

COSIC

# Provable Secure Software Masking in the Real-World

Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede

COSADE 2022



# Introduction

- We looked at open-source first order SW masked AES implementations and evaluated them for:
  - Side-channel leakage
  - Timing
  - Randomness requirements

Paper title	Published venue	masking method
Provably Secure Higher-Order Masking of AES	CHES 2010	boolean
Higher order masking of look-up tables	Eurocrypt 2014	boolean
All the AES You Need on Cortex-M3 and M4	SAC 2016	boolean
Consolidating Inner Product Masking	Asiacrypt 2017	inner product
First-Order Masking with Only Two Random Bits	CCS-TIS 2019	boolean
Side-channel Masking with Pseudo-Random Generator	Eurocrypt 2020	boolean
Detecting faults in inner product masking scheme	JCEN 2020	inner product
Fixslicing AES-like Ciphers	TCHES 2021	boolean

# What we found

- Key recovery with first order attack ●
- Incorrect TRNG instantiations ●
- Benchmarking issues ●
- Software bugs ●

	Paper title	Published venue	masking method
●	Provably Secure Higher-Order Masking of AES	CHES 2010	boolean
●	Higher order masking of look-up tables	Eurocrypt 2014	boolean
● ●	All the AES You Need on Cortex-M3 and M4	SAC 2016	boolean
	Consolidating Inner Product Masking	Asiacrypt 2017	inner product
●	First-Order Masking with Only Two Random Bits	CCS-TIS 2019	boolean
● ● ●	Side-channel Masking with Pseudo-Random Generator	Eurocrypt 2020	boolean
	Detecting faults in inner product masking scheme	JCEN 2020	inner product
●	Fixslicing AES-like Ciphers	TCHES 2021	boolean

# What this work is not





- An attack on the underlying theory of the masking schemes
- An in-depth security evaluation of the targeted schemes
- A critique aimed at a specific person or publication
- An attempt to discourage open-sourcing code

# Disclaimers

- Side-channel masking with PRG (<https://github.com/coron/htable/>)

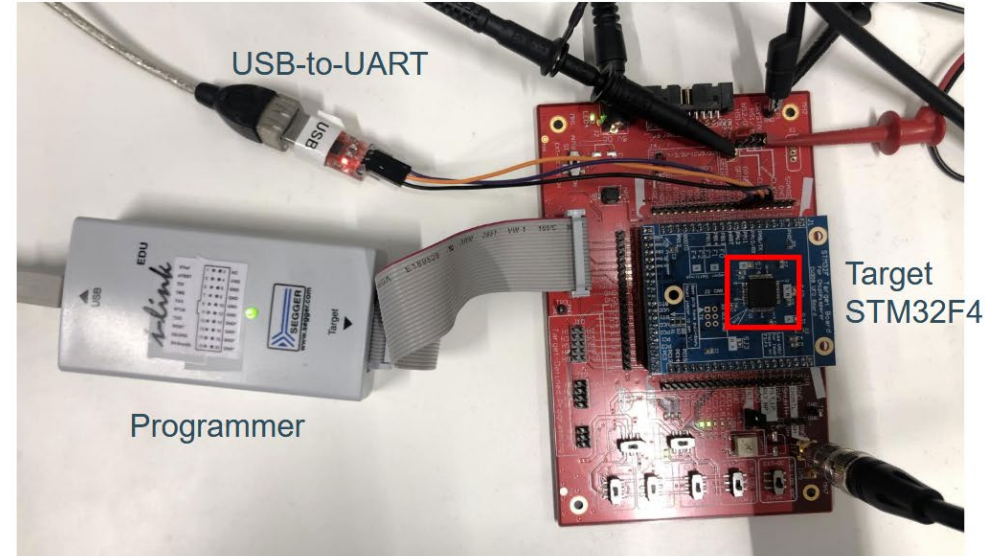
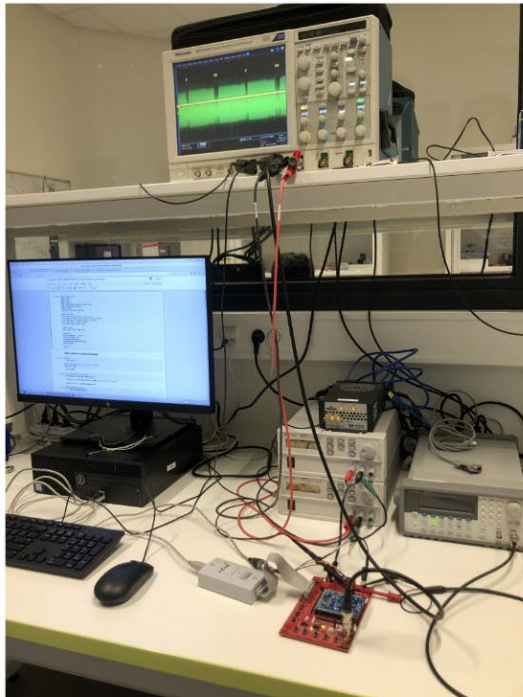
We do not claim that in practice the implementation would be secure against a  $t$ -th order attack. Namely the implementation is only provided **for illustrative purpose, and timing comparisons**. Obtaining a secure implementation would require to carefully examine the assembly code. In particular one should make sure that no two shares of the same variable are stored in the same register.

- Fixslicing AES-like ciphers (<https://github.com/aadomn/aes>)

  This masking scheme was mainly introduced to achieve first-order masking while limiting the amount of randomness to generate. Please be aware that other first-order masking schemes provide a better security level. Note that no practical evaluation has been undertaken to assess the security of our masked implementations!  

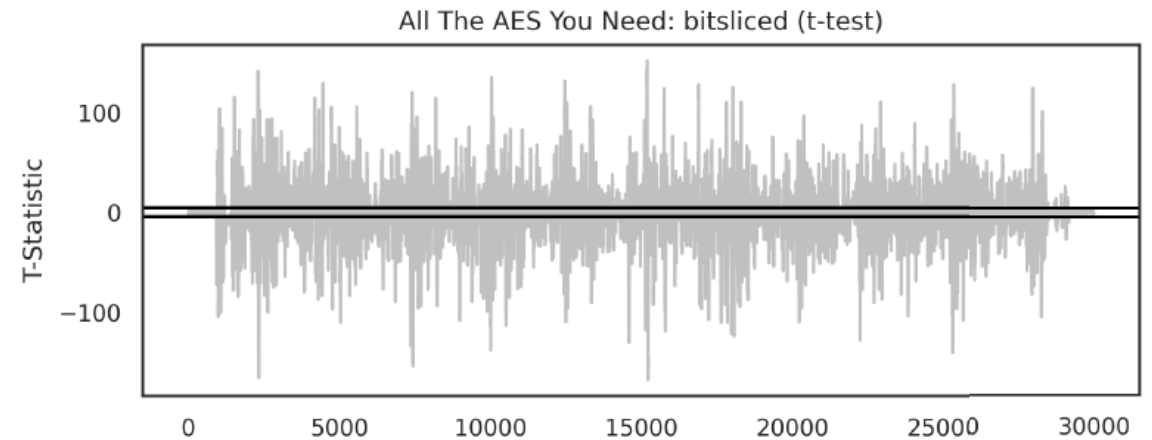
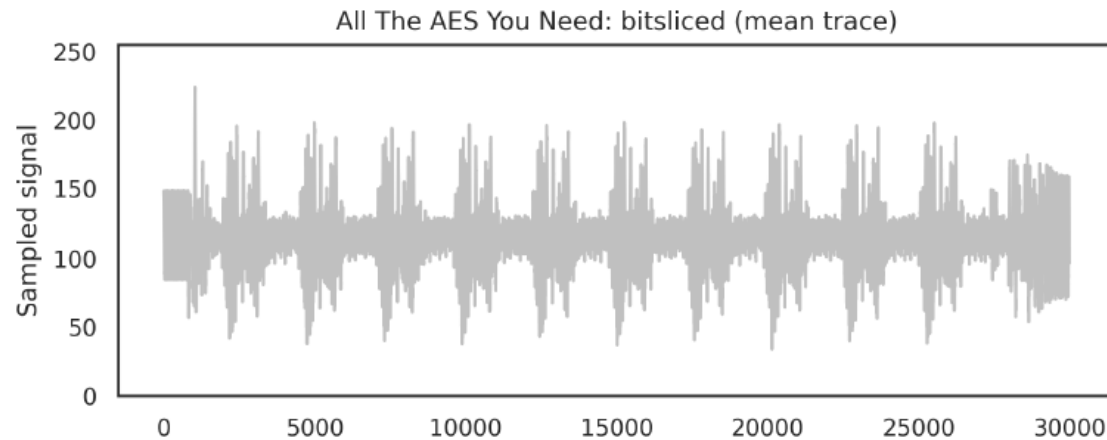
# Side channel evaluation

- Performed first order TVLA and CPA
- Same target for every evaluation: NewAE STM32F415
- Sampling rate: 200MS/s, 20MHz low-pass filter

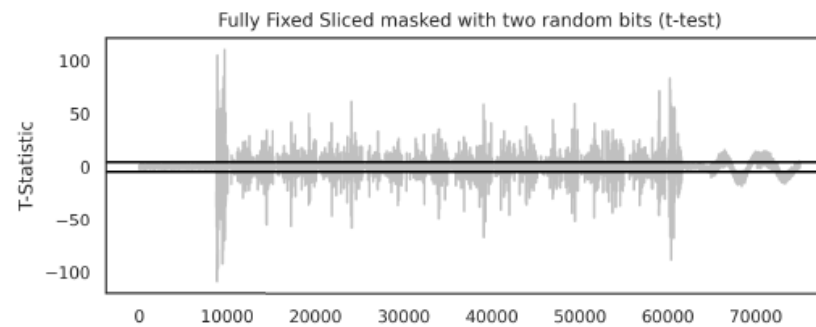
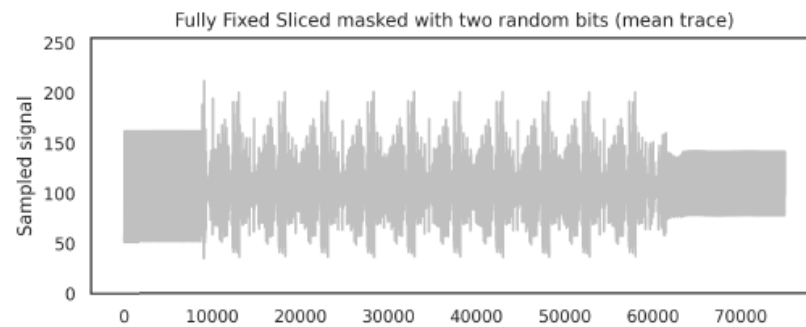
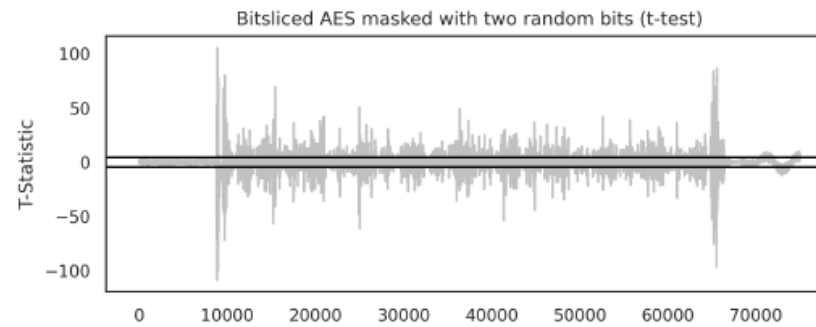
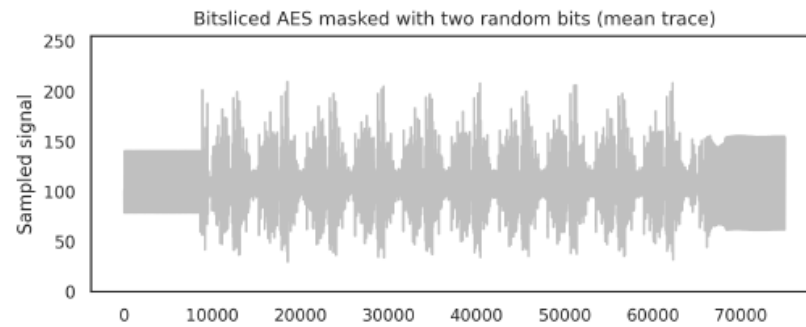
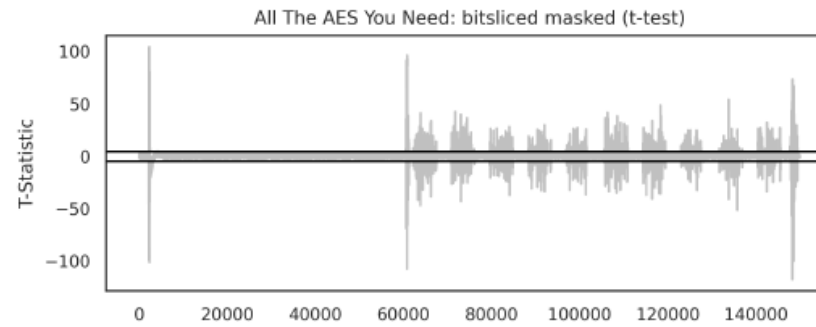
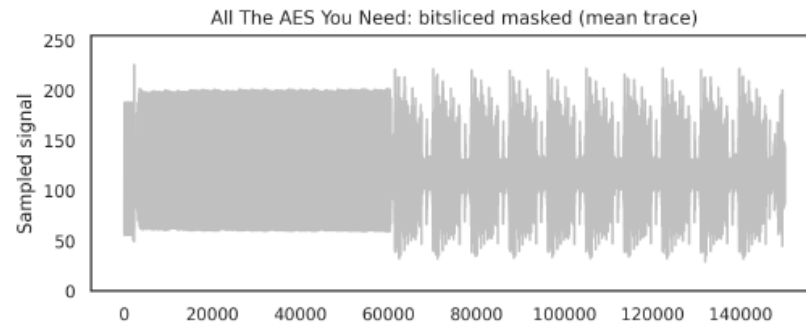


# TVLA

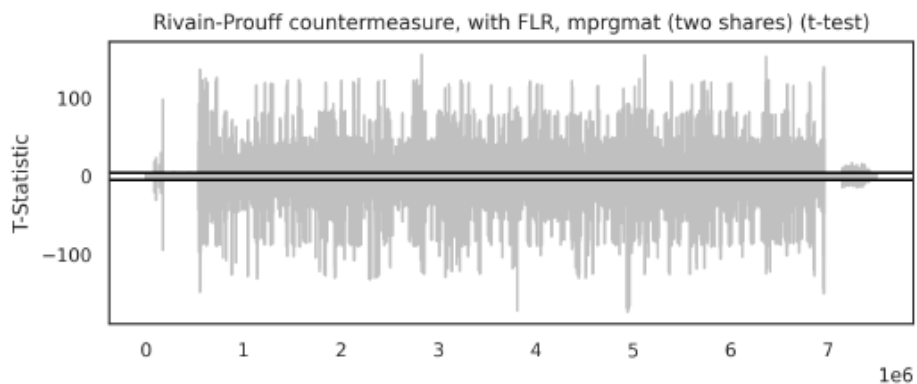
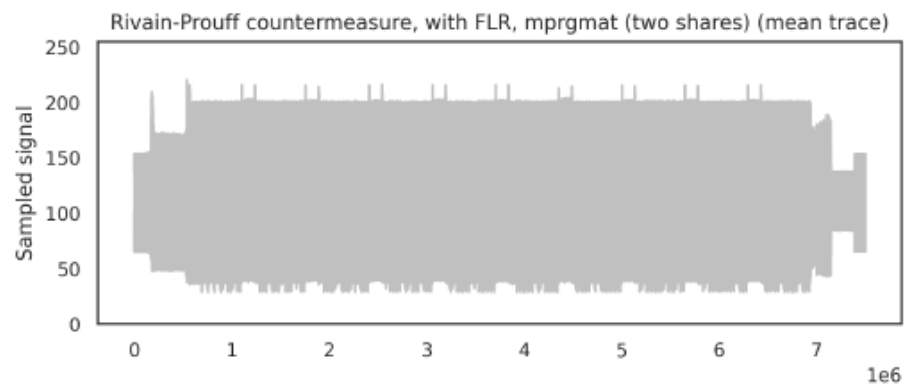
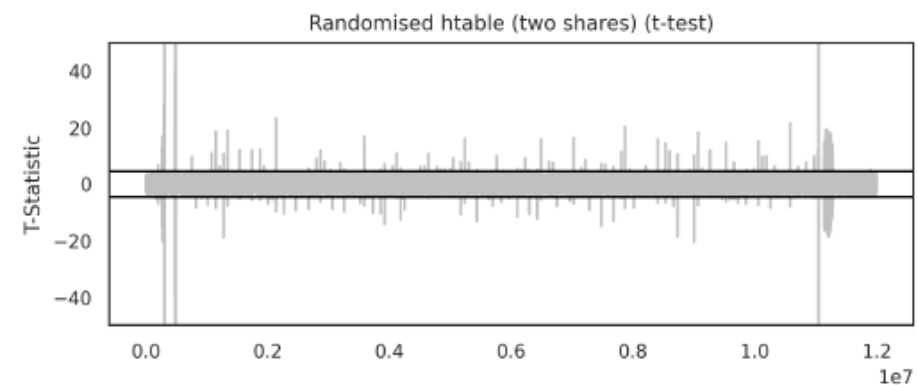
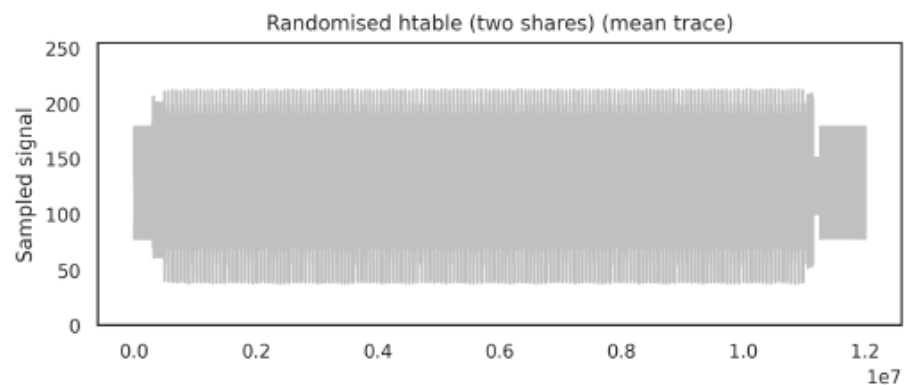
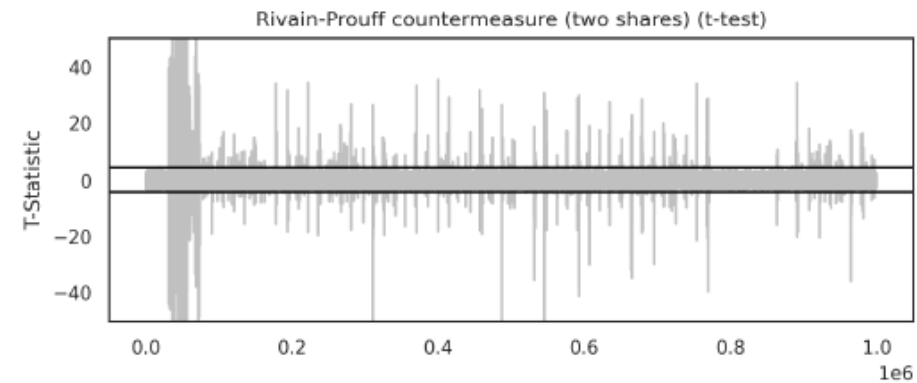
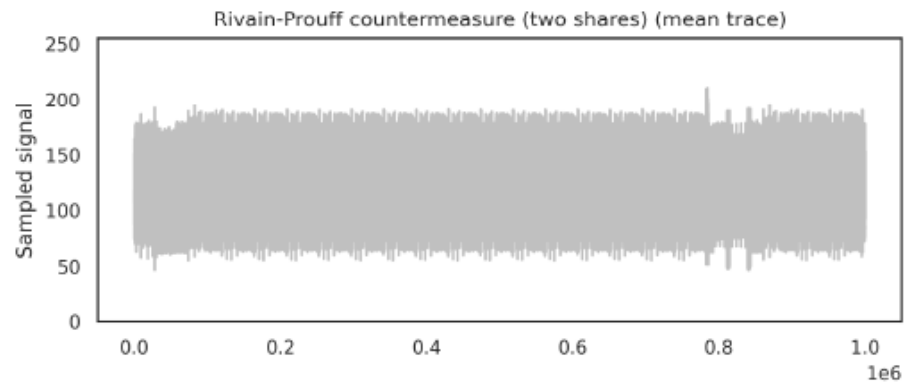
- All implementations compiled using given makefile
- Only inserted triggers
- We show only first order leakage
- 10k traces



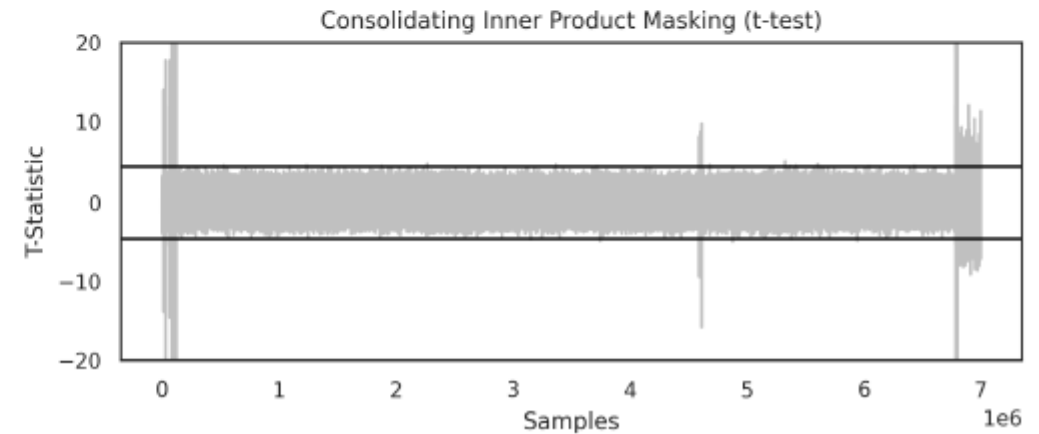
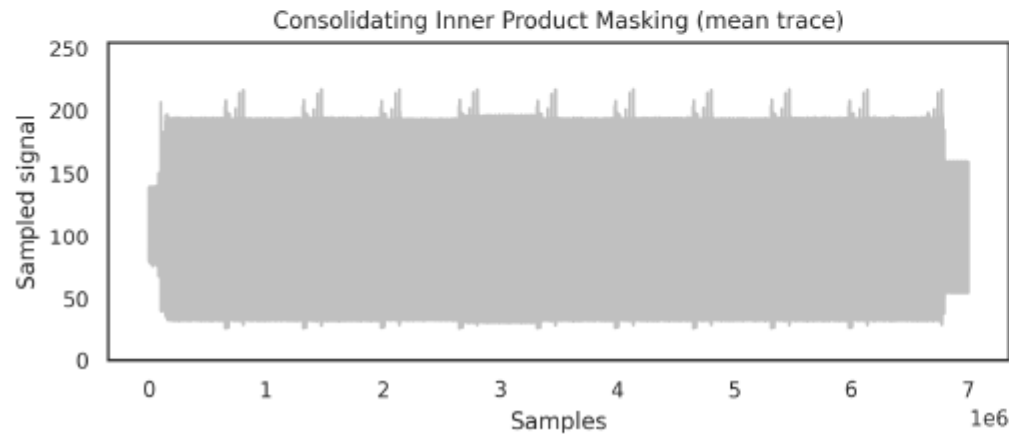
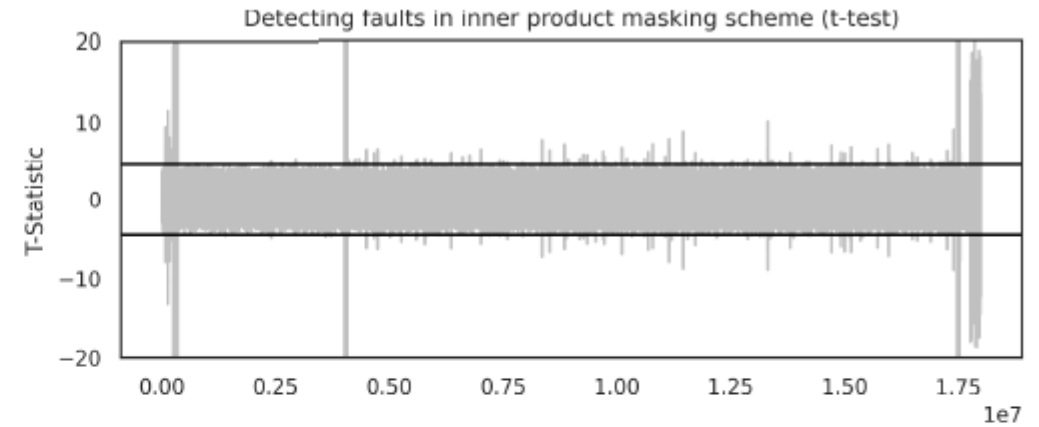
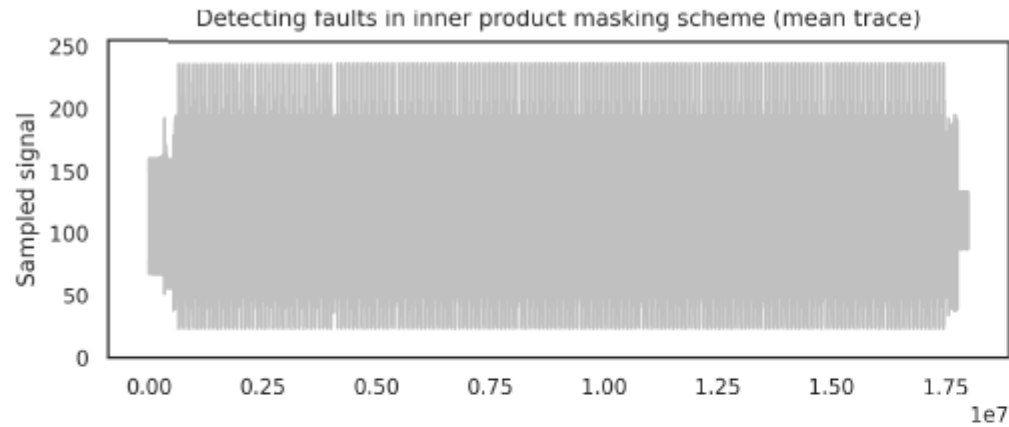
# TVLA results





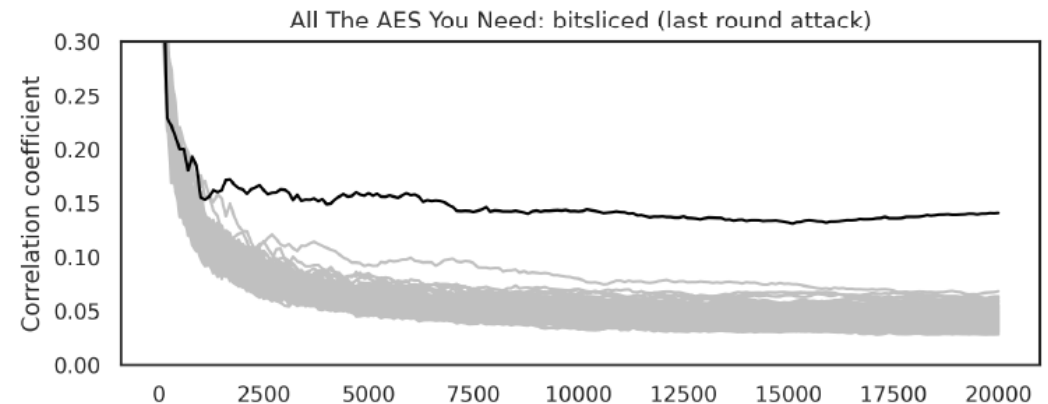
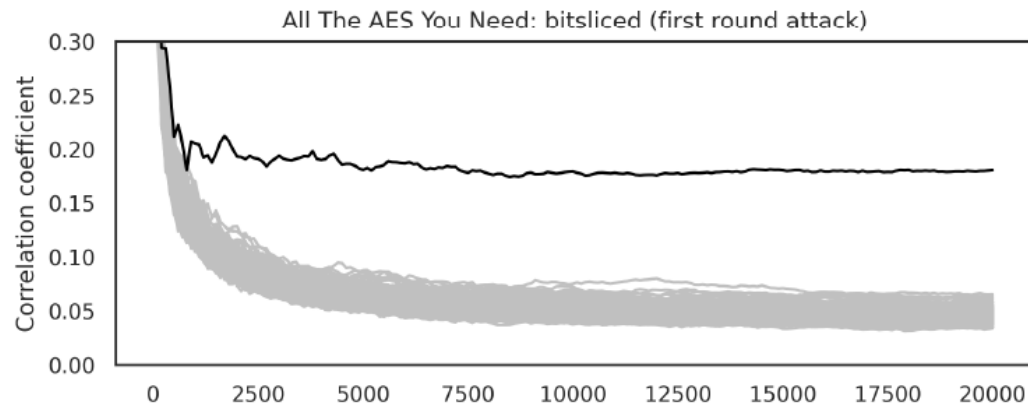


# TVLA: Inner product masking

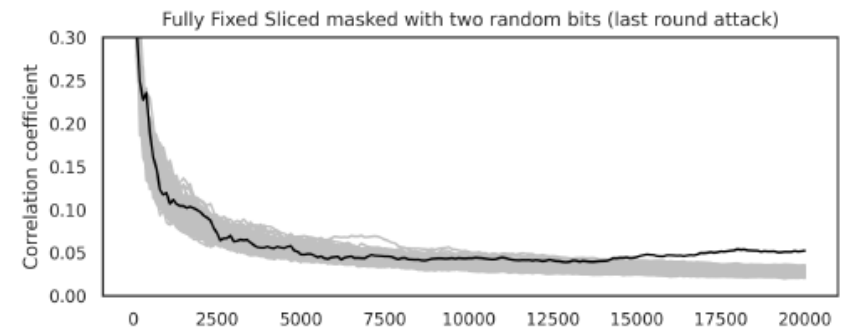
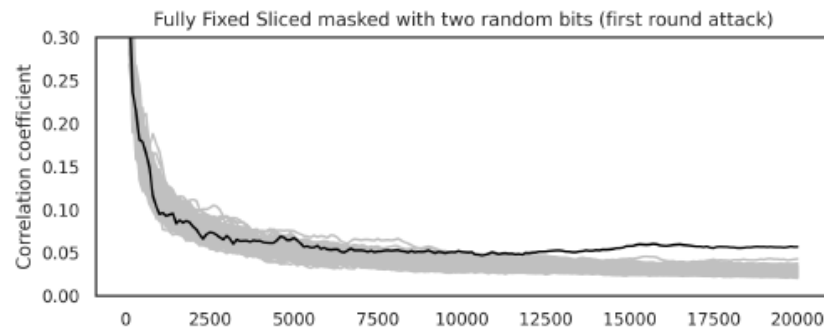
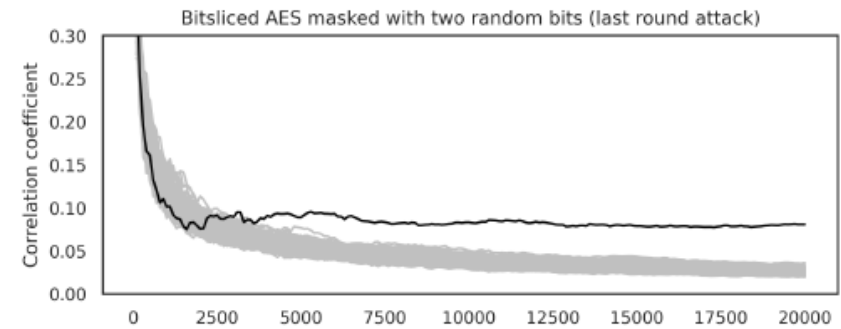
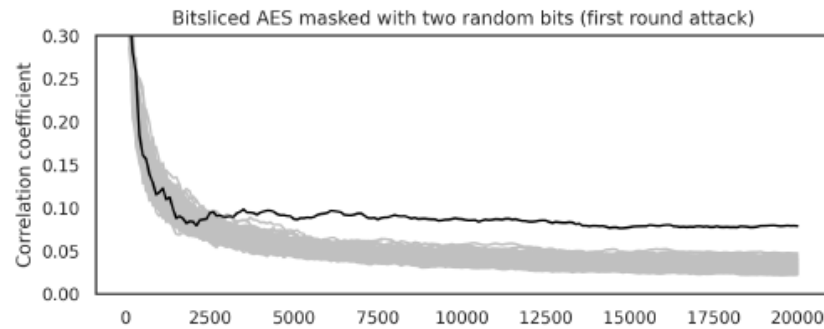
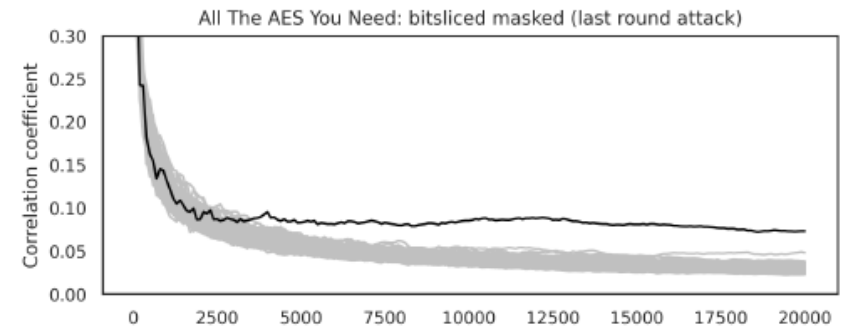
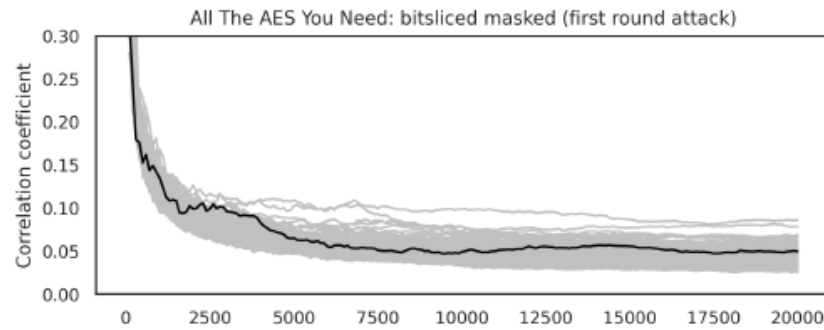


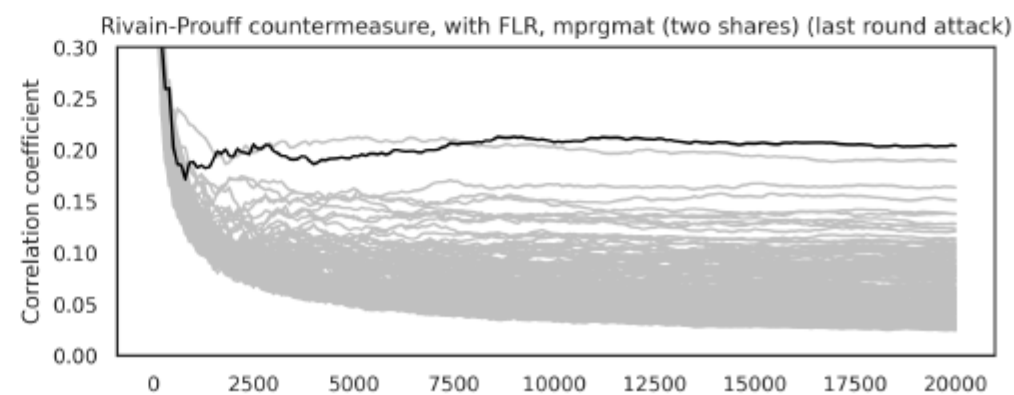
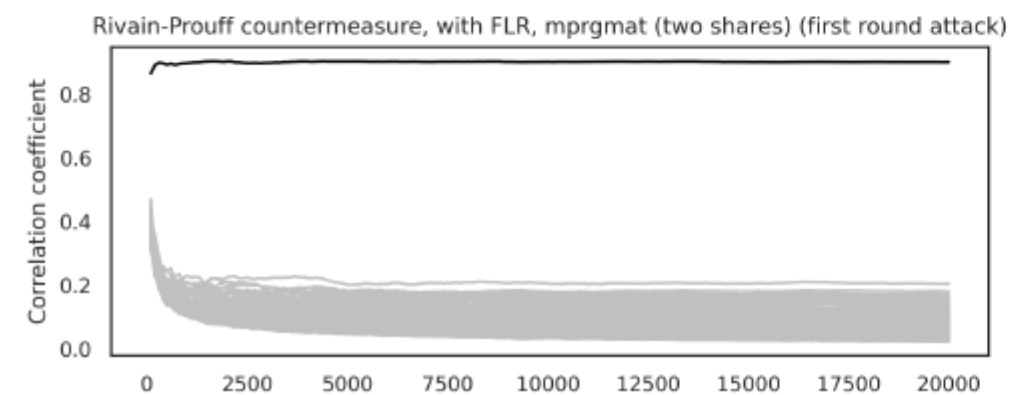
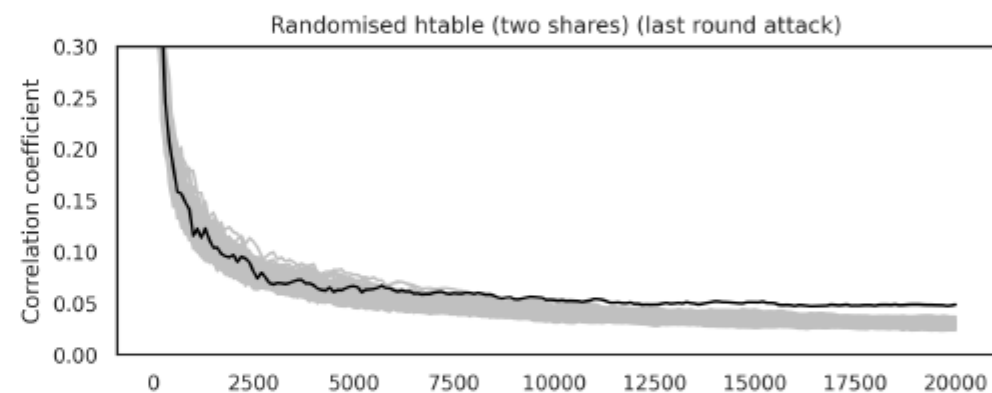
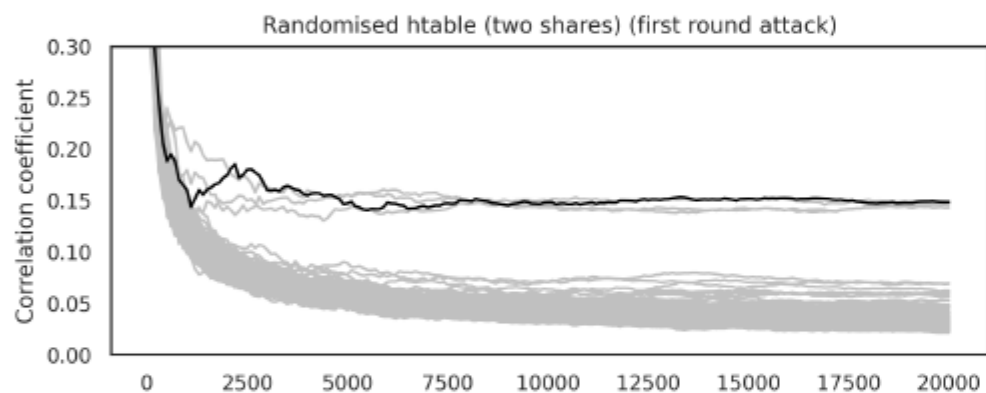
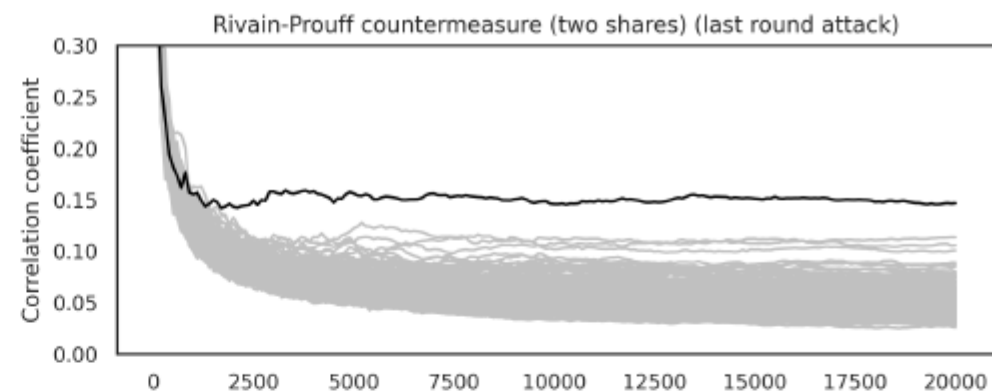
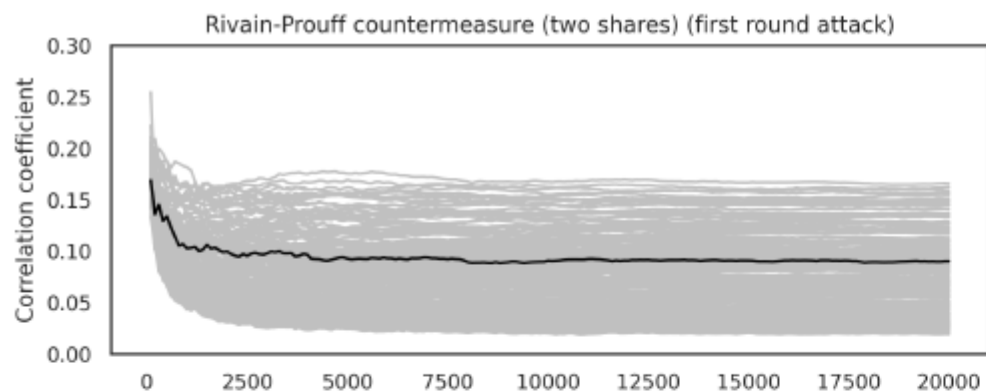
# CPA

- All implementations compiled using given makefile
- Only inserted triggers
- Textbook CPA:
  - SBOX in or output
  - HW leakage, or single bit when bitsliced
  - 20k traces

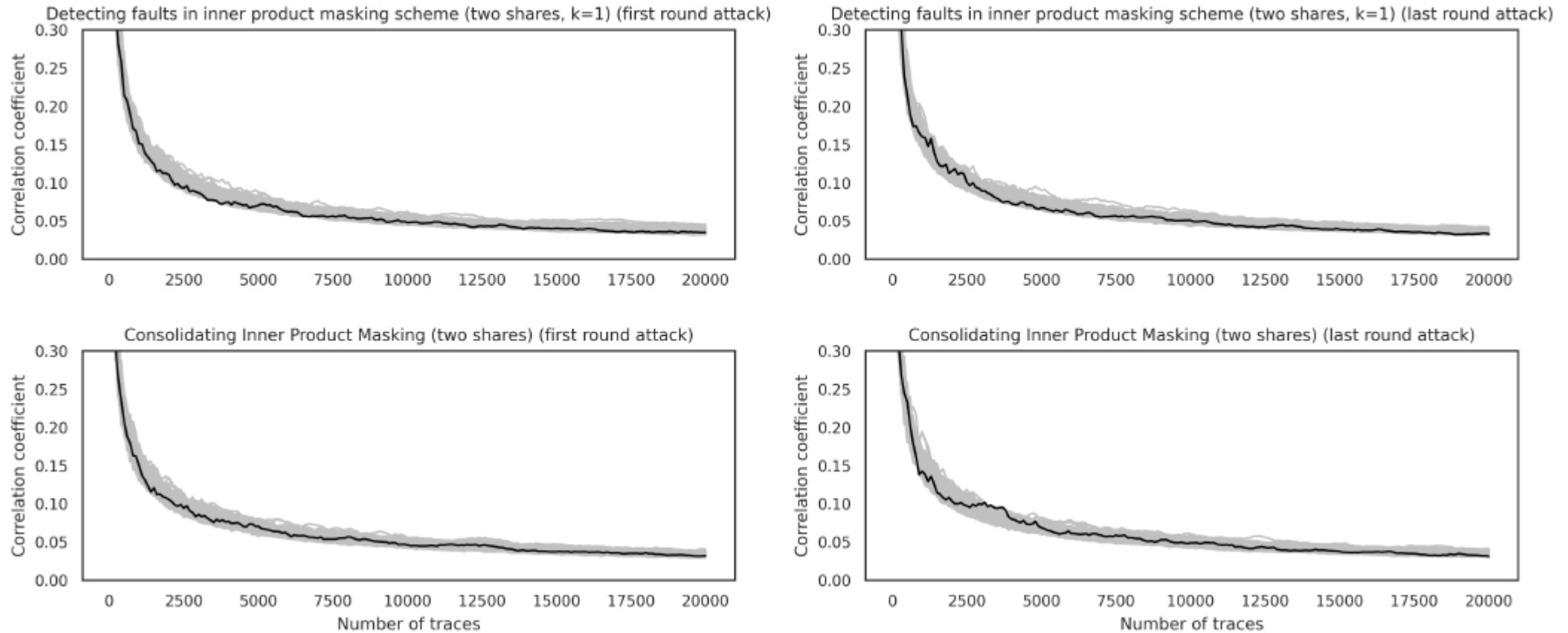


# CPA results





# CPA: inner product masking



# Root cause analysis



Unicorn Emulator

All the AES you need on Cortex-M3 and M4

```
1481 eor r11, r7, r11 //Exec y8 = x0 ^ x5; into r11
1482 eor r9, r6, r11 //Exec y3 = y5 ^ y8; into r9
1483 eor r2, r7, r2 //Exec y9 = x0 ^ x3; into r2
1484 str r11, [sp, #100] //Store r11/y8 on stack
1485 str r8, [sp, #96] //Store r8/y10 on stack
1486 str.w r5, [sp, #92] //Store r5/y20 on stack
1487 eor r11, r5, r2 //Exec y11 = y20 ^ y9; into r11
1488 eor r8, r8, r11 //Exec y17 = y10 ^ y11; into r8
1489 eor r0, r0, r11 //Exec y16 = t0 ^ y11; into r0
1490 str r8, [sp, #88] //Store r8/y17 on stack
1491 eor r5, r4, r11 //Exec y7 = x7 ^ y11; into r5
1492 ldr r8, [sp, #1496] //Exec t2 = rand() % 2; into r8
1493 str r9, [sp, #84] //Store r9/y3 on stack
1494 eor r10, r10, r8 //Exec u1 = u0 ^ t2; into r10
1495 eor r1, r10, r1 //Exec u3 = u1 ^ u2; into r1
1496 eor r3, r1, r3 //Exec u5 = u3 ^ u4; into r3
1497 eor r3, r3, r14 //Exec t2m = u5 ^ u6; into r3
1498 and r1, r9, r12 //Exec u0 = y3 & y6; into r1
1499 ldr r10, [sp, #112] //Load y6m into r10
1500 str r12, [sp, #80] //Store r12/y6 on stack
1501 and r14, r9, r10 //Exec u2 = y3 & y6m; into r14
1502 ldr r9, [sp, #120] //Load y3m into r9
1503 and r12, r9, r12 //Exec u4 = y3m & y6; into r12
```


→  $HW[y_{3m} \oplus y_3]$


[On the cost of lazy engineering for masked software implementations]

# Timing results

- All timings were done on STM32F415
- We set the system clock frequency at 24MHz and 168MHz
- Disabled caches
- Compiled with the provided make file
- Set the security order to the one used in the original paper

Implementation	cycles (measured)	randomness (RNG/PRNG)	clock frequency
All the AES You Need on Cortex-M3 and M4	17.5k	328/-	24 MHz
	62.8k	328/-	168 MHz
First-Order Masking with Only Two Random Bits	6.8k	2/-	24 MHz
	9.5k	2/-	168 MHz

 X3.6



 X1.4



# TRNG polling

Implementation	cycles (measured)	randomness (RNG/PRNG)	clock frequency
All the AES You Need on Cortex-M3 and M4	17.5k	328/-	24 MHz
	62.8k	328/-	168 MHz
First-Order Masking with Only Two Random Bits	6.8k	2/-	24 MHz
	9.5k	2/-	168 MHz

platform	function	word length	cycles	clock frequency
STM32F415 Cortex-M4	polling opencm3	32 bit	27	24 MHz
		32 bit	147	168 MHz
	polling assembly	32 bit	21	24 MHz
		32 bit	147	168 MHz
	PRNG XorShift 96	64 bit	39	24 MHz
		64 bit	63	168 MHz
LPC55S69 Cortex-M33	polling assembly	32 bit	104	25 MHz
		32 bit	361	150 MHz
SAM D5x Cortex-M4	according to datasheet	32 bit	84	24 MHz
		32 bit	84	140 MHz

 x7  
 x1.6



# Timing evaluation

Implementation	cycles measured	cycles reported	randomness (RNG/PRNG)	clock frequency
All the AES You Need on Cortex-M3 and M4	17.5k	14.8k	328/-	24 MHz
First-Order Masking with Only Two Random Bits	6.8k	6.8k	2/-	24 MHz
Fixslicing AES-like Ciphers	6.2k	6k	2/-	24 MHz
Provably Secure Higher-Order Masking of AES (n=3)	651k	20.6M*	2880/-	24 MHz
Higher order masking of look-up tables (n=3, randomized table)	9.099M	-	164,160/-	24 MHz
Side-channel Masking with Pseudo-Random Generator (n=3, multiple PRG, secmultFLR)	3.608M	12M*	52/5120	24 MHz
Consolidating Inner Product Masking	819k	-	1632/-	24 MHz
Detecting faults in inner product masking scheme (n=2, k=1)	1.650M	-	2432/-	24 MHz

# All the AES you need

- Uses STM32F4 TRNG
- Issue: misconfigured TRNG polling

Fix waiting until RNG is ready

[Browse files](#)

public

Ko- committed on Jul 12, 2020

1 parent 91799d8    commit 910d446f84b26eb84c0111d680a279b01217a500

Showing 1 changed file with 1 addition and 1 deletion.

Unified   Split

2 aes128ctrbsmasked/aes\_128\_ctr\_bs\_masked.s

...

↑ ..... @@ -1277,7 +1277,7 @@ encrypt\_blocks: //expect p in r0, RNG\_SR in r12, AES\_bsconst in r14

```
1277 1277     generate_random:
1278 1278         ldr r6, [r12]
1279 1279         tst r6, r7
1280 -       bne generate_random //wait until RNG_SR == RNG_SR_DRDY
1280 +       beq generate_random //wait until RNG_SR == RNG_SR_DRDY
1281 1281         ldr.w r6, [r5]
1282 1282         str r6, [r3, r4, lsl #2]
1283 1283         subs r4, #1
```

↓ .....

# Side-channel masking with PRNG

Implementation	cycles measured	cycles reported	randomness (RNG/PRNG)	clock frequency
Provably Secure Higher-Order Masking of AES ( $n=3$ )	651k	20.6M*	2880/-	24 MHz
Side-channel Masking with Pseudo-Random Generator ( $n=3$ , multiple PRG, secmultFLR)	3.608M	12M*	52/5120	24 MHz

## 5.4 Concrete Implementation

We have implemented our constructions for AES in C, on a 44 MHz ARM-Cortex M3 processor. The processor is used in a wide variety of products such as passports, bank cards, SIM cards, secure elements, etc. The embedded TRNG module can run in parallel of the CPU, but it is relatively slow: according to our measurements on emulator, it outputs 32 bits of random in approximately 6000 cycles. Our results, obtained by running the code on emulator, are given in Table 10, and are compared with the classical Rivain-Prouff countermeasure.

We see that the most efficient countermeasure is the SecMultFLR algorithm with multiple PRGs, using the 3-wise independent PRG. For  $n = 3$  and  $n = 4$  we obtain a 52% and 61% speedup respectively, compared to Rivain-Prouff. We provide the source code in [Cor19b].

<https://eprint.iacr.org/2019/1106.pdf>

# Recommendations

- Describe the side-channel setup in detail
- Perform a convincing side-channel leakage assessment
- List the randomness requirement of the masking scheme
- Benchmark the randomness sources
- Use a realistic benchmarking platform
- Provide all relevant platform settings
- Document the toolchain and compiler settings

# Conclusion

- A more thorough side-channel evaluation of software masking schemes should be required
- This work was only made possible by open sourcing of the implementations
- Academic code should be open sourced, but should also be checked before reuse