IEEE/ACM ICCAD
2022 INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN
41st Edition

Attack Directories on ARM big.LITTLE Processors

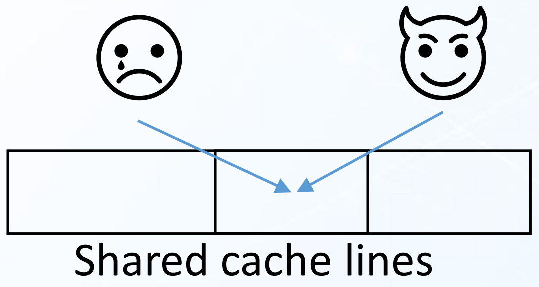
Zili KOU [1], Wenjian HE [1], Wei ZHANG [1], and Sharad Sinha [2]

[1] Hong Kong University of Science and Technology

[2] Indian Institute of Technology Goa

Cache Side-channel Attacks

- Cache side-channel
  - Utilize the timing difference between cache hit and miss



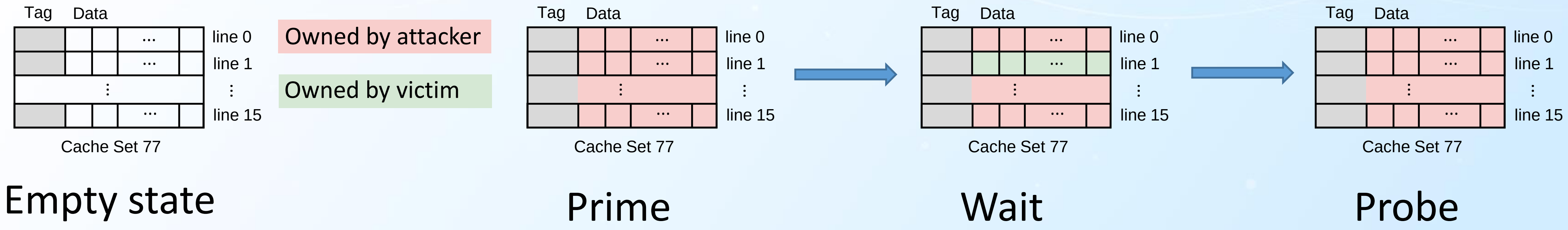
Cache hit: ≈ 20 cpu cycles

Cache miss: ≈ 100 cpu cycles

- Attack scenarios
  - covert channel communications
  - extracting cryptographic keys (RSA, AES, etc.)
  - speculative execution attacks

Cache Side-channel Attacks

- Three types
  - Coherence-based: [1], [2]
    - Exploit the cache coherence protocol between cores
  - Flush-based: Flush+Reload [3], Flush+Flush [4]
    - Clean and flush a cache line, and then reload/flush again
  - Evict-based: Prime+Probe [5]
    - Evict, occupy, and reload a whole cache set by using the **“eviction set”**



Cache Side-channel Attacks

	Cache cleaning instructions	Shared memory space
Coherence-based	Not required	Required
Flush-based	Required	Required
Evict-based	Not required	Not required

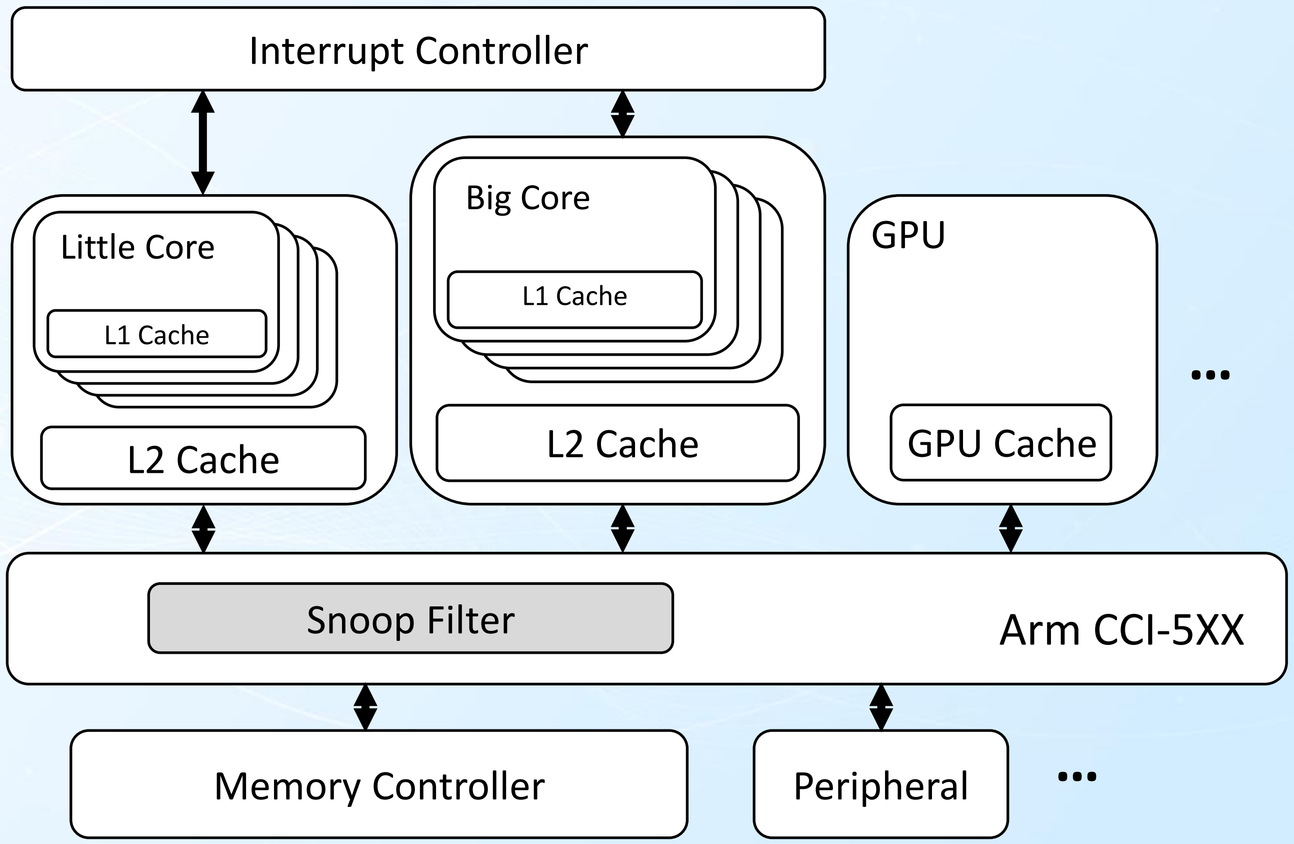
- Practicality?
  - Cache cleaning instructions is usually suggested to be forbidden
  - Shared memory with victim is impossible without page-sharing or memory deduplication
- Evict-based attacks utilize "Eviction Set (EV)"
  - Lower precision, though
  - More feasible and practical

Evict-based Cache Side-channel Attacks

- Prerequisites
  - Shared cache between attacker and victim
    - Usually exist on x86 CPUs, how about ARM CPUs?
  - Inclusive cache hierarchy
    - ARM CPUs usually adopt non-inclusive cache

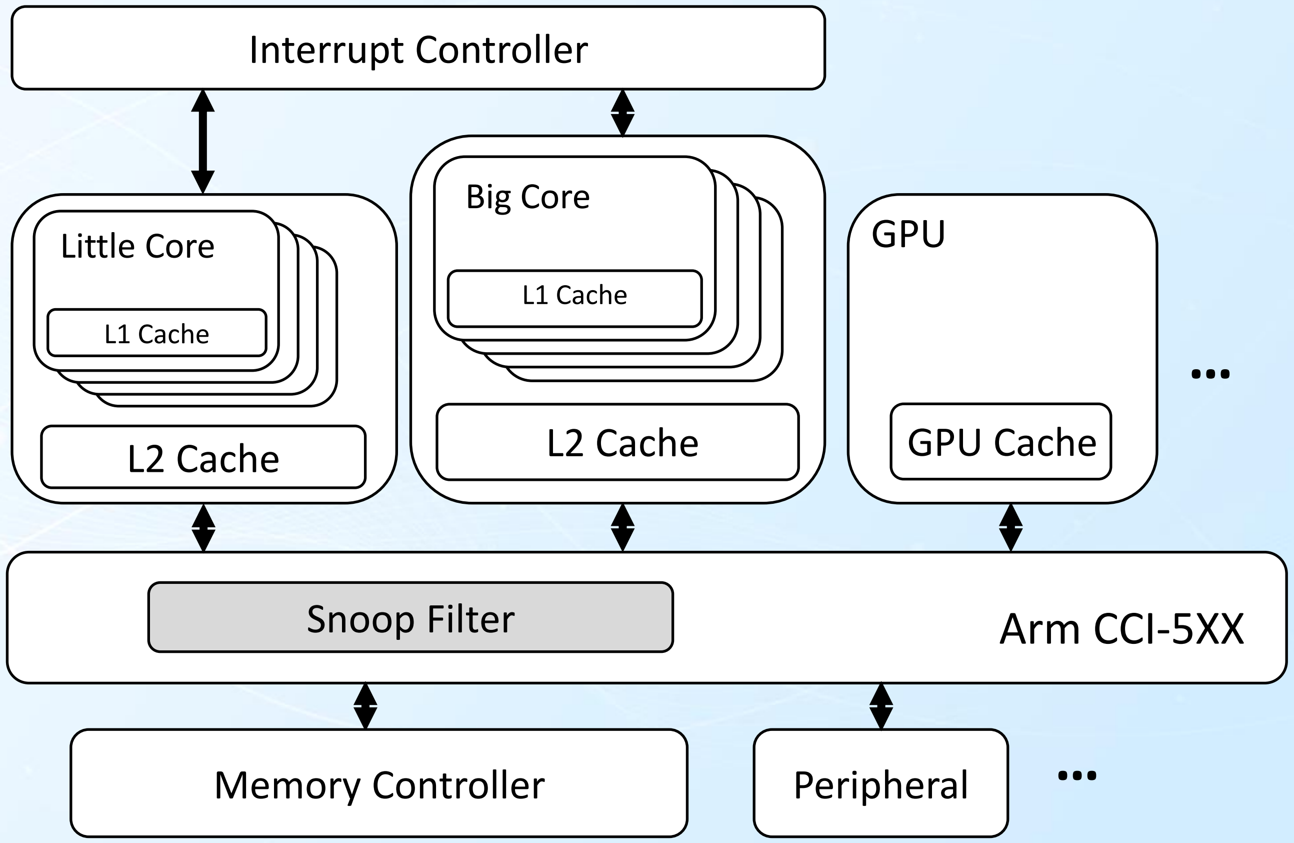
ARM big.LITTLE Arch

- Classic IPs
  - 4 A53 + 4 A73 cores
  - CCI-500/550 interconnect
- Attack scenarios
  - Single-core
  - Cross-core
  - Cross-cluster



Attack on ARM big.LITTLE Arch

- Attack scenarios
  - Single-core
    - L1 is easy to attack!
  - Cross-core
    - L2 is usually non-inclusive
  - Cross-cluster
    - **Even no shared cache!**



Cache Side-channel Attacks on ARM CPUs

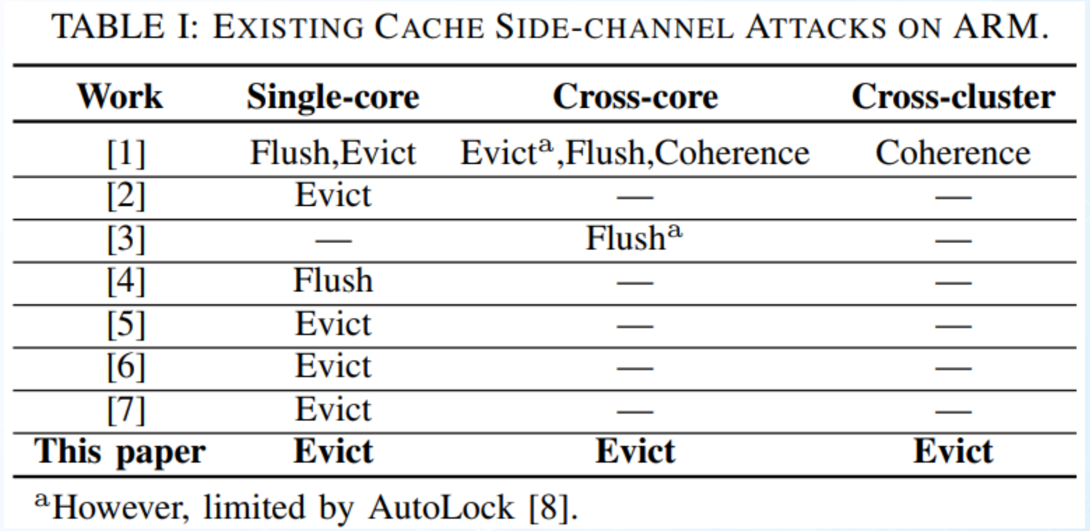


TABLE I: EXISTING CACHE SIDE-CHANNEL ATTACKS ON ARM.

Work	Single-core	Cross-core	Cross-cluster
[1]	Flush,Evict	Evict[a],Flush,Coherence	Coherence
[2]	Evict	—	—
[3]	—	Flush[a]	—
[4]	Flush	—	—
[5]	Evict	—	—
[6]	Evict	—	—
[7]	Evict	—	—
This paper	**Evict**	**Evict**	**Evict**

[a]However, limited by AutoLock [8].

- Research gap
  - Two cross-core attacks are limited by [8]
  - **No** cross-cluster evict-based attacks on ARM

Our Contribution

- Reveal and dive into the directories on ARM big.LITTLE CPUs
- Overcome the difficulties when attacking ARM big.LITTLE CPUs
- Fill the gap of cross-core / cross-cluster attacks

We introduce…

- Reverse engineering of the directory named Snoop Filter (SF)
  - **Structure and Properties**
  - SF eviction set construction
- Compare SF with cache in real attack applications
  - Covert channel
  - Attack Cryptographic algorithm
    - RSA
    - AES
  - TrustZone scenario

General design of SF

- A directory structure to store
  - Tags of cached data in each core
  - Cache coherence states
  - …
- Avoid broadcast every load/store request, reduce the bus overhead

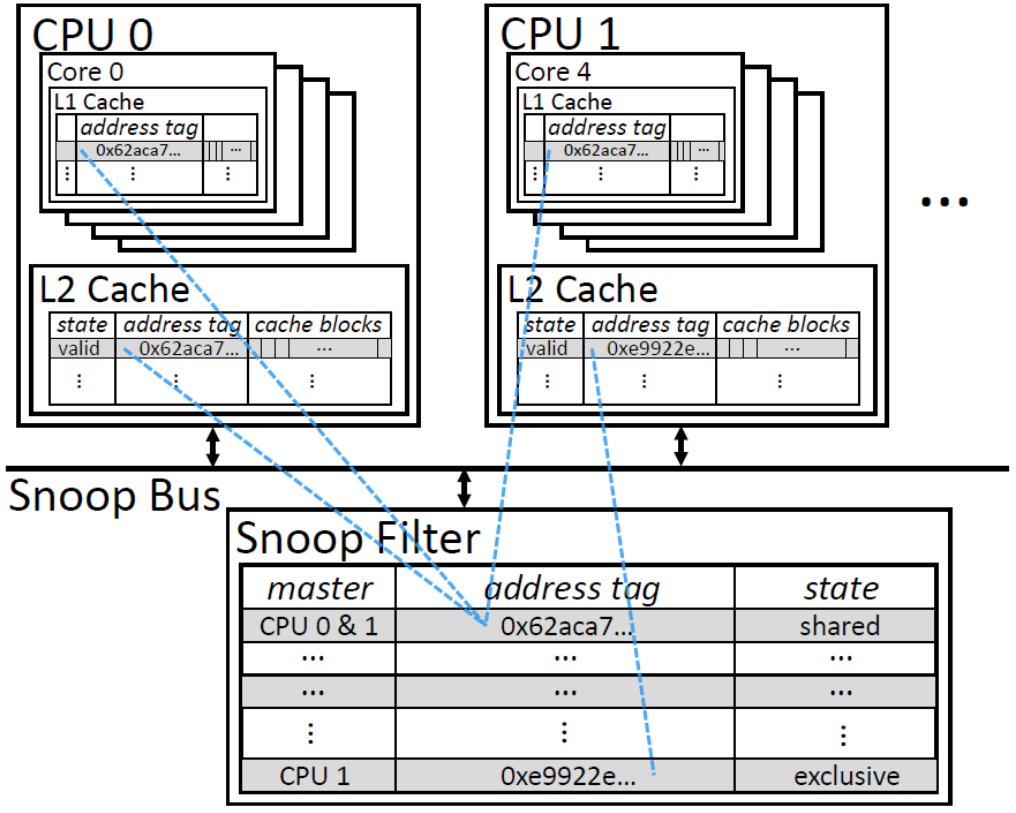


Fig. 3. General design of the snoop filter.

Experiment Platforms

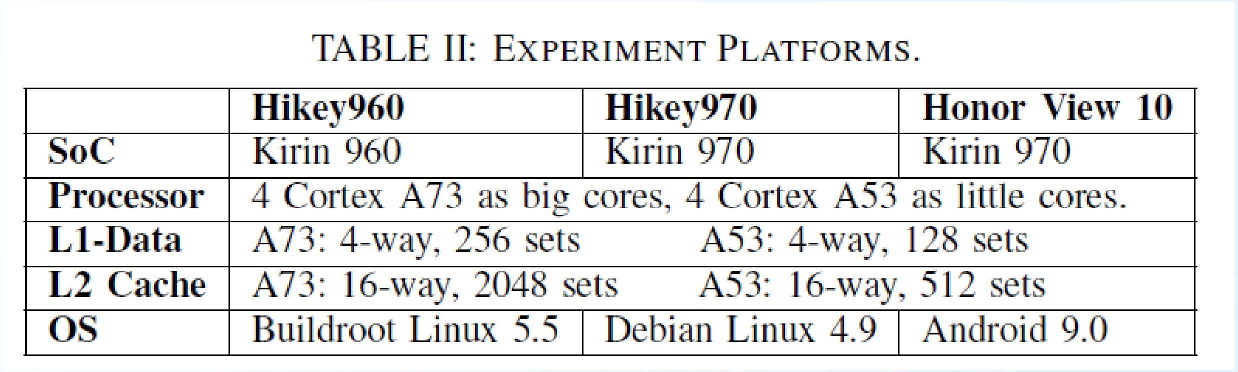


TABLE II: EXPERIMENT PLATFORMS.

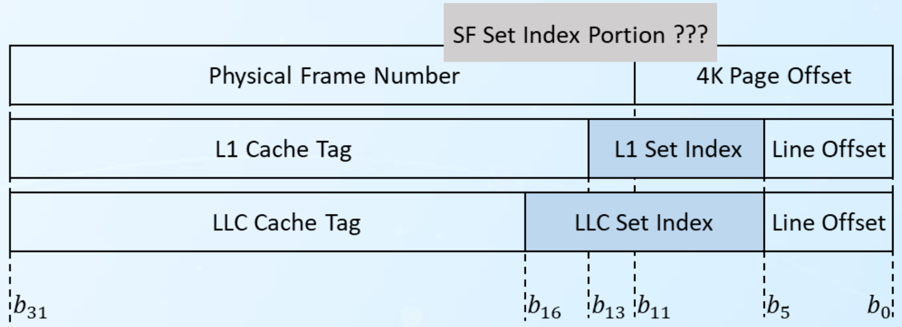
	Hikey960	**Hikey970**	**Honor View 10**
SoC	Kirin 960	Kirin 970	Kirin 970
Processor	4 Cortex A73 as big cores, 4 Cortex A53 as little cores.		
L1-Data	A73: 4-way, 256 sets	A53: 4-way, 128 sets	
L2 Cache	A73: 16-way, 2048 sets	A53: 16-way, 512 sets	
OS	Buildroot Linux 5.5	Debian Linux 4.9	Android 9.0

- Affected SoCs
  - ARM SoCs with CCI-500/550

Hints and Assumptions

- SF is not well documented and not yet explored!
  - We only got limited hints from official manuals
  - We must make “conservative” assumptions
- Assumptions
  - 8-way set-associative structure
  - May or may not be Bank-sliced structure
  - SF strictly contains all cached data, otherwise cannot keep the coherence
  - SF conflicts trigger “back-invalidate”

Find the first SF Eviction Set (EV)



Memory for a 4 GB address space from the aspect of the 4 KB page, L1 cache, and the L2 cache.

- Like cache EV, there exists the SF index portion
- Dedicated search algorithm
  - Allocate a set of data with the same lower-n bits of their address
  - Search out the first SF EV by **timing difference**

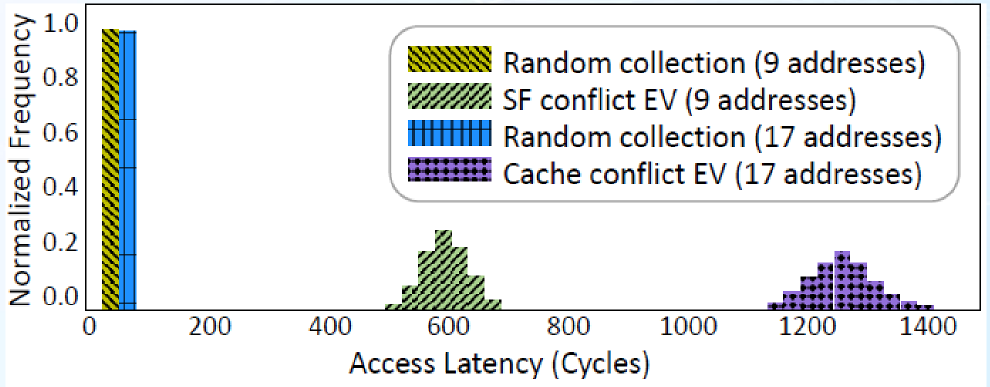


Fig. 3. Latencies of accessing a cache (or SF) conflict EV.

Reverse engineering

Set Associativity

- Test cache and SF EVs with different size

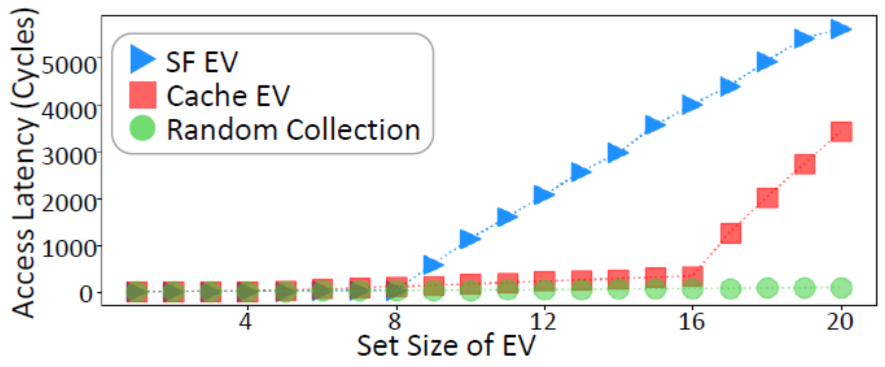


Fig. 4. Access latencies of EVs of different sizes. SF conflicts happen when the size of an SF EV is larger than 8.

- lines' slopes depict the capacity of an SF set and a cache set
- **Confirmed that SF is 8-way set associative**

Reverse engineering

Replacement Policy

- Possible replacement policy
  - Least Recently Used (LRU)
  - Random Replacement
  - …
- Place the data of a SF EV “in order”, observe which data would be selected to replace
  - Observed an even distribution of the 8 addresses
- **Most possibly Random Replacement**

Reverse engineering

Set Size

- how many mutually exclusive EVs are in a contiguous memory space?

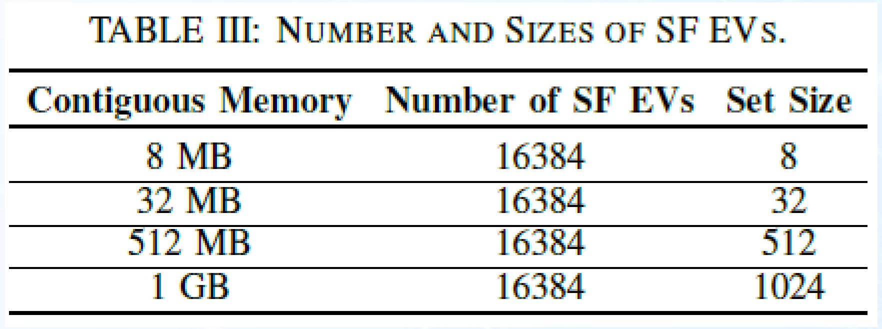


TABLE III: NUMBER AND SIZES OF SF EVS.

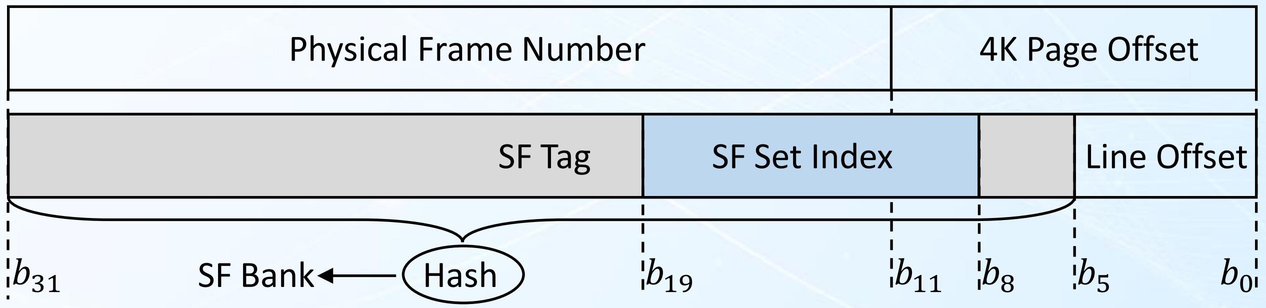
Contiguous Memory	Number of SF EVs	Set Size
8 MB	16384	8
32 MB	16384	32
512 MB	16384	512
1 GB	16384	1024

- **There are 16384 SF sets**

Reverse engineering

Set Index Portion

- The address portion used to index which SF set to map
- Check the identical bits of data in an SF EV



- **8th bit to 19th bit** (interesting that is doesn’t follow the line offset bits)

Reverse engineering

Bank Hash Function

- Is SF bank-sliced?
  - Set index portion has only 11 bits, while Set size is 16384
  - Indicate there are 8 SF banks
- Reverse engineered the bank hash function by dedicated algorithm

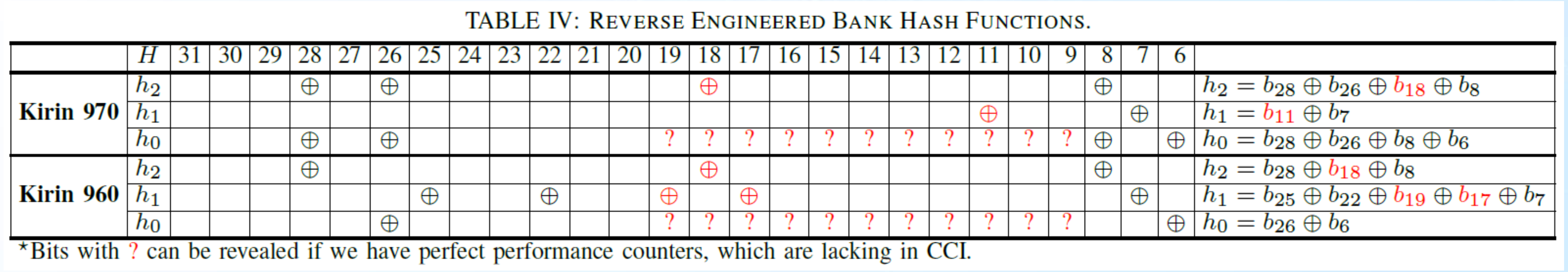


TABLE IV: REVERSE ENGINEERED BANK HASH FUNCTIONS.

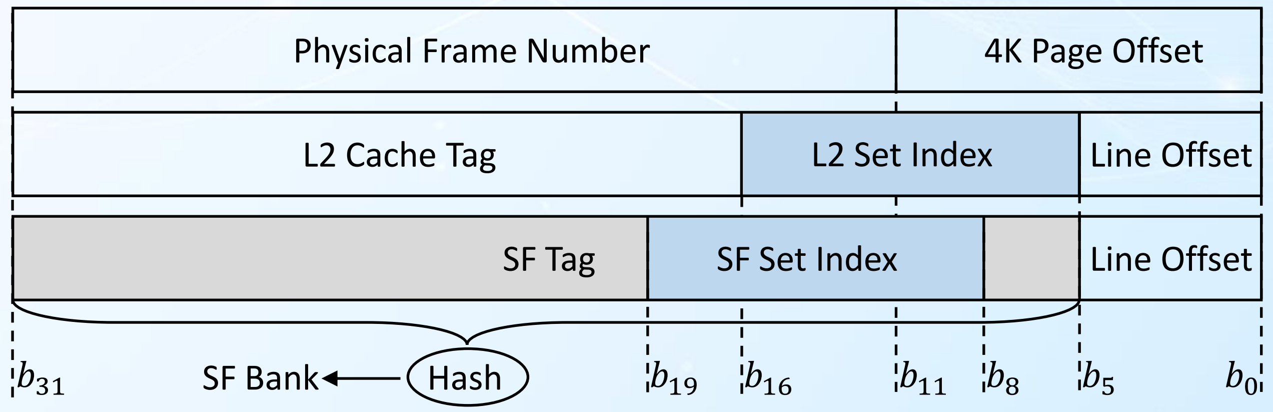
	H	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	
Kirin 970	h_2				$\oplus$		$\oplus$								$\oplus$										$\oplus$			$h_2 = b_{28} \oplus b_{26} \oplus b_{18} \oplus b_8$
	h_1																					$\oplus$				$\oplus$		$h_1 = b_{11} \oplus b_7$
	h_0				$\oplus$		$\oplus$							?	?	?	?	?	?	?	?	?	?	?	$\oplus$		$\oplus$	$h_0 = b_{28} \oplus b_{26} \oplus b_8 \oplus b_6$
Kirin 960	h_2				$\oplus$										$\oplus$										$\oplus$			$h_2 = b_{28} \oplus b_{18} \oplus b_8$
	h_1							$\oplus$			$\oplus$			$\oplus$		$\oplus$										$\oplus$		$h_1 = b_{25} \oplus b_{22} \oplus b_{19} \oplus b_{17} \oplus b_7$
	h_0						$\oplus$							?	?	?	?	?	?	?	?	?	?	?			$\oplus$	$h_0 = b_{26} \oplus b_6$

*Bits with ? can be revealed if we have perfect performance counters, which are lacking in CCI.

Reverse engineering

Summary

- 8-way set associative
- Random replacement policy
- 16384 SF sets
- Index mechanism (index portion and bank hash function)



Memory for a 4 GB address space from the aspect of the 4 KB page, L2 cache, and the SF.

Reverse engineering

Verify the attack surface of the SF

- SF-Prime+Probe

1. Randomly collect an address and construct its corresponding SF EV.
2. Access the address to cache it well and then access the SF EV
3. Check if the data is still cached.

Passed the verification in all three scenario:
Single-core, Cross-core, and Cross-cluster !

We introduce…

- Reverse engineering of the directory named Snoop Filter (SF)
  - Structure and Properties
  - **SF eviction set construction**

- Compare SF with cache in real attack applications
  - Covert channel
  - Attack Cryptographic algorithm
    - RSA
    - AES
  - TrustZone scenario

SF EV Construction

Construct SF EV in user space is not as easy as cache EV!

- Cannot directly deal with the SF, instead, we rely on caches to interact with the SF
- No way to distinguish SF conflicts from cache conflicts, as they both behave as cache misses
- SF conflicts appear much less often than cache conflicts

We propose,
avoid cache conflicts first, and then collect SF conflict samples

SF EV Construction

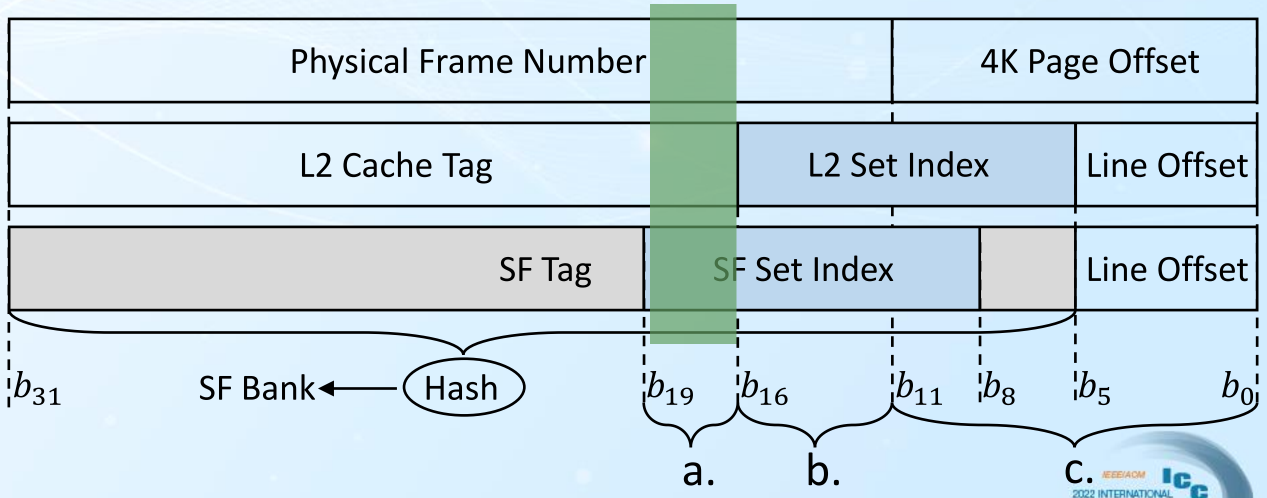
Probabilistic Approach to Construct SF EV

- In user space, attackers can only control the bits in *b.* and *c.*
  - By collecting a cache EV, we get a set of data with the same bits in *b.* and *c.*
- The bits in *a.* is out of attackers' control, they are "unknown bits"
  - The bits value follow a uniform distribution

$$P(unknown\ bits = \{00...0\}_2) = ...$$
$$... = P(unknown\ bits = \{11...1\}_2) = \frac{S}{16384}$$

- Collect *k* data to form an SF EV?
  - Possibility of a success collection

$$P(success\ collection)$$
$$= \sum_{i=9}^{k} P(success\ collection\text{: } i\ addresses\ case)$$
$$= \sum_{i=9}^{k} \frac{16384}{S} \binom{i}{k} \left(\frac{S}{16384}\right)^{i} \left(1 - \frac{S}{16384}\right)^{k-i}$$



SF EV Construction

Probabilistic Approach to Construct SF EV

- Feasible to construct an SF EV on a big core (A73)

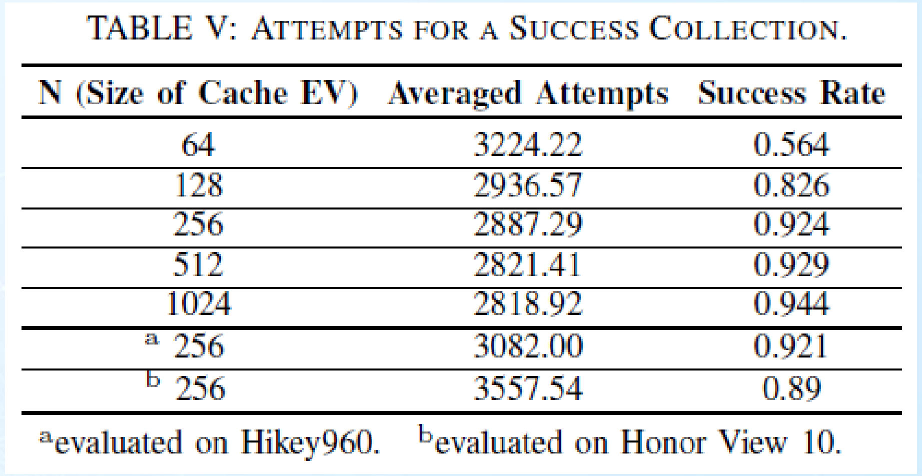


TABLE V: ATTEMPTS FOR A SUCCESS COLLECTION.

N (Size of Cache EV)	Averaged Attempts	Success Rate
64	3224.22	0.564
128	2936.57	0.826
256	2887.29	0.924
512	2821.41	0.929
1024	2818.92	0.944
[a] 256	3082.00	0.921
[b] 256	3557.54	0.89

[a]evaluated on Hikey960. [b]evaluated on Honor View 10.

- Infeasible to construct on a little core (A53)
  - The number of unknown bits increases ↑
  - The possibility of a success collection decreases ↓
- However,
  - Once constructed, it doesn’t matter attackers are on big or little cores
  - In covert channel communication, one can always construct SF EVs

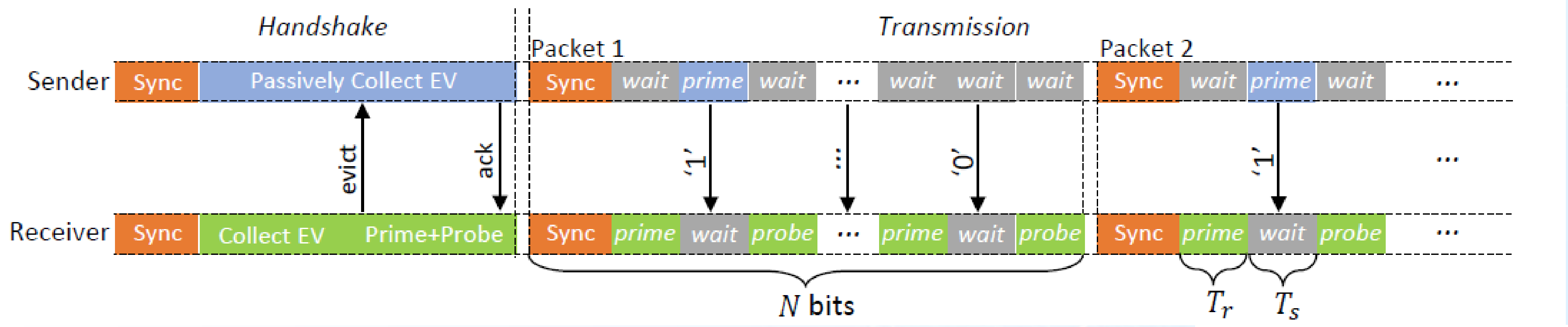
We introduce…

- Reverse engineering of the directory named Snoop Filter (SF)
  - Structure and Properties
  - SF eviction set construction
- Compare SF with cache in real attack applications
  - Covert channel
  - Attack Cryptographic algorithm
    - RSA
    - AES
  - TrustZone scenario

Side-channel attacks: SF vs Cache?

Covert channel communication

- Create stealthy communication via SF side-channel



- Results
  - Achieve the same level of performance as cache covert channel in cross-core scenario
  - In the cross-cluster scenario, the performance is still satisfying
  - We recommend SF covert channel!

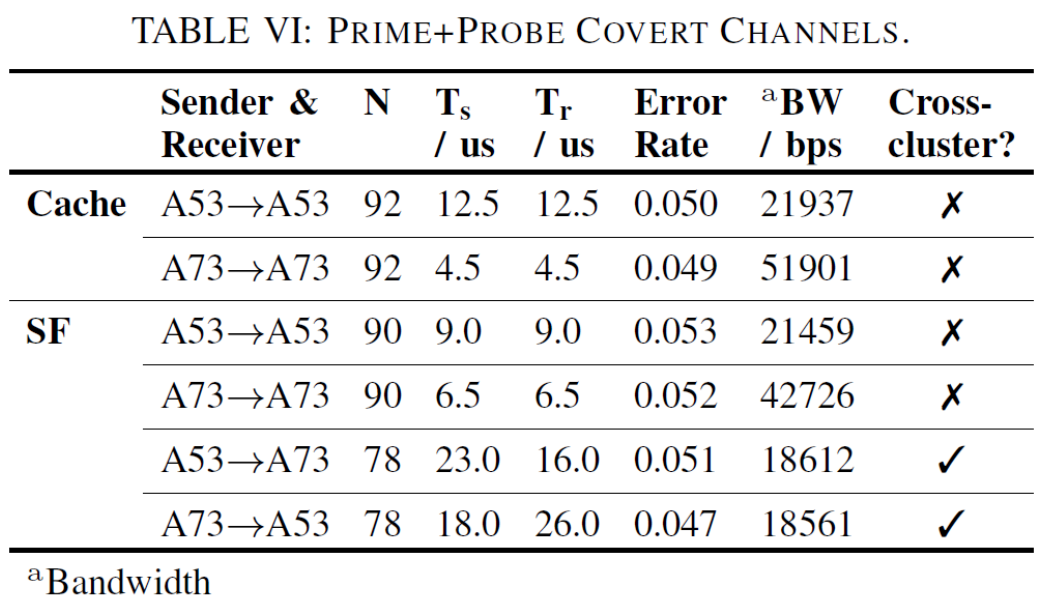


TABLE VI: PRIME+PROBE COVERT CHANNELS.

	Sender & Receiver	N	T_s / us	T_r / us	Error Rate	[a]BW / bps	Cross-cluster?
Cache	A53→A53	92	12.5	12.5	0.050	21937	✗
	A73→A73	92	4.5	4.5	0.049	51901	✗
SF	A53→A53	90	9.0	9.0	0.053	21459	✗
	A73→A73	90	6.5	6.5	0.052	42726	✗
	A53→A73	78	23.0	16.0	0.051	18612	✓
	A73→A53	78	18.0	26.0	0.047	18561	✓

[a]Bandwidth

Side-channel attacks: SF vs Cache?

128-bit T-table based AES decryption in OpenSSL 1.1.1a

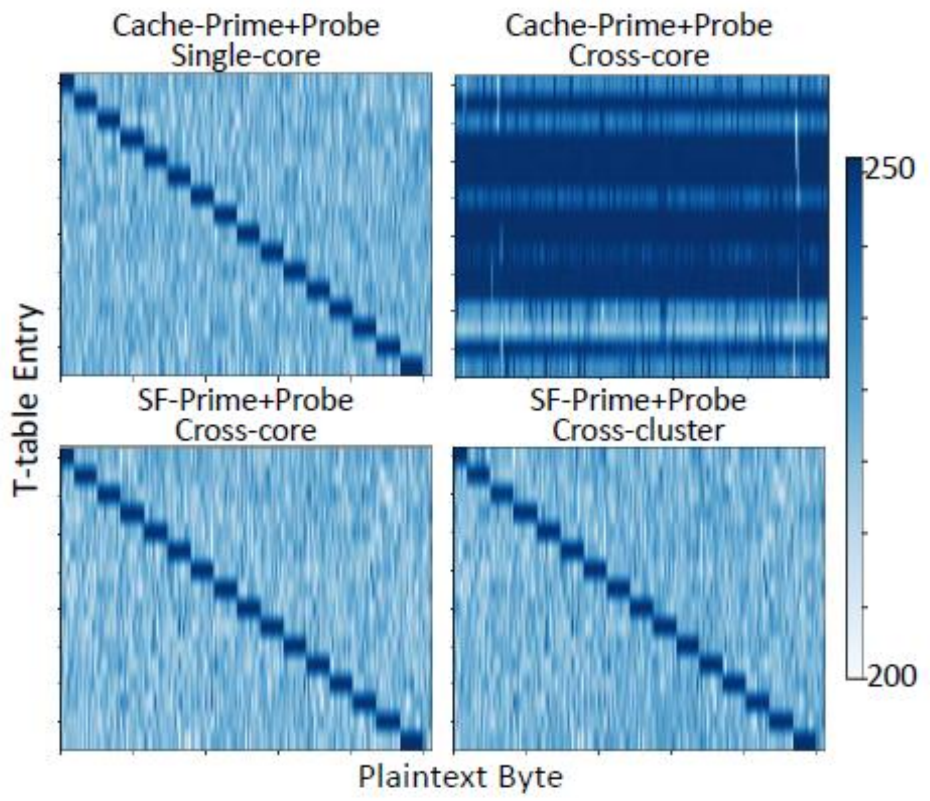


Fig. 7. Secret-dependent access patterns of T-table based AES.

First round attack

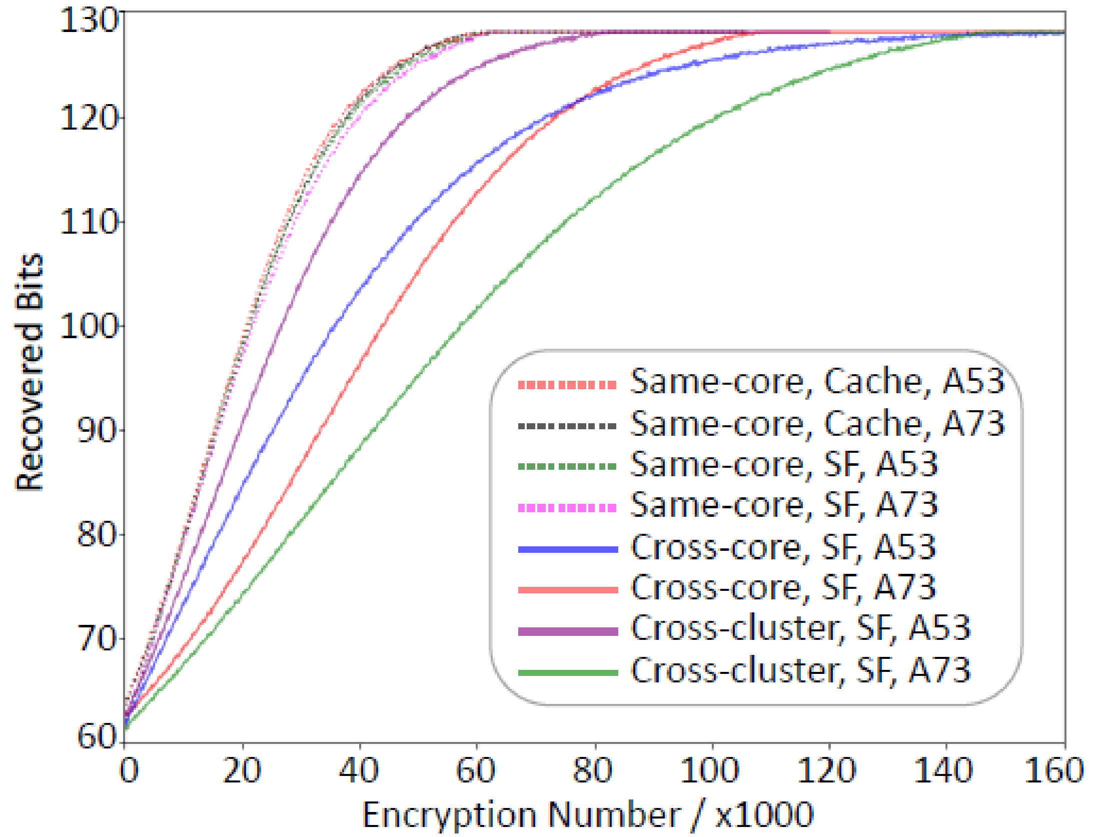


Fig. 8. Recovered bits increase with the encryption.

Last round attack

Side-channel attacks: SF vs Cache?

Sliding-window Based RSA in MbedTLS 2.26.0, window_size = 1

- Since RSA attack only happens in the cross-core / cross-cluster scenarios, we are the first to implement RSA side-channel attack on ARM CPUs.
  - 37 samples on average are sufficient to fully recover the private key

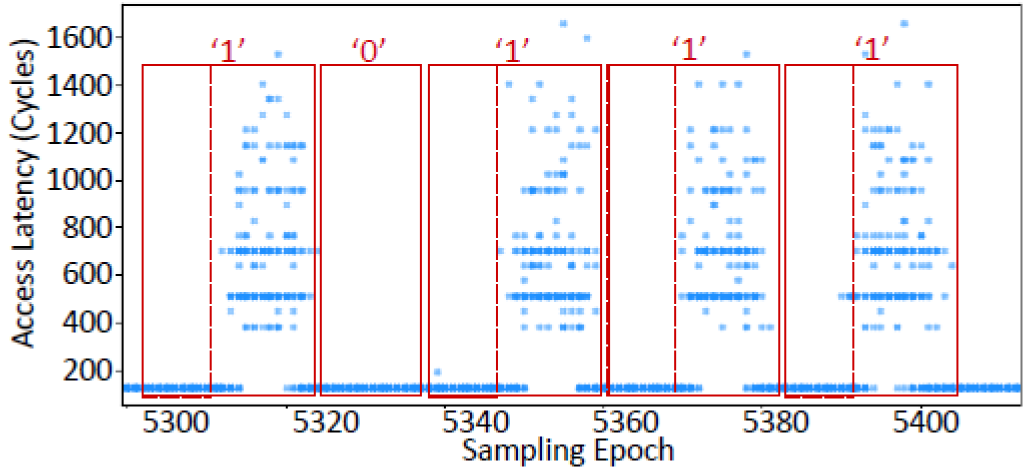


Fig. 9. Secret-dependent memory access of the sliding-window based RSA. From the access latency generated by SF-Prime+Probe, we can recover the key bits as '10111'.

Side-channel attacks: SF vs Cache?

Attack ARM TrustZone, Sliding-window Based RSA, window_size = 6

- Kernel privileged attackers in the TrustZone scenario is capable to conduct interrupt-based high-precision attacks [1]
  - Single trace profile is enough to recover the key
  - SF-Prime+Probe breaches
    - Exponent blinding defence
    - Strict cache clean defence

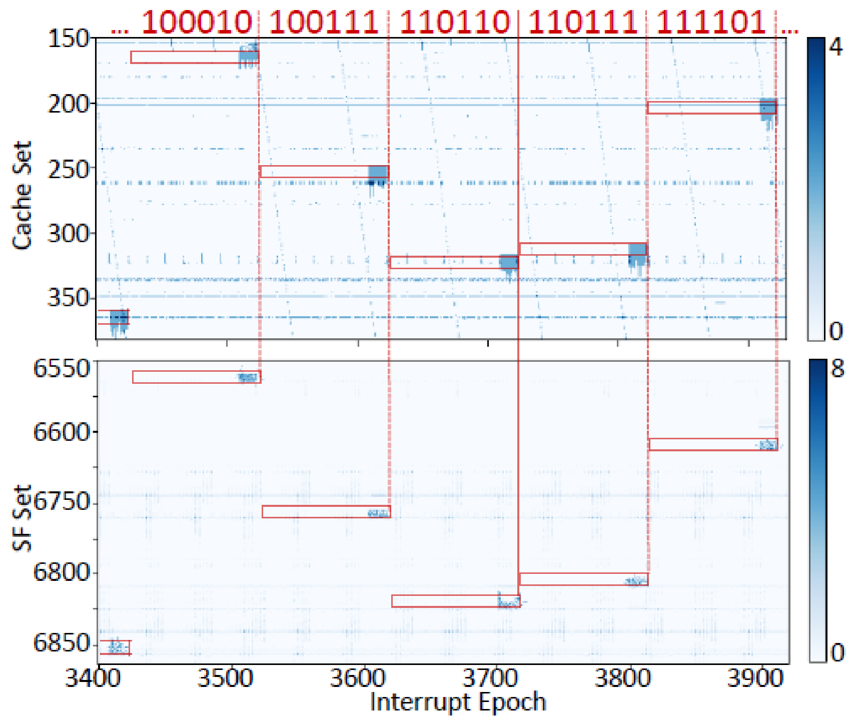


Fig. 10. Secret-dependent access pattern of the sliding-window based RSA. The upper one is generated by Cache-Prime+Probe in the single-core scenario. The lower one is generated by SF-Prime+Probe in the cross-cluster scenario when the strict cache clean defense is applied.

Attack directories on ARM big.LITTLE Processors!

- SF is more practical than cache in cross-core and cross-cluster scenario
- Satisfying performance
- Harder to defend, especially in TrustZone scenario

Thanks for listening!

Page 3:

[1] G. Irazoqui et al., “Cross processor cache attacks,” in ASIACCS, 2016.

[2] F. Yao et al., “Covert timing channels exploiting cache coherence hardware: Characterization and defense,” International Journal of Parallel Programming, 2019.

[3] Y. Yarom et al., “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in USENIX Sec., 2014.

[4] D. Gruss et al., “Flush+flush: a fast and stealthy cache attack,” in DIMVA, 2016.

[5] F. Liu et al., “Last-level cache side-channel attacks are practical,” in IEEE S&P, 2015.

Page 8:

[1] M. Lipp et al., “Armageddon: Cache attacks on mobile devices,” in USENIX Sec., 2016.

[2] N. Zhang et al., “Truspy: Cache side-channel information leakage from the secure world on ARM devices.” IACR Cryptol., 2016.

[3] X. Zhang et al., “Return-oriented flush-reload side channels on ARM and their implications for android devices,” in CCS, 2016.

[4] H. Lee et al., “Hardware-based flush+reload attack on Armv8 system via ACP,” in ICOIN, 2021.

[5] G. Haas et al., “itimed: Cache attacks on the apple a10 fusion soc,” IACR Cryptol., 2021.

[6] K. Ryan, “Hardware-backed heist: Extracting ECDSA keys from qualcomm’s trustzone,” in CCS, 2019.

[7] Z. Kou et al., “Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+evict,” in DAC, 2021.

[8] M. Green et al., “Autolock: Why cache attacks on ARM are harder than you think,” in USENIX Sec., 2017.

Page 31:

[1] Z. Kou et al., “Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+evict,” in DAC, 2021.

Reported by Zili KOU (zkou@connect.ust.hk)