Trace Buffer Attack: Security versus Observability Study in Post-Silicon Debug

Yuanwen Huang*, Anupam Chattopadhyay†, Prabhat Mishra*
*University of Florida, Gainesville, Florida, USA
†Nanyang Technological University, Singapore
{*yuanwen, prabhat*}*@cise.ufl.edu, anupam@ntu.edu.sg*

***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. Along this line, in this paper, we propose a novel and effective attack, termed *Trace Buffer Attack*. Trace buffers are extensively used for post-silicon debug of digital designs. We identify this as a source of information leakage and show that, unless proper countermeasure is taken, *Trace Buffer Attack* is capable of partially recovering the secret keys of different AES implementations. We report the detailed process of trace-buffer attack with experimental results. We also propose a countermeasure in order to avoid such attack.**

Keywords: Cryptography, Cryptanalysis, Trace Buffer, Post-silicon Debug, AES.

I. 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 [1]. 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 here [2]. The protected design is still vulnerable to more sophisticated attacks and even then, incurs $4.6\times$ area- and $3.6\times$ 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. A background on AES and trace buffer is provided in Section II. Section III surveys related work on AES attack and trace buffer. Section IV describes the details of trace buffer attack. Section V presents the experimental studies. To prevent TBA, a countermeasure is proposed in Section VI. The paper is concluded with outline of future work in Section VII.

II. Background

A. 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-

This work was partially supported by the NSF grants (CCF-1218629 and CNS-1441667) and SRC grant (2014-TS-2554).

192 and AES-256, respectively[1]. We briefly review AES-128 here, for further details readers can refer to [3].

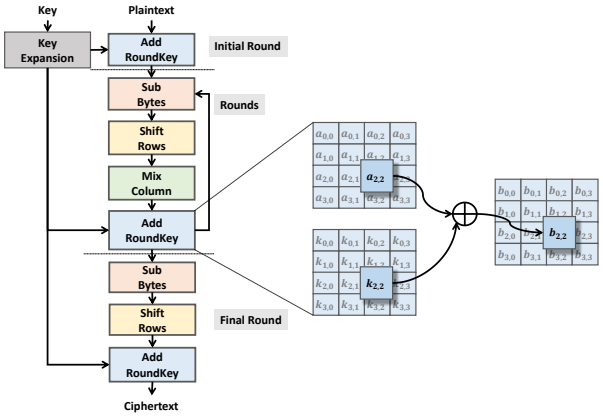


Fig. 1: AES Encryption Flow

The encryption flow of AES is shown in the Figure 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 Figure 1.

B. 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. 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 [4] 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 internal signals in the chip. The off-line debug and analysis would be based on the traced signals and the restored signals.

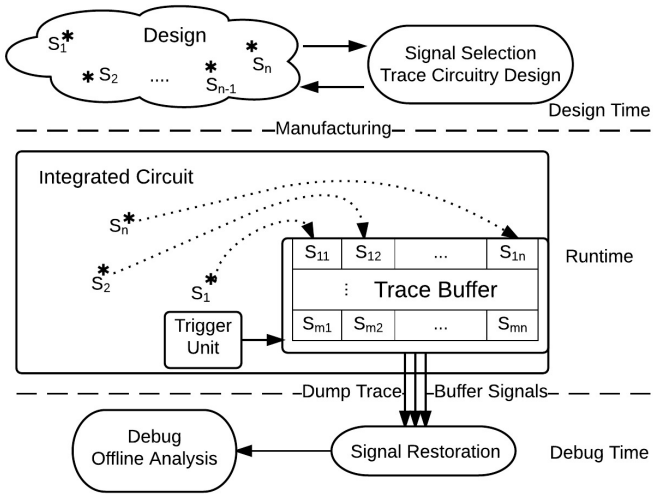


Fig. 2: Overview of trace buffer in system validation and debug

III. Related Work and Motivation

A. AES Attack

Since the pioneering works on differential power analysis [5], numerous side-channel attacks have been developed. Side-channel attacks are 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 [6] to more detailed hardware-oriented information leakages such as electromagnetic waves [7], acoustic waves [8] and optical fault injections [9]. Recent surveys on timing channels and invasive fault attacks are available in [10] and [11], 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 [12].

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

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

side-channel attacks reported against AES, attacks based on scan-chain [14] and external fault injections [15] are most prominent. For all these attacks, effective countermeasures are proposed and the inherent resilience of various design points [16] 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 [17].

B. Trace Buffer Observability versus Security

Trace buffer is widely used to improve the observability of circuit and thus assist post-silicon debug and analysis. The quality of selected trace signals will directly affect the observability that we can get from the trace buffer. 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. [18] proposed a metric based algorithm that employs total restorability for selecting the most profitable signals. Chatterjee et al. [19] proposed a simulation based algorithm which is shown to be more promising than metric based approaches. Li and Davoodi [20] proposed a hybrid approach which combines the advantages of metric and simulation based approaches.

While it is accepted in the research community that there is a strong link between observability/testability and security, 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 during post-silicon debug.

IV. Trace Buffer Attack

The proposed trace buffer 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 recover the user-specified key. Details of each step are elaborated in the following sections.

A. 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. 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.

1) Simulate the RTL implementation of the AES design with a random key k and a random input plaintext t for c cycles. During simulation, all the internal register values are stored.
2) Run the test chip with the same key k and the same input plaintext t for c cycles. Each traced signal will have a vector of c values stored in the trace buffer.
3) Dump out the values in trace buffer through JTAG. For each traced signal, we compare its value vector with all the register value vectors from RTL simulation. If a unique match is found in the RTL simulation, this traced signal is identified in the RTL description. Repeat the process until all the traced signals are uniquely identified.

For the two case studies in Section V, the above mapping process can be finished in no more than 512 cycles. For the iterative AES-128 in Section V-A, it takes 24 cycles (2 runs, each run takes 12 cycles) to uniquely identify the 32 traced signals. For the pipelined AES ciphers in Section V-B, 512 cycles suffice to uniquely identify all the 64 traced signals in each design.

B. Attack Step 2: Signal Restoration

Let us assume that the attacker has finished the preparation in Attack Step 1 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 will detail the approach for signal restoration based on trace buffer.

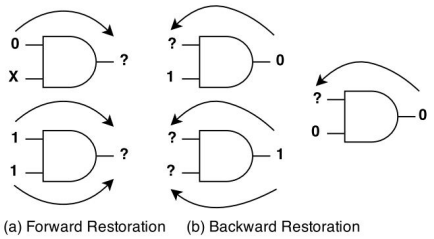


Fig. 3: Illustration of signal restoration for an AND gate

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 3(a) 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 3(b) 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 1 outlines the major steps in a typical restoration algorithm. It assigns the signal values (based on trace buffer content) of the design and performs forward and backward restorations to construct value assignments for un-traced signals. This process continues until no new assignments are created. Although this algorithm has exponential complexity, in reality, it completes the process very fast (as demonstrated in Section V) since the number of new values created decreases significantly after each iteration.

Algorithm 1 SIGNAL RESTORATION ALGORITHM

Input: Traced signals (nodes) with values at each cycle
Output: Restored signal (node) values
Put all traced nodes into $UnderProcess$ 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 $UnderProcess$ to restore its neighbors
  for *each node in* N*'s* $BackwardNeighbors$ **do**
    Backward Restoration for this neighbor node
    if *value at any cycle is restored* **then**
      Push this neighbor into $UnderProcess$
  for *each node in* N*'s* $ForwardNeighbors$ **do**
    Forward Restoration for this neighor node
    if *value at any cycle is restored* **then**
      Push this neighbor into $UnderProcess$
return

V. EXPERIMENTAL RESULTS

We use the AES Verilog implementations (the iterative AES-128 [22], and the pipelined AES-128, AES-192 and AES-256 [23]) from the OpenCores website. The Synopsys Design Compiler is used to synthesize the RTL implementation into a gate-level netlist. We develop C++ code to simulate the gate-level circuits and run the signal restoration algorithm. We use [21] to select trace signals for the trace buffers since it produces signals that can maximize observability compared to the other signal selection techniques. The signal selection algorithm picks the best signals for debugging purpose without consideration of security, which means it is ignorant of the fact that the AES design contains security secrets (keys). The experiments were conducted on a computer with AMD Opteron 2.4GHz core and 32GB memory.

A. 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*, which is a one-bit control signal.
- *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.
- *u0.rcon[24..31]* and *u0.r0.rcnt[0..3]*, which are 8 temporary registers in the key expansion unit.
- *text_out[0..127]*, which is a 128-bit register holding the ciphertext.

The signals recorded in the trace buffer are identified by using methods detailed in Section IV-A. The selected signals for each buffer width is as follows:

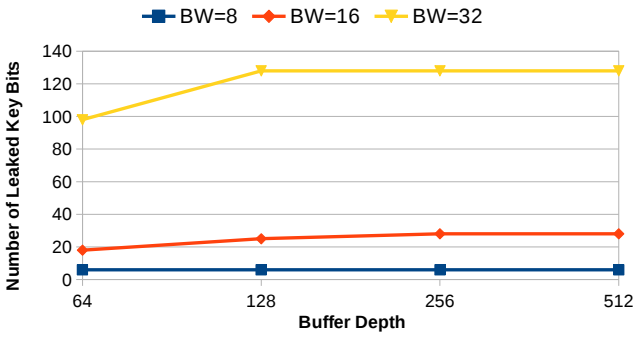
- BufferWidth=8: {dcnt[2], ld_r, w3[2], w3[1], w3[30], w3[27], w3[17], w3[13]}
- BufferWidth=16: {dcnt[2], ld_r, w3[4], w3[29], w3[27], w3[23], w3[22], w3[18], w3[16], w3[15], w3[14], w3[13], w3[12], w3[10], w1[9], w3[8]}
- BufferWidth=32: {dcnt[2], ld_r, sa03[7], sa13[7], w3[7], w3[6], w3[3], w3[2], 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 I: Iterative AES-128: Number of bits in the key recovered and memory/time requirements for signal restoration.

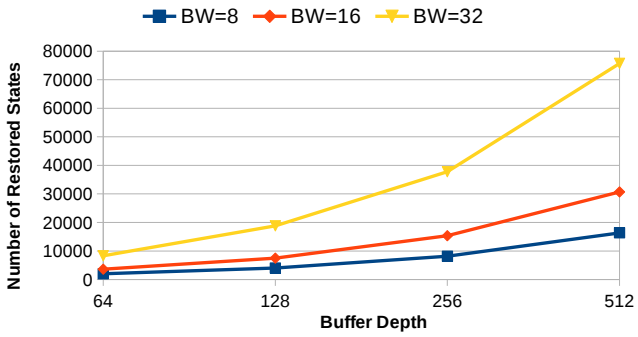
BufferWidth \ BufferDepth		64	128	256	512
8	**leaked key** (bits)	6	6	6	6
	memory (MB)	116.4	161.4	252.0	432.0
	time (mm:ss)	0:27.75	0:56.07	1:50.35	3:43.26
16	**leaked key** (bits)	18	25	28	28
	memory (MB)	116.4	161.4	252.0	432.0
	time (mm:ss)	0:27.82	0:55.94	1:51.00	3:44.10
32	**leaked key** (bits)	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

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, which should be suitable for the AES-128 design. Table I shows our results of restoring the primary key from the trace buffer content 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 4(a) shows the number of bits in the user key leaked with different buffer sizes. Figure 4(b) 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 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.



(a) Number of primary key bits leaked



(b) Number of flip-flop states restored

Fig. 4: 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.

The fact that the 32×512 trace buffer can restore all 128-bit primary key is not surprising. 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. As shown above, 2 signals are from the control unit; 2 signals are from the intermediate result register; the other 28 signals are from the round key register in the key expansion unit. The selected signals from the key expansion unit are most responsible for giving away information to restore the primary key. The success of recovering the full primary key is due to the observability provided by the trace buffer.

B. 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 II shows the experimental results on the pipelined implementation of AES-128, AES-192, and AES-256 ciphers. 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 5 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.

[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.

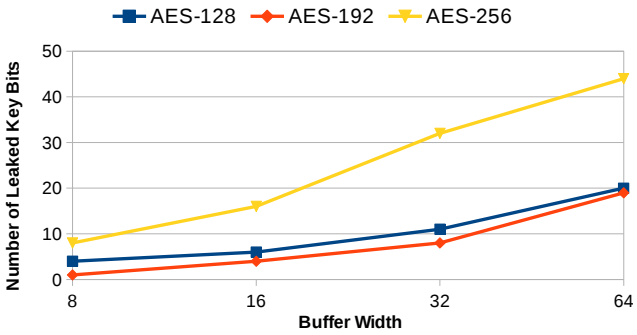
TABLE II: Pipelined AES-128, AES-192 and AES-256: Number of bits in the key recovered and memory/time requirements for signal restoration.

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

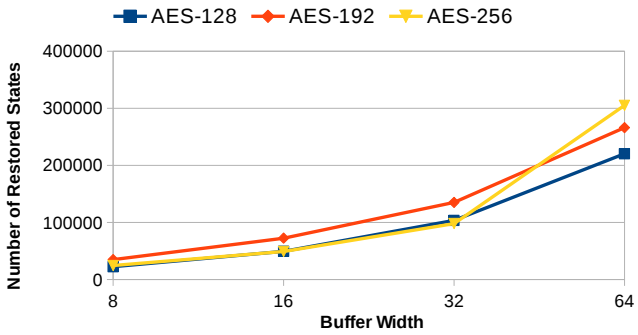
VI. Proposed Countermeasure

To prevent the trace-buffer attack, we propose a countermeasure based on built-in randomness assumption. Physically Unclonable Functions (PUFs) [24] are used as the chosen source of randomness. 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 countermeasure is graphically shown in the Figure 6. During the recording of the trace signals, 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 easily. For a



(a) Number of primary key bits leaked



(b) Number of flip-flop states restored

Fig. 5: Pipelined AES ciphers: security and observability trade-off.

malicious user, recovering the original trace signals is hard. The idea of this countermeasure closely follows a similar countermeasure proposed for scan-chain attacks [25]. Note that the countermeasure is described in generic fashion as it can be scaled to larger bit-widths as needed.

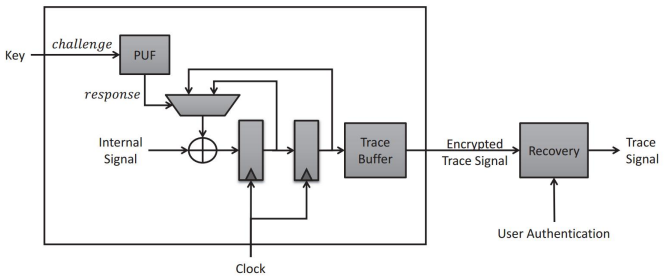


Fig. 6: PUF-based Countermeasure

VII. Summary and Outlook

In this paper *Trace Buffer Attack* is introduced. We study the current practice in trace buffer signal selection and restoration algorithms. Based on that, it is experimentally demonstrated that AES, the currently dominant block cipher, is vulnerable. With a trace buffer size of 32×128, the full key of the iterative AES-128 can be restored with a computation time of one minute. For pipelined AES, partial key can be restored in a few hours. This leads to a trade-off between security and debug observability. An efficient PUF-based countermeasure is proposed to prevent the trace buffer attack. This work can be extended in multiple directions. One may take up further experiments to determine the actual PUF overhead, to account for the PUF modeling attacks, to construct hybrid attacks involving trace buffers as well as study the vulnerability of other ciphers apart from AES.

References

[1] A. Bogdanov et al., "Biclique cryptanalysis of the full AES," in *Advances in Cryptology: ASIACRYPT 2011*.

[2] A. Moradi et al., "Pushing the limits: A very compact and a threshold implementation of AES," in *Advances in Cryptology: EUROCRYPT 2011*.

[3] *FIPS 197, Advanced Encryption Standard*, 2001. [Online]. Available: csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[4] *ARM Embedded Trace Buffer*, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0168b/ar01s03s03.html

[5] P. C. Kocher et al., "Differential power analysis," in CRYPTO , 1999. D. J. Bernstein, *Cache-timing attacks on AES*, 2005. [Online]. Available: http://cr.yp.to/papers.html#cachetiming

[6] D. Osvik et al., "Cache attacks and countermeasures: The case of AES," CT-RSA, 2006.

[7] D. Genkin et al., "Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation," Cryptology ePrint Archive, Report 2015/170, 2015.

[8] D. Genkin et al., "RSA key extraction via low-bandwidth acoustic cryptanalysis," CRYPTO, 2014.

[9] S. Skorobogatov and R. Anderson, "Optical fault induction attacks,"Springer, 2003.

[10] C. Rebeiro et al., *Timing Channels in Cryptography: A Micro-Architectural Perspective*, Springer, 2015 edition.

[11] A. Barenghi et al., "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, 2012.

[12] S. Bhunia et al., "Hardware trojan attacks: Threat analysis and countermeasures," *Proceedings of the IEEE*, 2014.

[13] B. Ege et al., "Differential scan attack on AES with x-tolerant and x-masked test response compactor," in *Digital System Design (DSD), 2012*.

[14] S. Ali et al., "Test-mode-only scan attack using the boundary scan chain," in ETS, 2014.

[15] D. Mukhopadhyay, "An improved fault based attack of the advanced encryption standard," in *Progress in Cryptology: AFRICACRYPT 2009*.

[16] S. Ali et al., "AES design space exploration new line for scan attack resiliency," in VLSI-SoC, 2014.

[17] F. Regazzoni et al., "Interaction between fault attack countermeasures and the resistance against power analysis attacks," in *Fault Analysis in Cryptography*, 2012.

[18] K. Basu and P. Mishra, "RATS: restoration-aware trace signal selection for post-silicon validation," *IEEE Trans. VLSI Syst.*, 2013.

[19] D. Chatterjee et al., "Simulation-based signal selection for state restoration in silicon debug," in ICCAD, 2011.

[20] M. Li and A. Davoodi, "A hybrid approach for fast and accurate trace signal selection for post-silicon debug," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2014.

[21] K. Rahmani et al., "Efficient trace signal selection using augmentation and ILP techniques," in ISQED 2014.

[22] *OpenCores AES-128 cipher*, [Online]. Available: http://opencores.org/project,aes_core

[23] *OpenCores AES ciphers (all key sizes)*, [Online]. Available: http://opencores.org/project,tiny_aes

[24] I. Verbauwhede and R. Maes, "Physically unclonable functions: Manufacturing variability as an unclonable device identifier," in GLSVLSI, 2011, pp. 455–460.

[25] S. Banik et al., "Cryptanalysis of the double-feedback xor-chain scheme proposed in Indocrypt 2013" in *Progress in Cryptology – INDOCRYPT 2014*.