110	01001100011001100100100101000001
RK_{10}	101x01001110x1110101101111xx1011	001x11101001x0101x10001000xx0001	001x00111110x0010x01000111xx1111	01xx11111000x1x10x01100010xx1110
(E) Step 4: Apply Eq. 1 to push back to get RK_0 (which is the primary key)				
RK_0	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
RK_1	01100010011000110110001101100011	01100010011000110110001101100011	01100010011000110110001101100011	01100010011000110110001101100011
RK_2	10011011100110001001100011001001	11111001111110111111101110101010	10011011100110001001100011001001	11111001111110111111101110101010
RK_3	10010000100101110011010001010000	01101001011011001100111111111010	11110010111101000101011100110011	00001011000011111010110010011001
RK_4	11101110000001101101101001111011	10000111011010100001010110000001	01110101100111100100001010110010	01111110100100011110111000101011
RK_5	01111111001011100010101110001000	11111000010001000011111000001001	10001101110110100111110010111011	11110011010010111001001010010000
RK_6	11101100011000010100101110000101	00010100001001010111010110001100	10011001111111110000100100110111	01101010101101001001101110100111

Table 3 Iterative AES-128: number of bits in the key recovered and memory/time requirements for signal restoration

BufferWidth \ BufferDepth		64	128	256	512
8	Leaked key (bits)	6	6	6	6
	Memory (MB)	116.4	161.4	252.0	432.0
	Time (mm:ss)	0:27.75	0:56.07	1:50.35	3:43.26
16	Leaked key (bits)	18	25	28	28
	Memory (MB)	116.4	161.4	252.0	432.0
	Time (mm:ss)	0:27.82	0:55.94	1:51.00	3:44.10
32	Leaked key (bits)	98	128	128	128
	Memory (MB)	116.4	161.4	252.0	432.0
	Time (mm:ss)	0:28.01	0:55.98	1:52.81	3:51.38

depth is big enough (256 cycles or more). The 8×512, 16×512 and 32×512 trace buffer can respectively restore 6, 28 and 128 bits of the primary key. The fact that the 32×512 trace buffer can restore all 128-bit primary key is not surprising. The success of recovering the full primary key is due to the observability provided by the trace buffer. The iterative AES-128 design[2] has relatively short pathways with only 530 flip-flops in total. The 32 signals selected out of the 530 flip-flops is the set of signals which could offer best observability to the debugger.

The attack with RTL implementation is more powerful than without RTL in two ways. First, the attack with RTL can identify all signals traced in the buffer, which means the attack with RTL has more information to start with. Second, the restoration in Algorithm 2 (with RTL knowledge) can deterministically propagate values forward and backward in the AES circuit, while the restoration in Algorithm 4 (without RTL) would need a lot more brute-force effort to test and verify all possible mappings if the *sbox* lookup table cannot find a unique mapping.

6.2 Case Study 2: Pipelined AES Ciphers

The main difference from the iterative version is that the pipelined implementation unrolls all the encryption rounds to be independent hardware units, which makes the pipelined version about 10-15 times as large as the iterative. For example, the pipelined AES-128 cipher has 6720 flip-flops and about 290,000 logic gates, which is roughly 10 times (10 encryption rounds) as large as the iterative AES-128. This poses a greater challenge for the restoration process, because many signal values are not inferable due to the long pathways between the known signals. Only signals that are very close to the input can be propagated backward and possibly restore the primary key bits.

We explore different trace buffer sizes with buffer widths of 8, 16, 32 and 64, buffer depth of 512 in our experiments. We set the buffer depth to be 512 cycles, which should be suitable for the pipelined AES ciphers. Table 4 shows the experimental results on the pipelined implementation of AES-128, AES-192, and AES-256 ciphers by using the attack method with RTL knowledge. For a buffer width of 64, we are able to respectively restore 20, 19 and 44 bits of the primary key for AES-128, AES-192 and AES-256 in a few hours.

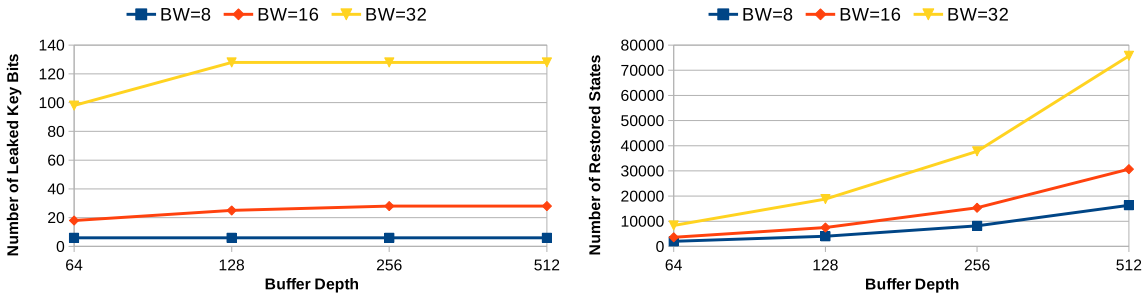
Figure 8 shows our experimental results of pipelined AES ciphers as we increase the trace buffer width. As the trace buffer width increases, both observability and the leaked number of key bits increase. The restoration algorithm is not able to restore the full primary key for any of the pipelined AES ciphers. Nevertheless, considerable knowledge about the key is gained, which does not suffice to recover the secret though, can aid other modes of cryptanalysis.

7 Proposed Countermeasures

Trace buffer attack is possible because the attacker can observe the internal values of the circuit by taking advantage of the trace buffer used for DfT. One approach to obtain a secure IC is to blow test circuitry [19] after production test. This technique is broadly used in the smartcard community, which guarantees that the chip secrecy will not be abused as a test engineer could do. However, it is not acceptable from the SoC point of view because this technique disables the test mode activation after production test. This contradicts the purpose of trace buffer for online monitoring and offline debugging in post-silicon debug.

In scan chain based DfT, several solutions have been proposed so that scan chains can provide visibility without compromising security [26, 28, 33]. Most of these approaches scrambles the structure of scan chain and make the scanned outputs difficult or impossible for the attacker to comprehend. Paul et al. [28] scramble the scan chain by reordering the scan cells and only the authorized user can get the correct order. Sengar et al. [33] insert inverters to scan chains to make it difficult for attackers to understand the internal scan structure. Another secure scan architecture is proposed in [26], which reorganizes the scan chain in a tree structure. However, these techniques cannot be directly applied to trace buffer because we don't have a chain-like structure in the trace buffer. The ultimate goal of the countermeasure is to protect the content of the trace buffer. In fact, any approach that can encrypt a block of memory will be applicable here. We explore a LFSR-based approach and

[2] For iterative implementation, the restoration is clearly able to recover the key and we expect the same trend to follow for AES-192 and AES-256.



(a) Number of primary key bits leaked

(b) Number of flip-flop states restored

Fig. 7 Iterative AES-128: security and observability trade-off using Buffer Widths (BW) of 8, 16 and 32, and Buffer Depths of 64, 128, 256 and 512. The 32×128, 32×256, and 32×512 trace buffers are able to recover the full primary key

a PUF-based approach and compare the pros and cons of these two approaches.

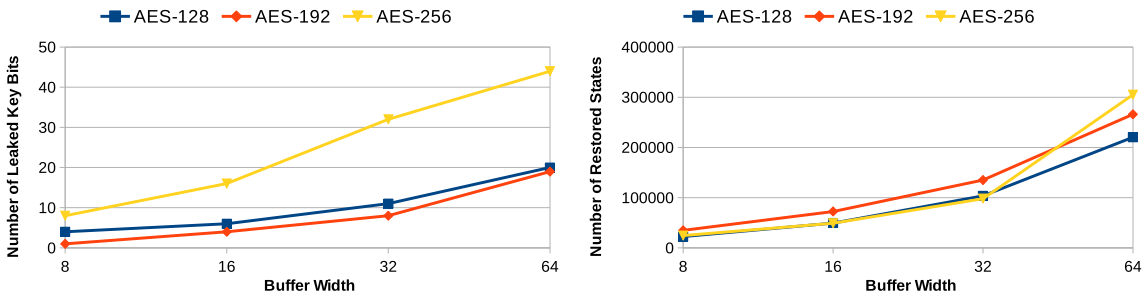
The LFSR-based approach, as shown in Fig. 9, uses a Left Feedback Shift Register (LFSR) to scramble the traced signals before they are recorded in the trace buffer. LFSR requires an initial value (i.e. the seed) to set the initial state of the shift register, and a well-chosen feedback function (XOR of some bits as in this example). LFSR can produce a sequence of pseudo-random numbers based on the seed and feedback function. The pseudo-random number will be added to the traced signals at each clock cycle. The structure of the LFSR is simple and the overhead for implementing LFSR would be minimal. Considering a trace buffer of 32-bit width, we need a 32-bit LFSR. Suppose the feedback function is XOR of eight selected bits, we would need 32 flip-flops for the shift register, and 7 two-input XOR gates for feedback function. We also need 32 XOR gates for adding the pseudo-random number with the original trace signals. Thus, the overhead is 32 flip-flops and 39 gates, which is minimal. The drawback with LFSR-based approach is that the pseudo-random sequence depends on the secrecy of the seed. The seed needs to be properly maintained as a secret by key management.

Table 4 Pipelined AES-128, AES-192, and AES-256: number of bits in the key recovered and memory/time requirements for signal restoration

BufferWidth \ AESciphers		AES-128	AES-192	AES-256
8	Leaked key (bits)	4	1	8
	Memory (GB)	4.66	5.37	6.56
	Time (h:mm:ss)	3:51:45	4:29:05	6:38:06
16	Leaked key (bits)	6	4	16
	Memory (GB)	4.66	5.37	6.56
	Time (h:mm:ss)	3:44:14	4:12:22	6:22:59
32	Leaked key (bits)	11	8	32
	Memory (GB)	4.66	5.37	6.56
	Time (h:mm:ss)	3:19:12	4:10:25	6:31:08
64	Leaked key (bits)	20	19	44
	Memory (GB)	4.66	5.37	6.56
	Time (h:mm:ss)	3:42:02	4:08:43	6:03:15

The PUF-based approach, as shown in Fig. 10, uses a Physical Unclonable Function (PUF) to introduce built-in randomness into the traced signals. The idea of this countermeasure closely follows a similar countermeasure proposed for scan-chain attacks [7]. The signals from consecutive clock cycles are XOR-ed according to a PUF response. Since the PUF response is only known to the valid user, he/she can recover the trace signal values easily. For a malicious user, recovering the original trace signal values is hard. PUF provides a challenge-response mechanism, where the mapping from a challenge to a response is controlled by the manufacturing process as well as the nature of the Integrated Circuit (IC). This complex control makes PUF structures hard to clone and at the same time a unique device identification can be obtained. Compared to the look-up table-based storage of key, PUF provides a large set of challenge-response keys with a storage requirement that increases linearly with the number of challenge bits. Only a valid user is aware of the challenge-response sets. The drawback with PUF-based approach is that a reliable PUF ("strong" PUF) has very high overhead. An arbiter-based strong PUF [14] has been implemented in $0.02mm^2$ chip area in 180 nm fabrication technology. Another SRAM-based strong PUF [10] is implemented with $0.08mm^2$ chip area in 65 nm technology.

As shown in Table 5, PUF-based approach provides stronger protection than LFSR-based method but incurs higher area overhead. An LFSR is a linear system, which leads to easy cryptanalysis [37]. A recent work on fault countermeasures [6] has shown that a bad choice of internal randomness source can lead to the complete failure of the countermeasure itself. For the PUF-based countermeasure, the randomness comes from the manufacturing process as well as the nature of the Integrated Circuit (IC). Compared with block-cipher or stream-cipher (including LFSR)



(a) Number of primary key bits leaked (b) Number of flip-flop states restored

Fig. 8 Pipelined AES-128, AES-192, and AES-256 ciphers: security and observability trade-off

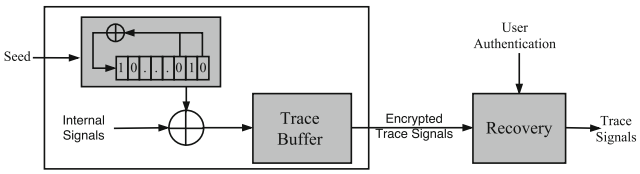


Fig. 9 LFSR-based Countermeasure

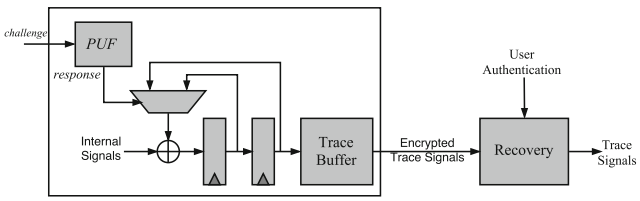


Fig. 10 PUF-based Countermeasure

Table 5 Comparison of LFSR-based and PUF-based countermeasures

Countermeasures	Protection level	Area overhead
LFSR-based approach	Low	Low
PUF-based approach	High	High

based countermeasures, the PUF-based countermeasure will be the most robust. To protect from trace buffer attack, the designer needs to trade-off between protection level and area overhead of different countermeasures. PUF-based countermeasure should always be chosen if area overhead is acceptable.

8 Conclusion

In this paper, we introduce a novel attack, *Trace Buffer Attack*, on the AES cipher. The attack is mounted with the help of trace buffers, which provides observability for post-silicon debug. We identify this as a source of information leakage and experimentally demonstrate that AES, the currently dominant block cipher, is vulnerable. We show that we can mount a strong attack with the knowledge of the RTL implementation. We are also able to take advantage of the patterns in Rijndael's key expansion, and restore the primary key even when RTL implementation is not available. With a trace buffer size of 32 × 128, the full key of the iterative AES-128 can be restored in a few minutes. For pipelined AES, partial key can be restored in a few hours. This work illustrates the need for security-aware trace signal selection, and highlights the need for further research in understanding the trade-off between security and debug observability.

Acknowledgments This work was partially supported by the NSF grants (CCF-1218629 and CNS-1441667) and SRC grant (2014-TS-2554). We would like to thank Prof. Anupam Chattopadhyay (Nanyang Technological University, Singapore) for his helpful comments and suggestions.

References

1. Arm embedded trace buffer. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0168b/ar01s03s03.html [Online]
2. OpenCores AES ciphers. http://opencores.org/project,aes_core and http://opencores.org/project,tiny_aes [Online]
3. FIPS 197, Advanced Encryption Standard. csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001 [Online]

4. Ali S, Sinanoglu O, Karri R (2014) Aes design space exploration new line for scan attack resiliency. In: 2014 22nd international conference on very large scale integration (VLSI-SoC), pp 1–6
5. Ali S, Sinanoglu O, Karri R (2014) Test-mode-only scan attack using the boundary scan chain. In: 2014 19th IEEE European test symposium (ETS), pp 1–6
6. Banik S, Bogdanov A (2015) Cryptanalysis of two fault countermeasure schemes. In: Proceedings of the 16th international conference on progress in cryptology INDOCRYPT 2015. Springer-Verlag, pp 241–252
7. Banik S, Chattopadhyay A, Chowdhury A (2014) Cryptanalysis of the double-feedback xor-chain scheme proposed in indocrypt 2013. In: Meier W, Mukhopadhyay D (eds) Progress in cryptology – INDOCRYPT 2014, pp 179–196
8. Barenghi A, Breveglieri L, Koren I, Naccache D (2012) Fault injection attacks on cryptographic devices Theory, practice, and countermeasures. Proc IEEE 100(11):3056–3076
9. Basu K, Mishra P (2013) Rats: Restoration-aware trace signal selection for post-silicon validation. IEEE Trans Very Large Scale Integr VLSI Syst 21(4):605–613
10. Bhargava M, Mai K (2014) An efficient reliable puf-based cryptographic key generator in 65nm cmos. In: 2014 design, automation test in Europe conference exhibition (DATE), pp 1–6
11. Bogdanov A, Khovratovich D, Rechberger C (2011) Biclique cryptanalysis of the full aes. In: Proceedings of the 17th international conference on the theory and application of cryptology and information security, ASIACRYPT11. Springer-Verlag, pp 344–371
12. Chatterjee D, McCarter C, Bertacco V (2011) Simulation-based signal selection for state restoration in silicon debug. In: 2011 IEEE/ACM international conference on computer-aided design (ICCAD), pp 595–601
13. Chester Rebeiro SB, Mukhopadhyay D (2015) Timing channels in cryptography: A micro-Architectural perspective. Springer
14. Devadas S, Suh E, Paral S, Sowell R, Ziola T, Khandelwal V (2008) Design and implementation of puf-based "unclonable" rfid ics for anti-counterfeiting and security applications. In: 2008 IEEE international conference on RFID, pp 58–64
15. Farahmandi F, Huang Y, Mishra P (2017) Trojan localization using symbolic algebra. In: Asia and south pacific design automation conference (ASPDAC)
16. Farahmandi F, Morad R, Ziv A, Nevo Z, Mishra P (2017) Cost-effective analysis of post-silicon functional coverage events. In: Design Automation and Test in Europe (DATE)
17. Genkin D, Pachmanov L, Pipman I, Tromer E (2015) Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In: International workshop on cryptographic hardware and embedded systems. Springer, pp 207–228
18. Genkin D, Shamir A, Tromer E (2014) RSA key extraction via low-bandwidth acoustic cryptanalysis. Springer, pp 444–461
19. Hely D, Flottes ML, Bancel F, Rouzeyre B, Berard N, Renovell M (2004) Scan design and secure chip [secure ic testing]. In: Proceedings of the 10th IEEE international on-line testing symposium, pp 219–224
20. Huang Y, Bhunia S, Mishra P (2016) Mers: Statistical test generation for side-channel analysis based trojan detection. In: Proceedings of the 2016 ACMSIGSAC conference on computer and communications security, CCS '16. ACM, pp 130–141
21. Huang Y, Chattopadhyay A, Mishra P (2015) Trace buffer attack: Security versus observability study in post-silicon debug. In: 2015 IFIP/IEEE international conference on very large scale integration (VLSI-SoC), pp 355–360
22. Kocher PC, Jaffe J, Jun B (1999) Differential power analysis. In: Proceedings of the 19th annual international cryptology conference on advances in cryptology, CRYPTO 99. Springer-Verlag, pp 388–397
23. Li M, Davoodi A (2014) A hybrid approach for fast and accurate trace signal selection for post-silicon debug. IEEE Trans Comput-Aided Design Integ Circ Syst 33(7):1081–1094
24. Moradi A, Poschmann A, Ling S, Paar C, Wang H (2011) Pushing the limits a very compact and a threshold implementation of AES. Springer, pp 69–88
25. Mukhopadhyay D (2009) An improved fault based attack of the advanced encryption standard. In: Preneel B (ed) Progress in cryptology: AFRICACRYPT 2009, volume 5580 of lecture notes in computer science. Springer, pp 421–434
26. Mukhopadhyay D, Banerjee S, RoyChowdhury D, Bhattacharya BB (2005) Cryptoscan: A secured scan chain architecture. In: 14th asian test symposium (ATS'05), pp 348–353
27. Osvik DA, Shamir A, Tromer E (2006) Cache attacks and countermeasures: The case of aes. In: Proceedings of the 2006 the cryptographers' track at the RSA conference on topics in cryptology, CT-RSA'06. Springer-Verlag, pp 1–20
28. Paul S, Chakraborty RS, Bhunia S (2007) Vim-scan: A low overhead scan design approach for protection of secret key in scan-based secure chips. In: 25th IEEE VLSI test symposium (VTS'07), pp 455–460
29. Rahmani K, Mishra P, Ray S (2014) Efficient trace signal selection using augmentation and ilp techniques. In: Fifteenth international symposium on quality electronic design, pp 148–155
30. Rahmani K, Proch S, Mishra P (2016) Efficient selection of trace and scan signals for post-silicon debug. IEEE Trans Very Large Scale Integr VLSI Syst 24(1):313–323
31. Rahmani K, Ray S, Mishra P (2017) Postsilicon trace signal selection using machine learning techniques. IEEE Trans Very Large Scale Integr VLSI Syst 25(2):570–580
32. Regazzoni F, Breveglieri L, Ienne P, Koren I (2012) Interaction between fault attack countermeasures and the resistance against power analysis attacks. In: Joye M, Tunstall M (eds) Fault analysis in cryptography, information security and cryptography. Springer, pp 257–272
33. Sengar G, Mukhopadhyay D, Chowdhury DR (2007) Secured flipped scan-chain model for crypto-architecture. IEEE Trans Comput Aided Des Integr Circuits Syst 26(11):2080–2084
34. Skorobogatov SP, Anderson RJ (2003) Optical fault induction attacks. Springer, pp 2–12
35. Yang B, Wu K, Karri R (2004) Scan based side channel attack on dedicated hardware implementations of data encryption standard. In: 2004 international conference on test, pp 339–344
36. Yang B, Wu K, Karri R (2005) Secure scan: a design-for-test architecture for crypto chips. In: Proceedings of the 42nd design automation conference, 2005, pp 135–140
37. Zenner E (2004) Cryptanalysis of lfsr-based pseudorandom generators – a survey Technical report

Yuanwen Huang received the B.E. degree from the Department of Control Science and Engineering, Huazhong University of Science and Technology, China, in 2012. He is currently pursuing the Ph.D. degree from the Department of Computer and Information Science and Engineering, University of Florida. His research interests include design automation of embedded systems, dynamic cache reconfiguration, energy and reliability optimization, hardware Trojan detection, hardware security and trust. He was a recipient of the Best Paper Award from the International Symposium on Quality Electronic Design in 2016.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include design automation of embedded systems, energy-aware computing, hardware security and trust, system validation and verification, reconfigurable architectures, and post-silicon debug. He received his Ph.D. in Computer Science and Engineering from the University of California, Irvine. He has published five books and more than 125 research articles in premier international journals and conferences. His research has been recognized by several awards including the NSF CAREER Award, IBM Faculty Award, three best paper awards, and EDAA Outstanding Dissertation Award. Prof. Mishra currently serves as the Deputy Editor-in-Chief of IET Computers & Digital Techniques, and as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, IEEE Transactions on VLSI Systems, and Journal of Electronic Testing. He has served on many conference organizing committees and technical program committees of premier ACM and IEEE conferences. He is currently serving as an ACM Distinguished Speaker. Prof. Mishra is an ACM Distinguished Scientist and a Senior Member of IEEE.