Evaluating Cache Security using Cache FX

Abhinav Kumar Singh
Avinash Saini

Aim

- 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)**’.
- How secure the cache is against a ‘**Contention-Based Attack’**?

Terminology

- **Associative Cache** with **Random Replacement policy**
- **Eviction Set**
  - Fills up the cache
  - 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.
  - If we only want to evict the contents of cache, we can always create an infinitely large eviction set for <u>any type of cache, with any replacement policy</u>(**conflict set**). However, working on such a set can be time consuming.
  - Minimal eviction set.

Terminology

- **Contention-based attacks**
  - Cache sizes are limited.
  - The attacker first evicts the contents of the cache by filling it with addresses from the eviction set and, lets the victim have some time to execute.
  - Measures the time to access the contents of eviction set. Slow access indicates that the data is no longer cached, suggesting eviction from a cache set due to victim activity.
- Eviction Set creation is a **major step in Contention-based attacks** since we are able to manipulate the contents of cache and once the cache is filled with our eviction set, we can extract the targeted data.

Terminology(Algorithms)

For eviction set creation, we use the following algorithms:

- **Single Holdout Method**
  - start with the set of **lines** and an empty **conflict set**
  - for each **line** in the set, we first read the **elements** of our conflict set and then, try to read the **line**. Add the **line** to the **conflict set** if it is <u>not evicted</u>. 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 try to find a **line** in the **conflict set**, which after removing from the **conflict set**, it no longer evicts the **element**. If such a **line** exists, we push it into our **eviction set**.

Terminology(Algorithms)

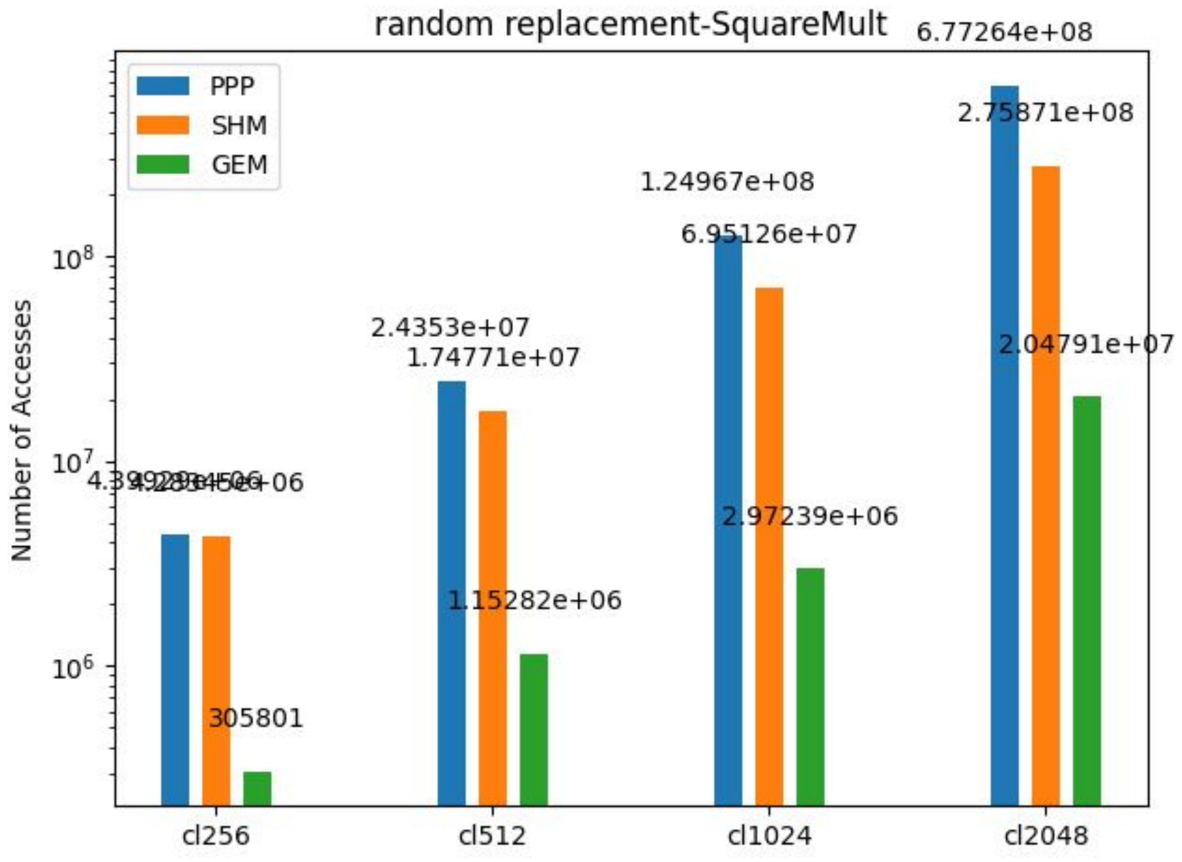
- **Group Elimination Method**
  - It is pretty similar to SHM; the only difference is that instead of taking each line individually, we divide the lines into **groups** and then follow the algorithm.
- **Prime+Prune+Probe**
  - Pre-fill the cache with the set of **candidate** addresses
  - Trigger the victim addresses of interest
  - Any cache misses in the candidate set facilitate locating any **conflicting addresses**(recursive)

Method

- About Cache FX
  - Cache security evaluating **framework**
  - Liberty to generate varying caches
  - Eviction set building - hits/misses count, eviction set size…
  - 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**.
- Limitations
  - Noise-free environment
  - Attacker synchronized with victim
  - Lower bound for cache security

Observations

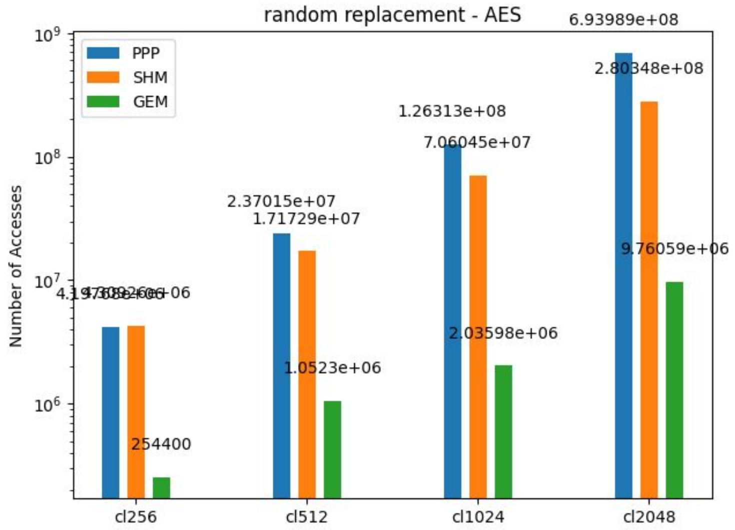
- Number of accesses required to create an eviction set increases with size(almost ten times on each time we double the cache size)
- Trend among our eviction set-creating algorithms is the same in all our cache sizes

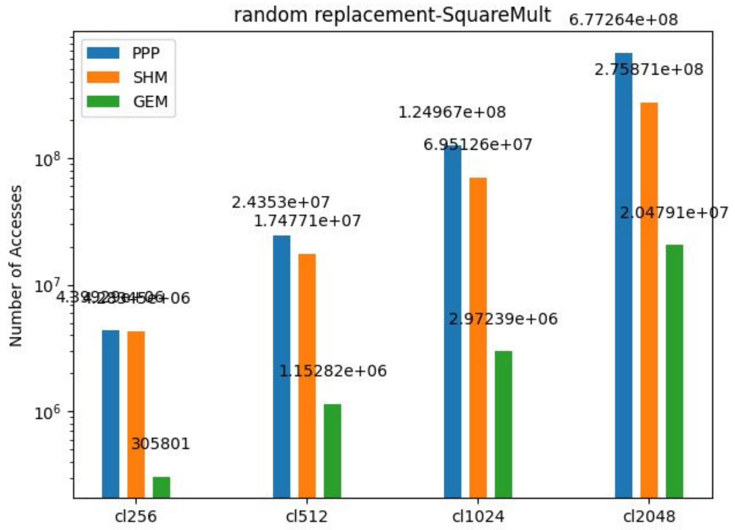


Observations

Different Victim programs

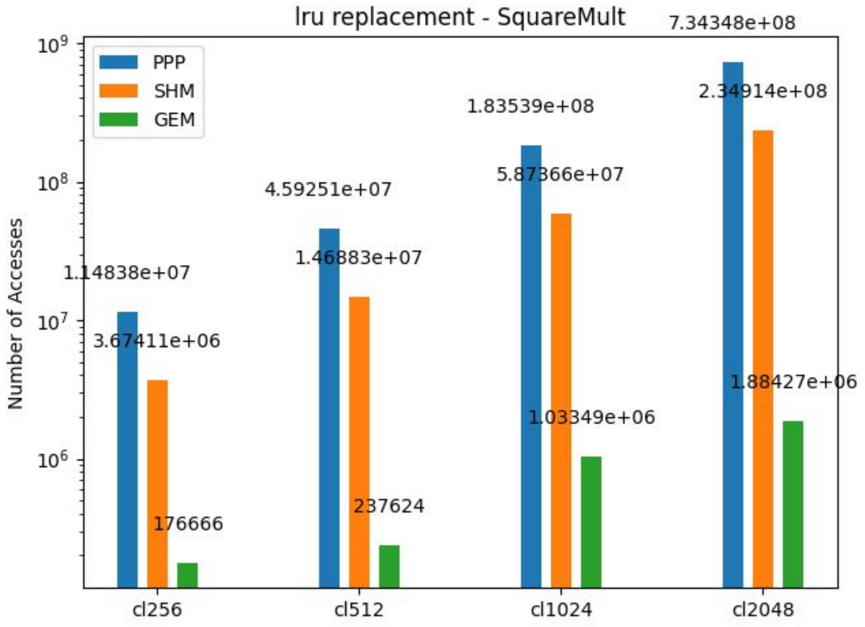
- Almost similar results. The eviction set creation algorithms follow the same trends.
- It becomes evident that the **victim program has very little to do with how hard it is to create an eviction set.**
- slight variations because of a change in the nature of the victim program, the order of the number of accesses remains the same
- Same results on same conditions, Cache FX provides **no randomness**

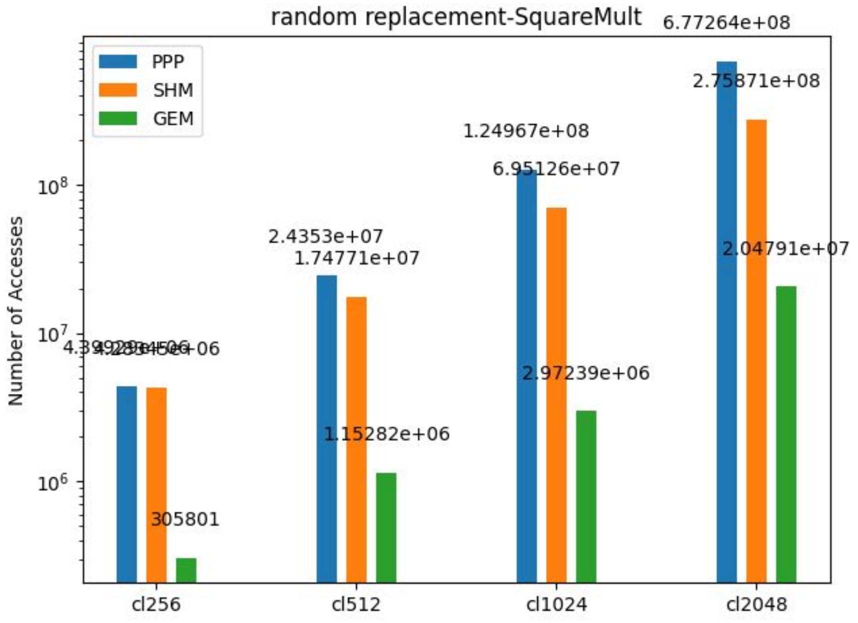




Observations LRU vs Random

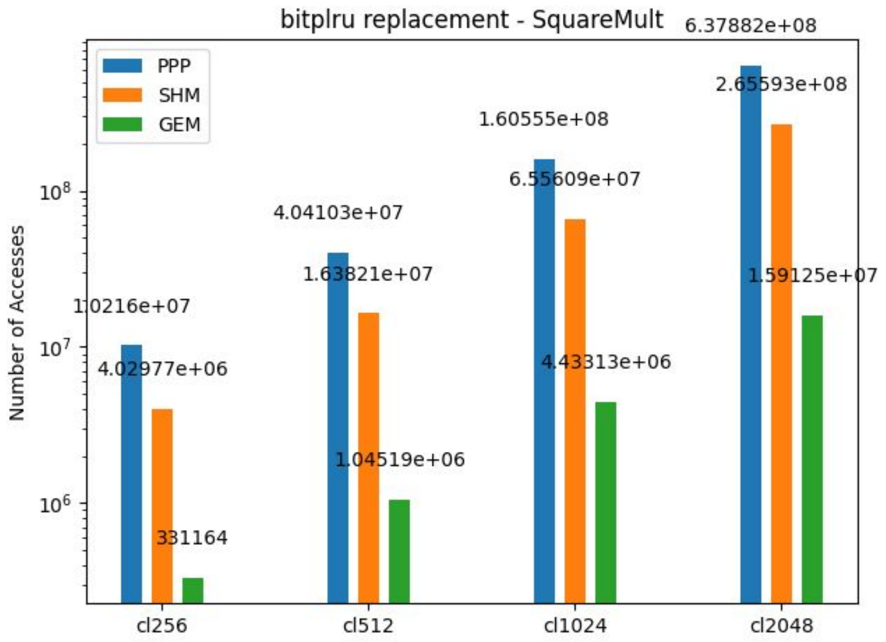
- Significantly better endurance against PPP for smaller cache sizes
- Performs poorly against GEM.

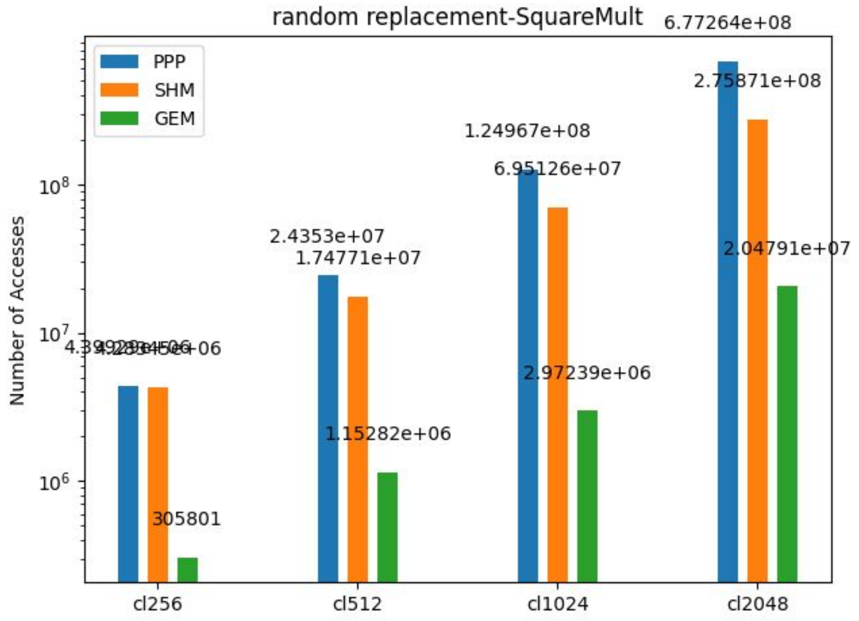




Observations
Bitplru vs Random

- More secure against PPP for smaller cache sizes

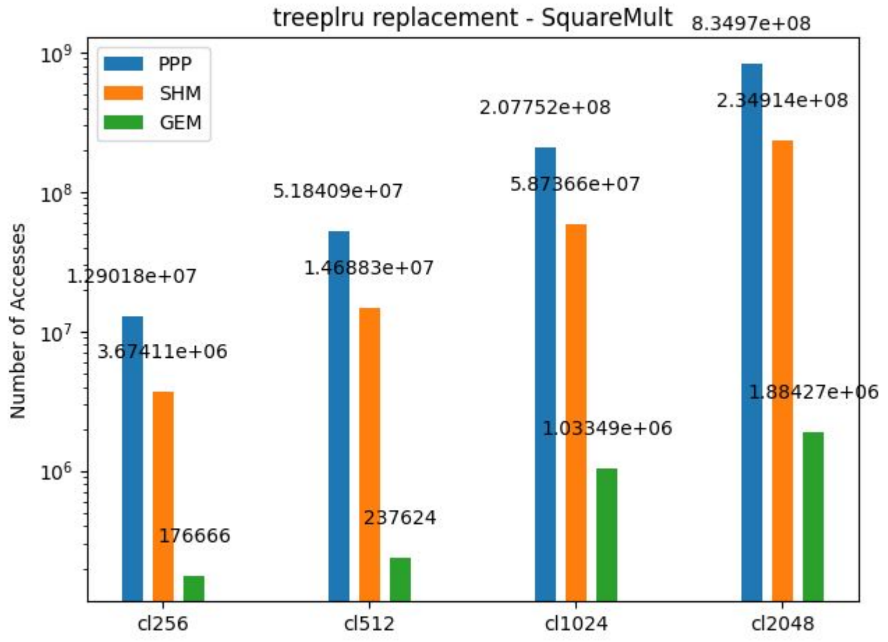


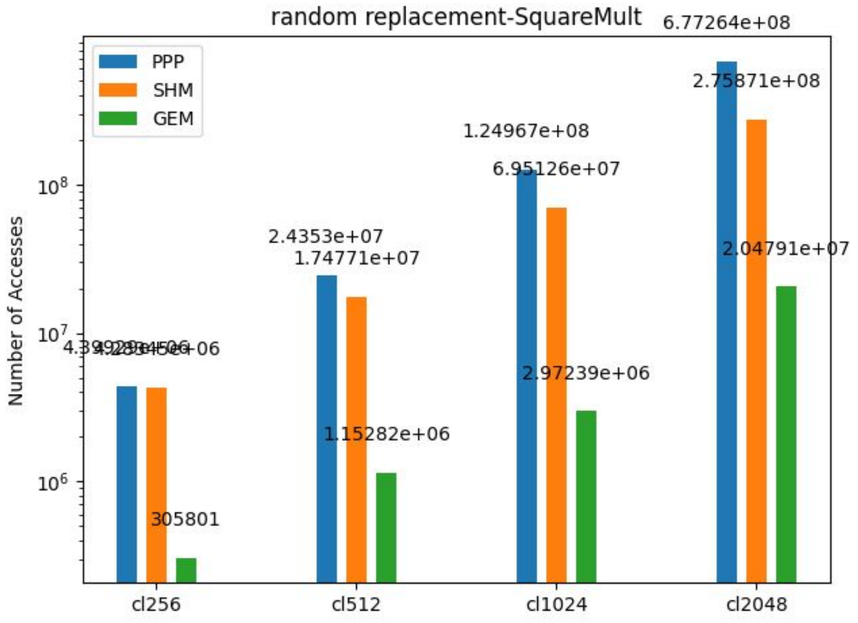


Observations Treeplru vs Random

- Significantly better endurance against PPP for smaller cache sizes
- Performs poorly against GEM.

“Same as LRU”





Results

- GEM performs exceptionally well than other eviction set-building strategies
- The random policy is, after all, not so random.
    - computers are deterministic machines(Mostly)
    - the fact that we can create an eviction set against this replacement policy verifies it
    - the same eviction set, read in the same sequence, always evict the entire cache

Results

- The eviction set size is equal to the cache size
    - In an ideal case, there might also be self-evictions in the cache while creating our eviction set, making the cache more secure.
    - A larger 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.
- Possible Solution: Increasing Cache size?
    - 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.

Thank You