Evaluating Cache Security Using Cache FX

Abhinav Kumar Singh
Avinash Saini

1 Objective

This project aims to measure how secure the **'Fully-Associative'** cache is against various eviction set-building techniques like: **'Single Holdout Method(SHM)'**, **'Group Elimination Method(GEM)'** and **'Prime+Prune+Probe(PPP)'**.

Since eviction set building is a part of contention-based attacks, we can also call it a measure of how secure the cache is against a **'Contention-Based Attack'**.

2 Terminology

For this project, we use an Associative Cache, a typical tag-block cache with a **random replacement policy**.

Eviction Set is a set of addresses which fills up the cache. In other words, if we start accessing the elements of the eviction set, the cache will contain addresses only from the eviction set at the end of this process. Any address which was previously in the cache is evicted. We can always create an infinitely large eviction set for any type of cache with any replacement policy. However, working on such a set can be time-consuming and will contain self-evictions lowering the attack efficiency. So here we try to generate a minimal eviction set.

Contention-based attacks exploit the fact that the cache sizes are limited. The attacker first evicts the contents of the cache by filling it with addresses from the eviction set and, after letting the victim have some time to execute, measures the time to access the cached data. Slow access indicates that the data is no longer cached, suggesting eviction from a cache set due to victim activity.

3 Eviction set-creating algorithms

3.1 SHM

We start with the set of **lines** and an **empty conflict set**. We pick a line randomly, and for each **line** in the set, we first read the elements of our **conflict set** and then, try to read the **line**. We add it to the **conflict set** if it is in the cache. We iterate through all the **lines** doing this.

After successfully generating the **conflict set**, we iterate through the set **"lines - conflict set"**. For each element in this set, if the **conflict set** evicts the element, we find the line in the conflict set, which after removing, no longer evicts the element. If such a **line** exists, we push it into our eviction set.

3.2 GEM

It is pretty similar to SHM; the only difference is that instead of working on lines, we work on **groups of lines** and then follow the algorithm.

3.3 PPP

We pre-fill the cache with the set of **candidate addresses** and then trigger the **victim addresses** of interest. Any cache misses in the candidate set facilitate locating any conflicting addresses.

4 Method

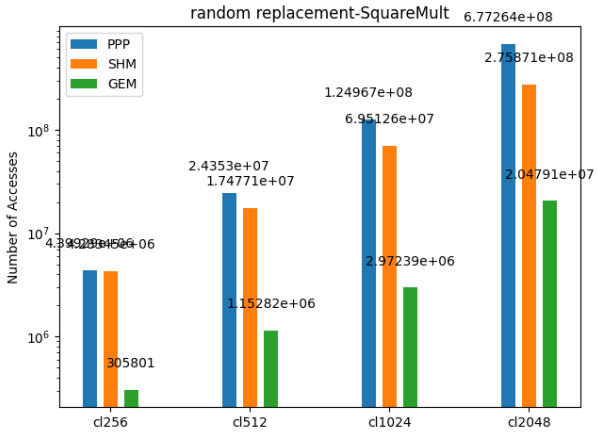
We use **Cache FX** to run simulations on our cache, a framework for evaluating the security of cache designs. The framework allows us to generate cache designs(in .xml format), and simulations can be run on them. It tracks all the hits and misses by both attackers and victims, the size of the eviction set generated, conflicting addresses etc. and gives us a complete simulation report.
Cache FX iterates until we find the minimum number of addresses required for an eviction set or until we reach a predefined maximum iteration count. We never exceeded the maximum count in our simulations, and an eviction set was always generated.

We mainly focus on the number of accesses required by the attacker to generate an eviction set.

The framework poses a few limitations as the environment is noise-free and unrealistic. Furthermore, we give the attacker way too much power, and the attacker is synchronized with the victim, which is highly unlikely in a real-world scenario. Also, the attacker is aware of the victim's address space which in a real-world scenario is hard to attain. There might even be a situation where an attacker might get hold of a few address lines, but it may go out of scope before even extracting data. So it can be considered as a lower bound for evaluating cache security. Given the best-case scenario, how easy or hard is it to create an eviction set for a cache and, subsequently, extract targetted data?

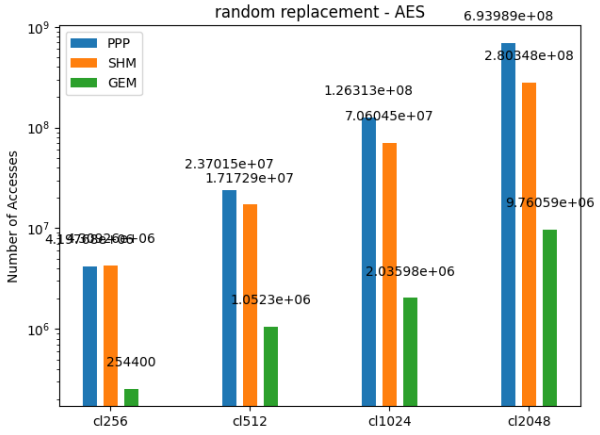
5 Observations

So, for simulations, we have used two different victim programs. The graph for the program "Square-Mult" is as shown.



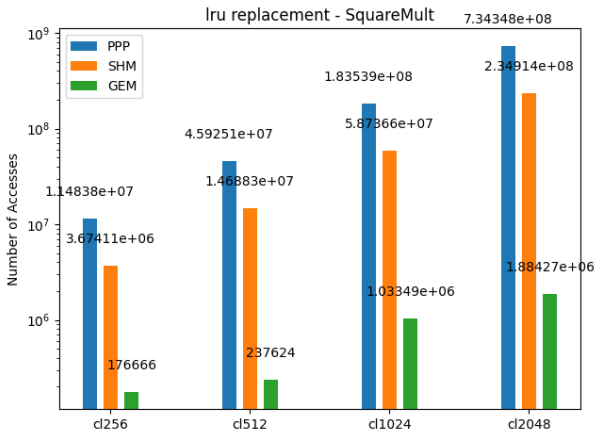
As expected, the number of accesses required to create an eviction set increases with size, with the difference being almost ten times each time we double the size of our fully-associative cache. That keeping aside, we also see that the trend among our eviction set-creating algorithms is the same in all our cache sizes, with PPP having the most considerable number of accesses, followed by SHM and finally GEM, having the least number of accesses.

Further moving, we have the graph of the other victim program, "AES".



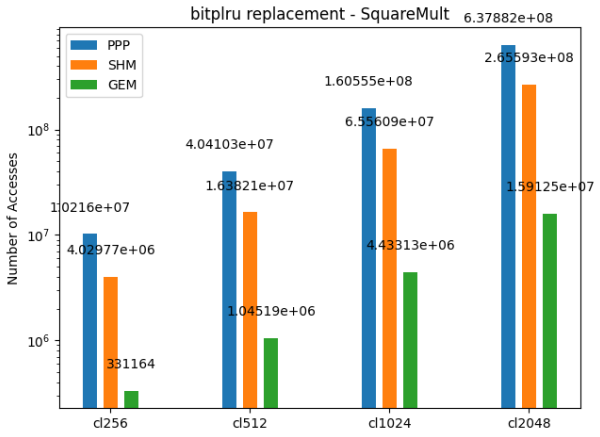
There is not much difference between the two. The eviction set creation algorithms follow the same trend and increase with the cache sizes. So, it becomes evident that the victim program has very little to do with how hard it is to create an eviction set. Though there are slight variations because of a change in the nature of the victim program, the order of the number of accesses remains the same.

Moreover, we see similar results by changing the replacement policy from random to lru.

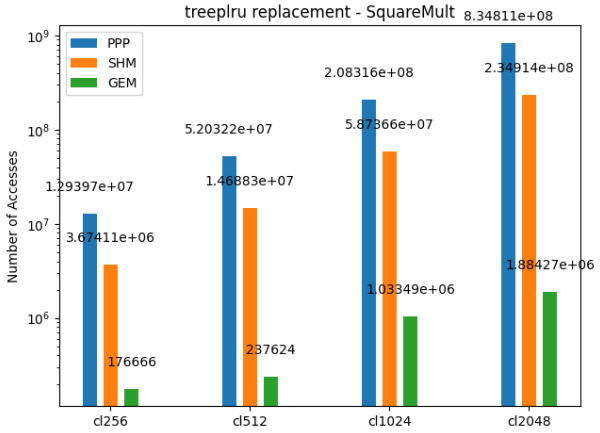


A few things to note here, though:

- The results for SHM remain almost the same.
- The lru replacement policies show significantly better endurance against PPP for smaller cache sizes. On the contrary, lru performs poorly against GEM.



The only notable difference for the replacement policy as bitplru is that it is more secure against PPP in the case of caches with smaller sizes.



Using treeplru as our replacement policy, we find the same results as lru. We have better results against PPP, but it is at a significant loss against GEM.

6 Results

So, based on our findings, we come to the following results:

- GEM performs exceptionally well than other eviction set-building strategies against associative cache with multiple replacement policies.
- The random policy is, after all, not so random. As mentioned in our first presentation, computers are deterministic machines, and we can always predict them. The fact that we can create an eviction set against this replacement policy verifies it.
- The eviction set size is equal to the cache size for obvious reasons. In an ideal case, there might also be self-evictions in the cache while creating our eviction set, making the cache more secure. Increasing the size of the eviction set increases the probability of cache conflicts between elements of the eviction set. These self-evictions introduce noise that increases the samples the attacker needs to observe to locate our victim addresses. Furthermore, larger eviction sets require more memory.

- Though increasing size may be one way to evade our problem, at the same time, we may end up giving up the sole purpose of caches to achieve that. Increasing cache size will increase lookup time, and system performance will degrade.

7 Links

- Github