# PiPoMonitor: Mitigating Cross-core Cache Attacks Using the Auto-Cuckoo Filter

Fengkai Yuan*,Kai Wang†, Rui Hou*‡, Xiaoxin Li*, Peinan Li*, Lutan Zhao*, Jiameng Ying*, Amro Awad§, Dan Meng*
*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
†Department of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
§Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, USA
*{yuanfengkai, hourui, lixiaoxin, lipeinan, zhaolutan, yingjiameng}@iie.ac.cn
†wk1220ym@163.com, §ajawad@ncsu.edu

***Abstract*—Cache side channel attacks obtain victim cache line access footprint to infer security-critical information. Among them, cross-core attacks exploiting the shared last level cache are more threatening as their simplicity to set up and high capacity. Stateful approaches of detection-based mitigation observe precise cache behaviors and protect specific cache lines that are suspected of being attacked. However, their recording structures incur large storage overhead and are vulnerable to reverse engineering attacks. Exploring the intrinsic non-determinate layout of a traditional Cuckoo filter, this paper proposes a space efficient Auto-Cuckoo filter to record access footprints, which succeed to decrease storage overhead and resist reverse engineering attacks at the same time. With Auto-Cuckoo filter, we propose PiPoMonitor to detect *Ping-Pong patterns* and prefetch specific cache line to interfere with adversaries' cache probes. Security analysis shows the PiPoMonitor can effectively mitigate cross-core attacks and the Auto-Cuckoo filter is immune to reverse engineering attacks. Evaluation results indicate PiPoMonitor has negligible impact on performance and the storage overhead is only 0.37%, an order of magnitude lower than previous stateful approaches.**

***Index Terms*—cache side channels, Cuckoo filters, detection-based mitigation**

## I. INTRODUCTION

Cache hierarchy is one of the most critical designs to improve performance in commercial processors. However, the execution footprints left on different cache levels can be observed by attackers to infer security-critical information. In particular, cross-core attacks on the last level cache (LLC) are dangerous and has been widely exploited due to its high bandwidth, low noise and low construction difficulty [1].

There are three orthogonal kinds of mitigation against cross-core attacks: ❶*Partition-based mechanisms* physically isolate cache resources to protect victim process from attackers in different domains; ❷*Randomization-based mechanisms* change the cache layout periodically to confuse attackers to infer secrets, which mitigate attacks in both intra- and inter- security domain; ❸*Detection-based mechanisms* monitor the cache behaviors to detect suspicious operations, providing the information when there may exist attacks. The first two mechanisms have been well studied. This paper focuses on the third mechanism.

Existing detection-based defenses fall into three categories. ① Machine Learning-based approaches identify malicious operations based on machine learning algorithms. The accuracy of this method largely depends on the test suites and it cannot figure out the root cause of the information leakage. ② Stateless approaches [2], [3], [4] observe the events of back-invalidations in private cache lines from LLC evictions. However, back-invalidations vastly exist in benign execution, resulting in high false positives and performance overhead. ③ Stateful approaches [5], [6] avoid high false positives by observing more precise cache behaviors. They introduce extra structure to record suspicious temporal correlated LLC-memory interactions (called *Ping-Pong patterns*) of the same cache line. However, recording long history to capture the suspicious access pattern results in large storage overhead. Even worse, the limited-size recording structure is vulnerable to reverse engineering attacks.

In this paper, we propose a novel stateful approach with low false positives, low storage overhead and resistance to reverse engineering attacks, named as PiPoMonitor. It captures *Ping-Pong pattern* by monitoring the traffic history between LLC and memory. Once detecting abnormal behaviors, the corresponding cache lines are prefetched to interfere the secrets inference. To minimize the storage overhead and resist reverse engineering attacks, we deploy cuckoo filter to record traffic since it record the short hash results of addresses instead of themselves. However, attackers can delete certain records in traditional Cuckoo filter to bypass detection, resulting in false negatives. Thus, we propose the Auto-Cuckoo filter, which autonomically deletes the record stored in the i-th relocated destination where i is a predefined threshold. The autonomic deletion exponentially increases the uncertainty of a record eviction, rendering reverse engineering attacks impractical.

This paper makes the following contributions:

1) We first propose PiPoMonitor to record traffic history between LLC and memory with space efficient hardware Cuckoo filter to detect *Ping-Pong patters* in cross-core cache side-channel attacks effectively.
2) Based on traditional Cuckoo filter, we implement an Auto-Cuckoo filter with autonomic deletion to prevent malicious bypass in PiPoMonitor aware attacks.
3) After detecting malicious behaviors, PiPoMonitor prefetches specific cache lines to confuse the attackers.

‡Corresponding Author: Rui Hou

978-3-9819263-5-4/DATE21/©2021 EDAA

## II. Background

### A. Ping-Pong Patterns

To infer secrets of victim through cross-core cache side-channel attacks, attackers iteratively probe the target secret-dependent cache lines, causing frequent migration back and forth between LLC and memory. These cache line access behaviors under attack are defined as *Ping-Pong patterns* [5], [6].

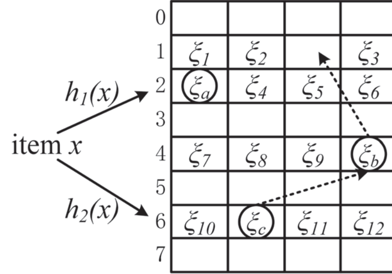



Fig. 1. Record relocations of the Cuckoo filter.

### B. Cuckoo Filter

A traditional Cuckoo filter [7] is a software data structure with three significant features:

**Space-efficiency:** It stores compact fingerprint records instead of the raw items. As shown in Fig 1, the filter first computes two candidate buckets indices for each item $x$ with following two hashing functions. Then, it records the hash results as the fingerprints, dubbed $\xi x$, to a matrix.

**Probabilistic false positives:** Since hash results of different items may be the same, a Cuckoo filter falsely responds that a match exists with a bounded possibility for its specifications.

**Non-determinate Layout:** For a high occupancy, Cuckoo filters relocate records to search for more vacancies. While inserting a new item, a filter allocates a entry in one of the two candidate bucket rows. If all entries in the two bucket rows have already been occupied, the filter then selects a record randomly to relocate it to its alternative bucket row. As Cuckoo filters only store fingerprints, it uses the partial-key Cuckoo hashing to compute bucket indices:

$$h_1(x) = hash(x)$$
$$h_2(x) = h_1(x) \oplus hash(\xi x)$$

Thus, the alternative bucket index of a relocated record can be computed by performing an XOR based on current bucket and the fingerprint. If the corresponding bucket rows are full, it has to select one more victim record and relocate it. Shown by Fig. 1, bucket row 6 is full while inserting item $x$. Then record $\xi c$ is selected to relocate to its alternative candidate bucket row 4. However, bucket row 4 is full as well, randomly selected $\xi b$ is relocated to bucket row 1 and find a vacancy. Cuckoo filters specify an upper bound of the number of relocations (maximal number of kicks, denoted as MNK) to indicate that the filter is full. When the filter is full, Cuckoo filters support deletion operations to remove some records manually.

TABLE I
Notations of the Auto-Cuckoo filter

| Notations | Description |
|---|---|
| $x$ | a cache line address |
| $\xi x$ | fingerprint of $x$ |
| $\mu x$, $\sigma x$ | two candidate buckets of $x$ |
| $l$ | number of buckets |
| $b$ | number of entries per bucket |
| $f$ | length of fingerprint in bits |
| $\varepsilon$ | false positive rate |
| *MNK* | the maximal number of kicks |
| *fPrint* | fingerprint field of a filter entry |
| *Security* | reAccess counter of a filter entry |
| *secThr* | the saturation value of *Security* |

## III. Threat Model

PiPoMonitor aims at mitigating cross-core last-level cache side channel attacks. We assume that adversaries are not privileged and do not share a physical core with the victim. Our current implementation focus on attacks on inclusive LLC. With PiPoMonitor guarding LLC-memory traffic, non-inclusive LLC can be slightly extended to defend against directory attacks [8].

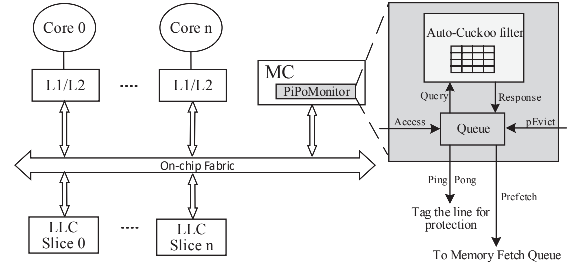



Fig. 2. Architecture of the proposed approach

## IV. Overview of PiPoMonitor

As shown in Figure 2, PiPoMonitor locates inside the on-chip memory controller (MC) and observes the memory access requests from LLC without extra network traffics. Deployed with a hardware Auto-Cuckoo filter based on the Cuckoo filter, PiPoMonitor records cache line re-accesses and identifies lines behaving in *Ping-Pong patterns* (Ping-Pong Lines). Note that the PiPoMonitor works in parallel with memory fetches. It helps to hide latency of Response-waiting and avoid a critical path. To demonstrate the Auto-Cuckoo filter conveniently, Table I summaries the notations and their descriptions.

**Capturing Ping-Pong lines.** The Auto-Cuckoo filter records the fingerprints of *access requests to main memory* (***Access***) in the buckets. We use ***reAccess*** to define the Access for the recorded line. For each recorded line, the field of *Security* to count the Access for the same line. For an Access $x$ Query, the Auto-Cuckoo filter first checks whether there is a valid entry of $\xi x$ in the buckets $\mu x$ or $\sigma x$. If there is no valid entry of $x$, the Auto-Cuckoo filter inserts a new entry in one of its candidate buckets, and its *Security* is initialized as zero and returned to Response. If the result is 'exist', the *Security* of the entry is increased by 1 to count reAccess and it is returned to Response.

When *Security* counts up to the pre-defined *secThr*, it satisfies the Ping-Pong pattern and $x$ will be captured as a Ping-Pong.

**Prefetching Ping-Pong lines.** Once a Ping-Pong line is captured, the cache line will be tagged as Ping-Pong in LLC when it is retrieved to MC from memory. When a tagged line is evicted, LLC will send a ***pEvict*** message to PiPoMonitor. After receiving the message, the latter waits for a pre-defined delay, and then sends a request to the memory fetch queue of MC to prefetch the line back to LLC, obfuscating adversaries' cache probes. The delay is to avoid memory bandwidth preemption with the writeback of the same line.

Noting that once a Ping-Pong line has undergone Prefetch, it is tagged and will record whether it has been accessed. Only when the tagged-accessed line is evicted, it will be prefetched, so as to avoid over-protection by endless prefetching.

## V. Implementation

### A. Autonomic Deletion

The deletion of a classic Cuckoo filter is vulnerable. Because of false positives, addresses controlled by an adversary may have the same fingerprint and candidate buckets as the target line. Adversaries can exploit false deletions to remove one fingerprint (record) by deleting another under control.

We introduce autonomic deletions to modify insertions by making them never fails. When the number of relocations reaches *MNK*, the Auto-Cuckoo filter evicts the last fingerprint that needs to be relocated. Recall that the fingerprint kicked out is randomly selected, and the alternative bucket for each fingerprint is different. The random kick paths inevitably lead to rich randomness of the fingerprint that is eventually evicted, which essentially increases the difficulty of constructing an eviction set by reverse attacks (proof in Section VI).

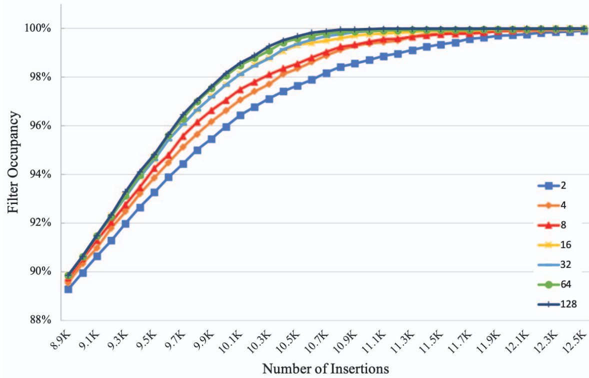



Fig. 3. The occupancy of the Auto-Cuckoo filter using different *MNK*

Besides, autonomic deletion substantially decreases the *MNK* to reach a high occupancy. *MNK* is set as 100∼1000 to avoid insertion failures in classic software Cuckoo filters. For hardware implementation, a large *MNK* means excessive hash computation and prohibitive relocation overhead. In response, the Auto-Cuckoo filter relies on historical insertions to continuously find vacancies such that the occupancy climbs to 100%, rather than counting on enormous relocations for each insertion. Figure 3 depicts the occupancy as the insertion number increases. The filter parameters follow the configuration in Table II. We randomly pick addresses from memory address space and insert them into the filter using different *MNK*. The results show the occupancy is not sensitive to *MNK*. When the insertion number is less than 9K, the occupancy is even identical. Even if *MNK* = 2, the occupancy can reach 100% when insertion number is 12.5K. We choose *MNK* = 4 considering the trade-off between performance and security (see Section VI).

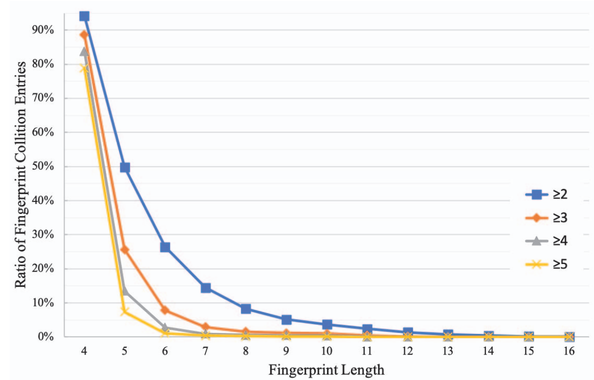



Fig. 4. The ratio of fingerprint collision entries in the $b$=8 Auto-Cuckoo filter with different $f$. The result is classified according to the number of addresses that have collided in the entries

### B. False Positive Rate

The false positive rate, $\varepsilon$, of the Auto-Cuckoo filter directly affects the detection precision. A false positive occurs when different addresses having the same fingerprint collide in their candidate buckets. When a fingerprint collision occurs during a Query, records are merged into one entry with *Security* increased. This accelerate the increase of *Security* to *secThr*, so that collision addresses are incorrectly captured as Ping-Pong.

Fortunately, the Auto-Cuckoo filter is superior to maintain a low $\varepsilon$ with a small storage overhead. When looking up a non-existent record, a query checks two candidate buckets each with $b$ entries. For each entry, the probability that $\xi x$ is matched with *fPrint* is $1/2^f$. After $2b$ comparisons, the upper bound of the total probability of a fingerprint collision is [7]:

$$\varepsilon = 1-(1-1/2^f)^{2b} \approx 2b/2^f$$

With $l$ and $b$ determined through security analysis in Section VI, the storage overhead of Auto-Cuckoo filter is **linear** with $f$. In contrast, above equation shows that $\varepsilon$ decreases **exponentially** as $f$ increases. This means the filter can maintain a low false positive rate with a small storage overhead. Fig. 4 shows after 6 million insertions, the ratios of entries with fingerprint collisions in the filter as $f$ increases. The ratio changes with $f$ approximately conforms to the equation. We choose $f$=12 making the ratio as low as 0.014 with $\varepsilon = 0.004$. Moreover, shown in Fig. 4, the ratio of the entries with more than 2 collision addresses approach 0 in this configuration.

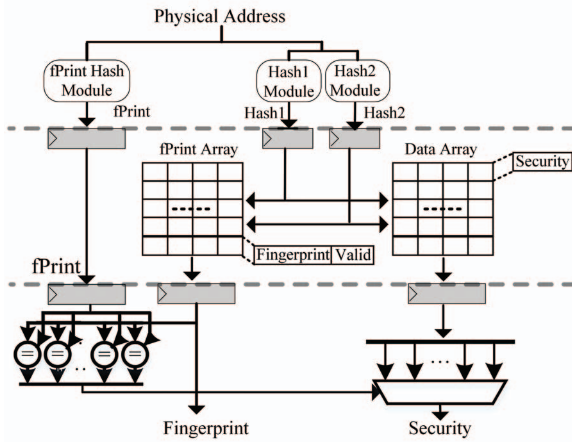



Fig. 5. Hardware Microarchitecture of the Auto-Cuckoo filter

### C. Hardware Microarchitecture

The microarchitecture of the Auto-Cuckoo filter is shown in Figure 5. Similar to cache, the filter is composed of *fPrint Array* and *data Array*. Correspondingly, each array has $l$ sets and each set has $b$ entries. Differently, each physical address has two candidate buckets to be recorded. Hash1 Module and Hash2 Module are deployed to generate the two indices hashed from physical address. Each entry can be inserted to any candidate set as long as it has a vacancy. It should be noticed that *Hash1*, *Hash2* and *fPrint Hash* satisfy the relationship described in Section II to guarantee that a candidate bucket index can deduce the other candidate bucket index with the result of *fPrint Hash*.

**fPrint Array** consists of two fields: The 1-bit Valid flag indicates whether the corresponding entry is meaningful. The $f$-bit fPrint records the hashed value from physical address of each entry during the insertion.

**Data Array** consists of one saturation counter: Ping-Pong is shaped when *Security* reaches the threshold (secThr = 3). The Auto-Cuckoo filter response the Query of PiPoMonitor by Response this value. A Response of *secThr* informs the latter the Access line behaves in Ping-Pong pattern.

## VI. Security Analysis

### A. Effectiveness on Ping-Pong Capturing

We launched Prime+Probe [9] attacks, trying to steal the key from a Square-and-Multiply algorithm (GnuPG version 1.4.13). The algorithm processes the key iteratively from high to low bits, one bit in each iteration. If the bit is 1, *square* and *multiply* performed; otherwise, only *multiply* performed. The sequence of above operations indirectly expose the key.

The attacker and victim run on different physical cores. Every 5000 cycles, the attacker probes two target addresses: the entry addresses of the *square* and *multiple*. In each attack iteration, the attacker accesses eviction sets to evict target addresses from LLC and re-accesses them. When a target address is accessed by victim, an address of the eviction set will be evicted from LLC, causing attacker's re-access having a large delay due to cache miss; otherwise, all re-accesses hit.

Figure 6 shows the probe results of 100 attack iterations with and without PiPoMonitor. Each blue dot represents that the attacker observed a large delay and inferred the victim may have accessed the target addresses. In Figure 6(a), the attacker obtained the victim's operation sequence of *square* and *multiply*. When deploying PiPoMonitor, the attacked addresses are recognized as Ping-Pong lines and be protected by Prefetch. In Figure 6(b), no matter whether the victim has accessed, the attacker always observes accesses, so that the genuine operation sequence cannot be obtained.

### B. Defeating Defense-Aware Adversary

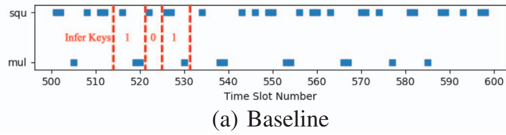
PiPoMonitor relies on the Auto-Cuckoo filter to record memory access traffic. However, due to hardware limitation, the filter cannot record all lines simultaneously. Not surprisingly, during each attack iteration, adversaries can try to deterministically evict the target record from the filter before the victim's re-accesses shape it into a Ping-Pong.

**Brute force.** An straightforward manner is to use enormous new addresses to fill the Auto-Cuckoo filter, causing conflicts and evicting the target record. We assume each fill of the adversary can evict one stored record in the filter without the interference of fingerprint collisions. Due to the randomness introduced by autonomous deletion, the probability that the attacker can evict the target record with each fill is:

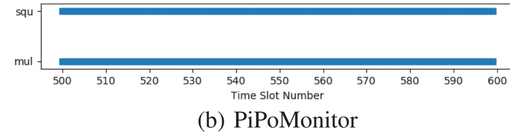
$$P(evict) = 1/(b \times l)$$

Therefore, the mathematical expectation of the fills required to evict a target record is $b * l$. Our experimental results has supported this conclusion, when $b$=8, $l$=1024, we found the adversary needed 8192 memory accesses on average to evict the target record. This severely slowed down the attack, and the time consumed exceeds the iteration interval limit of existing attacks to probe victims' cache accesses.

**Reverse engineering attacks.** The more effective method is to construct an eviction set and fill the filter with a small number of addresses. Unfortunately, the adversary have to deal with the **exponential** impact of *MNK* on eviction uncertainty. Shown by Fig. 7, the target record $T$ is located in the bottom bucket with $b$ entries. When $MNK = 0$, the adversary only needs to fill $b$ records $\{A, B...\}$ as an eviction set to evict $T$. Although the filter randomly selects victim to evict in a bucket, the adversary can continue to fill the set and evict $T$ in a linear time. When $MNK = 1$, only when $\{A, B...\}$ arrives at the bottom bucket after 1 relocation can probably trigger the eviction of $T$. The attacker needs to fill $b$ eviction sets $\{\{A_1, A_2...A_b\}, \{B_1, B_2...B_b\}...\}$ to relocate $\{A, B...\}$ from the middle buckets to the bottom to continuously triggers eviction and evicts $T$ eventually. By analogy, when $MNK = 2$, shown by Fig. 7, the size of the overall eviction set reaches $b^3$. The Auto-Cuckoo filter is configured as $b = 8, MNK = 4$ in this design, making the eviction set size reaching $b^{(MNK+1)} = 32768$. This makes the time cost of the reverse attack even exceed the brute force, rendering it impractical.



(a) Baseline



(b) PiPoMonitor

Fig. 6. Cache usage patterns of probe addresses extracted by the attacker.

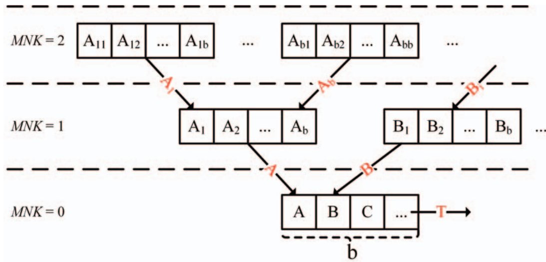


Fig. 7. The reverse attacks on the Auto-Cuckoo filter with different *MNK*

### C. False Negatives

Autonomous deletion facilitates the Auto-Cuckoo filter to build a dynamic observation window, but it inevitably introduces false negatives. In fact, the observed address space is extremely larger than the filter itself. It's still competent because the defense goal is to guarantee that within the window, the attacker's LLC evictions and victim's re-accesses will shape the target record into Ping-Pong. A reasonable $b \times l$ can generally ensure the time required for the record to become Ping-Pong, and sporadic false negatives cannot hinder the eventual capture when the subsequent Access arrives. Most importantly, autonomous deletion makes it impossible to deterministically evict a target record and create false negatives of actual threats.

## VII. Evaluations

### A. Methodology

We implemented PiPoMonitor on Gem5 [10], a cycle-accurate simulator. The baseline is configured as a quad-core processor with an inclusive cache architecture. Each processor core contains private L1 and L2 cache. The shared L3 cache is physically distributed as slices. The PiPoMonitor is deployed in the memory controller, and mainly consists of the Auto-Cuckoo filter. The detailed parameters are listed in Table II.

TABLE II
System Configurations

| Baseline Parameters | |
|---|---|
| CPU | 4 cores at 2.0 GHz |
| Coherence protocol | MESI |
| L1I/L1D caches | Inclusive, Private, 64 KB, 4-way, 2 cycles |
| L2 caches | Inclusive, Private, 256 KB/core, 8-way, 18 cycles |
| L3 caches | Inclusive, Shared, 4 MB, 16-way, 35 cycles |
| DRAM | 200-cycle latency |
| **PiPoMonitor Parameters** | |
| Auto-Cuckoo filter | $l$=1024, $b$=8, $f$=12, $\varepsilon$=0.004, *secThr*=3, *MNK*=4 |

As shown in the table III, we composed 10 workloads mixed by benchmarks from the SPEC CPU2006. Each workload contains 4 benchmarks with the reference input size. The benchmarks run concurrently on the quad-core processor. The baseline system runs the same workloads without PiPoMonitor. For each benchmark, we simulate 1 billion instructions in its core stage and compare the overall execution time.

TABLE III
Workloads used in our evaluation.

| MIX | Components |
|---|---|
| mix1 | libquantum-mcf-sphinx3-gobmk |
| mix2 | sphinx3-libquantum-bzip2-sjeng |
| mix3 | gobmk-bzip2-hmmer-sjeng |
| mix4 | libquantum-sjeng-calculix-h264ref |
| mix5 | astar-libquantum-mcf-calculix |
| mix6 | astar-mcf-gromacs-h264ref |
| mix7 | gcc-milc-gobmk-calculix |
| mix8 | gcc-mcf-gromacs-astar |
| mix9 | h264ref-astar-sjeng-gcc |
| mix10 | gromacs-gobmk-gcc-hmmer |

### B. Performance Evaluation

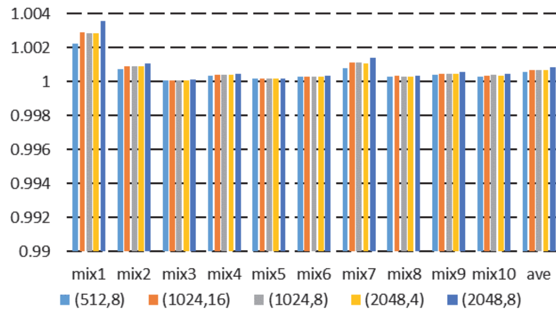
Fig. 8 (a) demonstrates the normalized performance of each workload to the baseline (so higher is better). Taking $l$ = 1024, $b$ = 8 as an example, the performance is improved by 0.1% on average. Among them, mix3, mix5 and other test sets have almost no performance changes, and the performance of mix1 has improved the most, reaching 0.3%.

The impact of PiPoMonitor on performance is caused by false positives. In our experiments, all cache lines having a Ping-Pong behavior and triggering Prefetch are considered as false positives. Fig. 8 (b) shows the number of false positives per million instructions in each mix. We can conclude that Prefetch these Ping-Pong cache lines is usually a benefit for performance. For example, when $l$=1024 and $b$=8, there are more false positives in mix1 and mix7, which are respectively 97 and 71 per million instructions. Their performance has also improved the most. On the other hand, the false positives of mix3 and mix6 are less than 20 per million instructions, and their performance is almost unchanged.

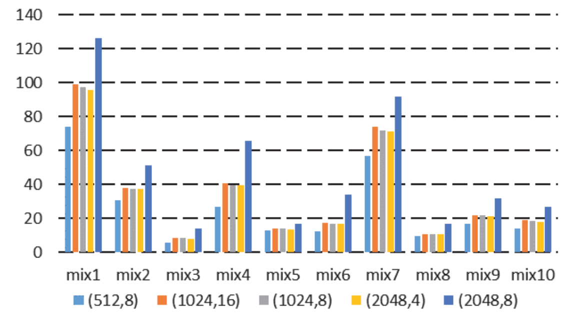
### C. Sensitivity Analysis

**Size of the Auto-Cuckoo filter.** Security analysis has proved that a larger Auto-Cuckoo filter means greater security. At the same time, a redundant filter means that more evicted cache lines can be recorded simultaneously, which may cause concerns about the impact of more false positives on performance. As shown in Figure 8 (b), we evaluated the Auto-Cuckoo filters configured with different size ($512 \times 8$, $1024 \times 8$, $1024 \times 16$, $2048 \times 4$, $2048 \times 8$) and found that the average impact on performance is less than 0.2%. Therefore, we suggest that the size of the Auto-Cuckoo filter should be considered from the perspective of storage overhead and security.

**Security Threshold (*secThr*).** The ***secThr*** affects the sensitivity of PiPoMonitor. When a line's Ping-Pong traffic exceeds




(a) Normalized performance




(b) Number of false positives per million instructions

Fig. 8. Performance evaluation with different Auto-Cuckoo filter sizes $(l, b)$.

this value, PiPoMonitor considers it suspicious and protects the line. Obviously, the smaller the value, the better it is for identifying abnormal behavior, but it may also generate more false positives. Experimental results affected by different *secThr* have confirmed this: the average performance when the threshold is 3 is better than when it is 1 or 2.

### D. Hardware Overhead

We evaluated the storage and area overhead of PiPoMonitor according to the configuration in table II. The calculation was performed using CACTI 7 [11] under the 22 nm technology. The Auto-Cuckoo filter contains 1024 × 8 = 8192 entries. Each filter entry has an *fPrint* field (12 bits), a *Security* counter (2 bits), and a valid (1 bit). Therefore, PiPoMonitor requires additional 15 KB storage overhead compared to the last-level cache, which is 0.37%. It occupies 0.013 $mm^2$, which is 0.32% more than the LLC. In addition, for a high-performance chip with more cores and larger LLC, the overhead could be further decrease.

## VIII. Related Work

Back-invalidation is a common feature for cross-core attacks. SHARP [3] modifies the cache replacement policy to randomly evict cache-lines while detecting the back-invalidations. BITP [4] employs a hardware prefetcher to prefetch the back-invalidation lines to interfere the observation of attackers. RIC [2] propose to relax the inclusion for read-only data and thread-private data, which prevent the back-invalidation of these data from other cores. However, there are large false positive ratios as back-invalidation behaviors happens in normal execution. Wang et al. proposed stateful approaches [5], [6] that extended directories to record cache line coherence events and identified more precise Ping-Pong patterns among cache-memory traffic. However, the storage overhead of the directory extension is essential and the directory itself is vulnerable to reverse attacks using eviction sets to evict target records. In contrast, PiPoMonitor addresses the challenge of a stateful approach by using space-efficient probabilistic hardware, the Auto-Cuckoo filter, to reduce storage overhead and defeat reverse engineering attacks.

## IX. Conclusion

This paper proposes PiPoMonitor, a novel stateful approach to defend against cross-core cache side channel attacks. It uses space-efficient probabilistic hardware, the Auto-Cuckoo filter to reduce storage overhead, and enhances layout non-determination against reverse engineering attacks. Evaluations and security analysis show PiPoMonitor achieves the defense goal with minimal performance and hardware overhead.

## X. Acknowledgment

This work was supported in part by the Strategic Priority Research Program of Chinese Academy of Sciences under grant No.XDC02010000, and in part by the National Scientific Research Project of China under grant No.2019603-001.

## References


[1] D. Ponom M. Kayaalp, N. Abu-Ghazaleh and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd DAC*, pages 72:1–72:6. ACM, 2016.

[2] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. RIC: relaxed inclusion caches for mitigating LLC side-channel attacks. In *Proceedings of the 54th DAC*, pages 7:1–7:6. ACM, 2017.

[3] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): defending against cache-based side channel atacks. In *Proceedings of the 44th ISCA*, pages 347–360. ACM, 2017.

[4] B. Panda. Fooling the sense of cross-core last-level cache eviction based attacker by prefetching common sense. In *28th International Conference on PACT*, pages 138–150. IEEE, 2019.

[5] K. Wang, F. Yuan, R. Hou, Z. Ji, and D. Meng. Capturing and obscuring ping-pong patterns to mitigate continuous attacks. In *DATE 2020*, pages 1408–1413. IEEE, 2020.

[6] K. Wang, F. Yuan, R. Hou, J. Lin, Z. Ji, and D. Meng. Cacheguard: a security-enhanced directory architecture against continuous attacks. In *Proceedings of the 16th CF*, pages 32–41. ACM, 2019.

[7] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[8] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. H. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on S&P*, pages 888–904. IEEE, 2019.

[9] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on SP*, pages 605–622. IEEE Computer Society, 2015.

[10] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, and et al. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[11] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *TACO*, 14(2):14:1–14:25, 2017.