Cache Template Attacks:

Automating Attacks on Inclusive Last-Level Caches

Daniel Gruss, Raphael Spreitzer, and Stefan Mangard

September 26, 2017

Cache Template Attacks

Cache Template Attacks

- State of the art: cache attacks are powerful
- Problem: manual identification of attack targets

Cache Template Attacks

- State of the art: cache attacks are powerful
- Problem: manual identification of attack targets
- Solution: Cache Template Attacks
- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software

Cache Template Attacks

- State of the art: cache attacks are powerful
- Problem: manual identification of attack targets
- Solution: Cache Template Attacks
- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software
- Examples:
  - Cache-based keylogger
  - Automatic attacks on crypto algorithms

Architecture and OS Assumptions

- Last-level cache on modern Intel CPUs:
  - Inclusive (to lower levels)
  - Not in Last-level cache $\Leftrightarrow$ not cached
  - Shared (between cores)

Architecture and OS Assumptions

- Last-level cache on modern Intel CPUs:
  - Inclusive (to lower levels)
  - Not in Last-level cache $\Leftrightarrow$ not cached
  - Shared (between cores)
- The OS allows programs to map other programs code and data into their own address space

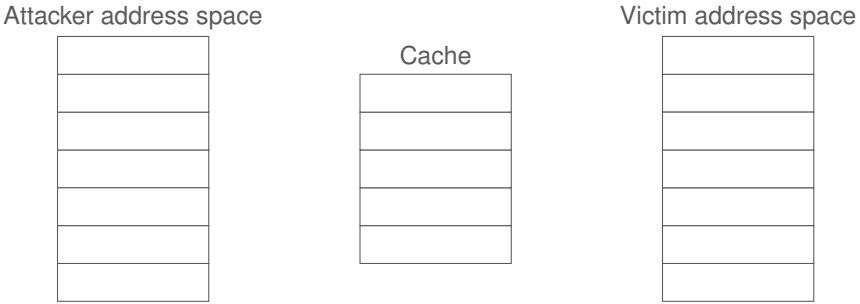
Memory Sharing Scenarios

- Forking a process, copy-on-write is used when the data is modified.
- Running another instance of an already running program
- Users can request shared memory using mmap
- Content-based deduplication – deployed by the hypervisor
  - All mappings to identical pages are redirected to only one, freeing other pages.
  - Memory is shared between unrelated processes in different machines

Flush+Reload

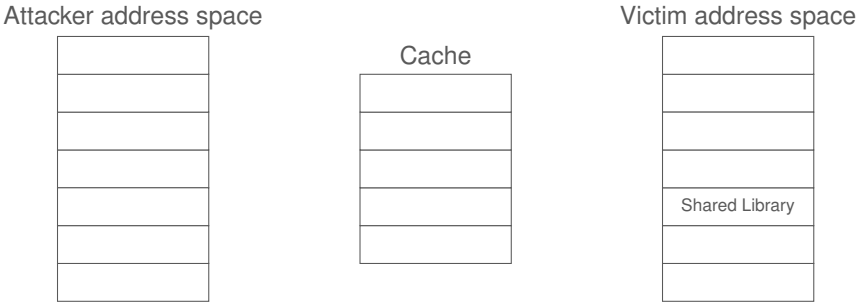
- Powerful cache attack
- Works on shared binaries/libraries
- Application on crypto algorithms

Flush+Reload



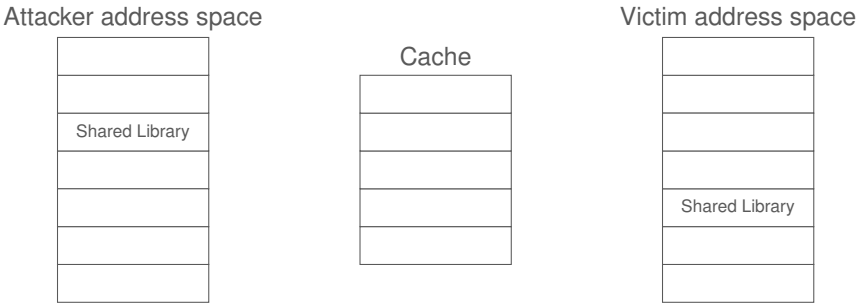
Cache is empty

Flush+Reload



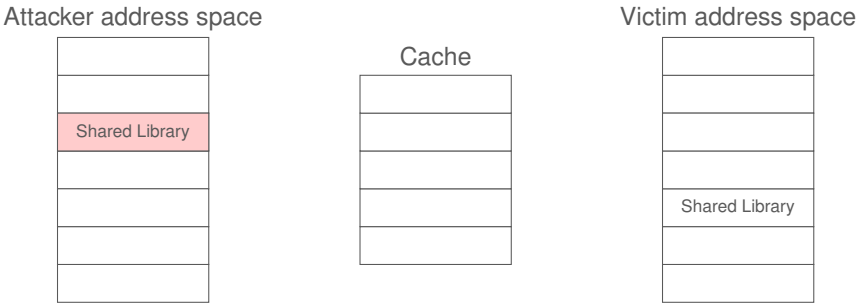
Victim maps shared library

Flush+Reload



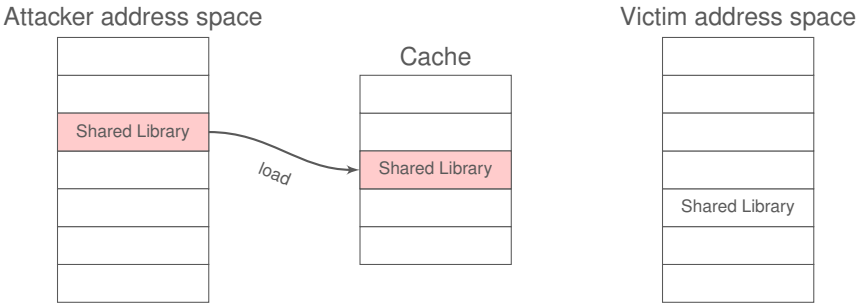
Attacker maps shared library

Flush+Reload



Attacker accesses shared library

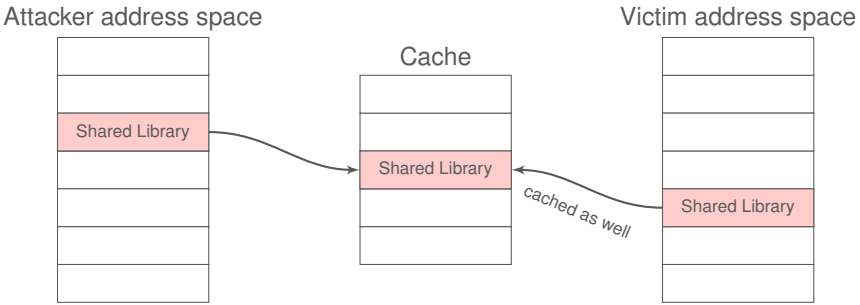
Flush+Reload



Loading into cache...

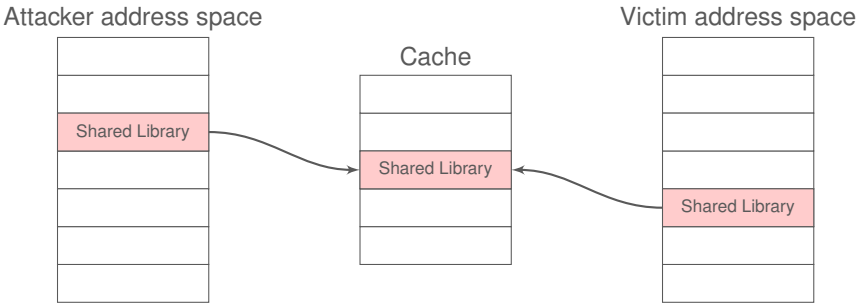
Presented By: Azin Heidarshenas

Flush+Reload



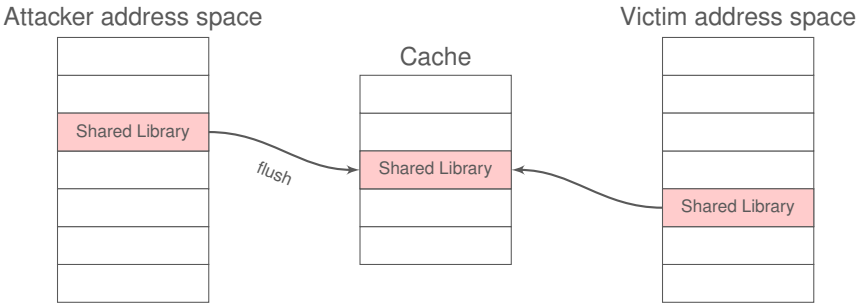
Loading into cache...

Flush+Reload



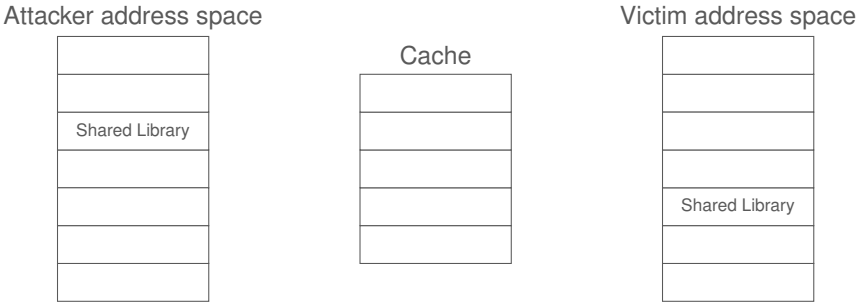
Attacker measures high latency

Flush+Reload



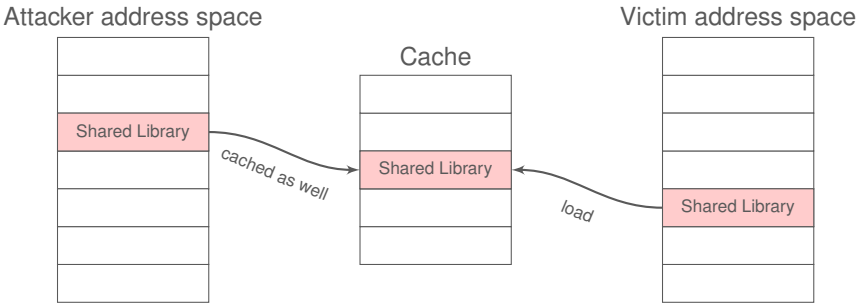
Attacker flushes shared library (“flush”)

Flush+Reload



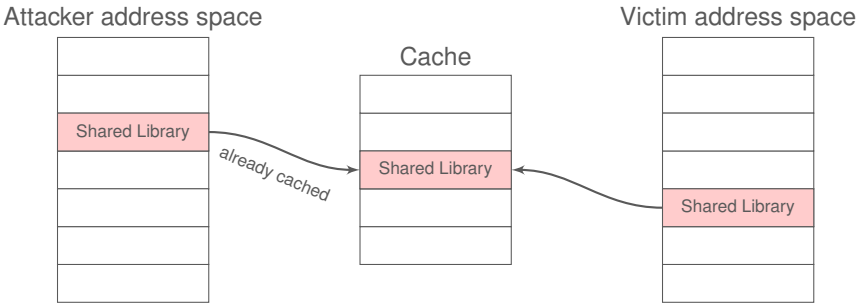
Cache is empty again

Flush+Reload



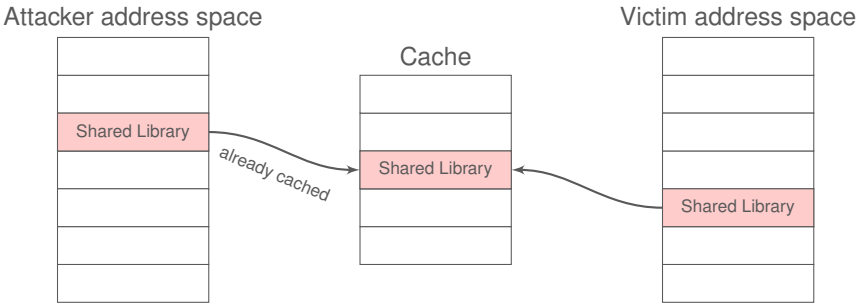
Victim accesses shared library

Flush+Reload



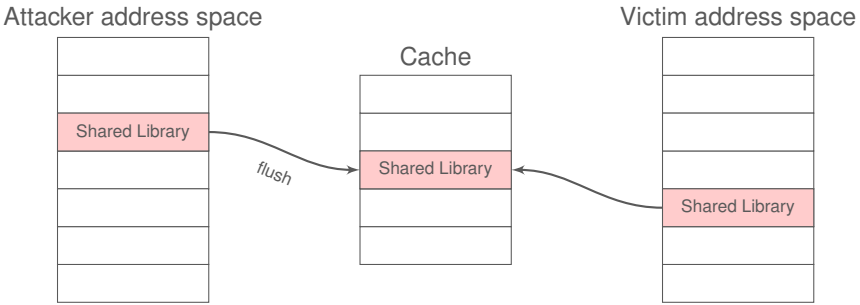
Attacker accesses shared library (“reload”)

Flush+Reload



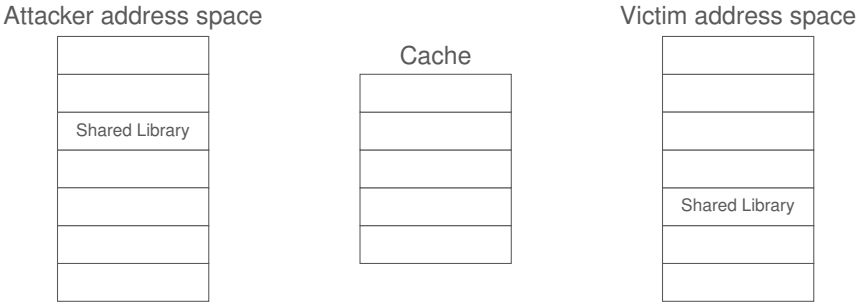
Attacker measures low latency

Flush+Reload



Attacker flushes shared library (“flush”)

Flush+Reload



Cache is empty again

Use Cases

- Can use this technique to build a key-logger by knowing the specific LLC cache line
- But ...

Challenges

- How to locate key-dependent memory accesses?

Challenges

- How to locate key-dependent memory accesses?
- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries

Challenges

- How to locate key-dependent memory accesses?
- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
- Difficult to find all exploitable addresses

Cache Template Attacks

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Exploitation Phase

- Monitor exploitable addresses

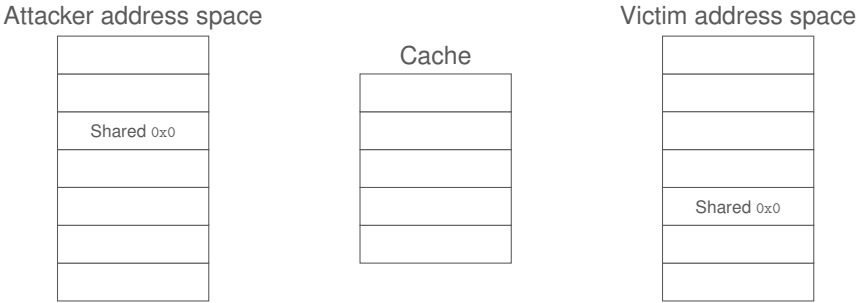
Profiling Algorithm

Algorithm 1: Profiling phase.

Input: Set of events E, target program binary B, duration d
Output: Cache Template matrix T

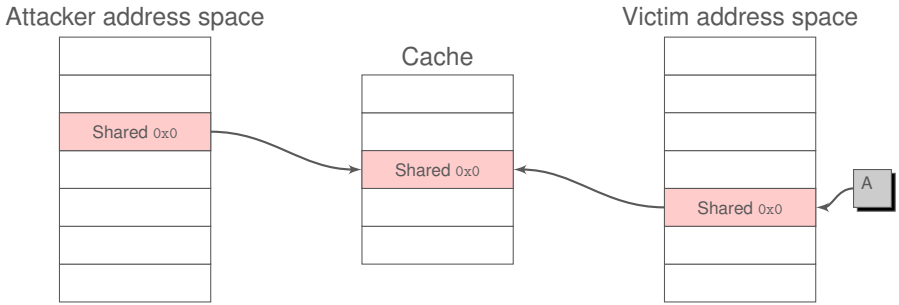
```
Map binary B into memory
foreach event e in E do
    foreach address a in binary B do
        while duration d not passed do
            simultaneously
            Trigger event e and save event trace g_{a,e}^{(E)}
            Flush+Reload attack on address a
                and save cache-hit trace g_{a,e}^{(H)}
        end
        Extract cache-hit ratio H_{a,e} from g_{a,e}^{(E)}
            and g_{a,e}^{(H)} and store it in T
    end
end
Prune Cache Template matrix T
```

Profiling Phase



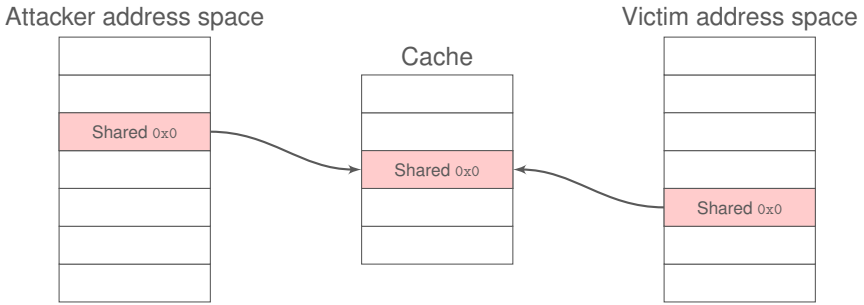
Cache is empty

Profiling Phase



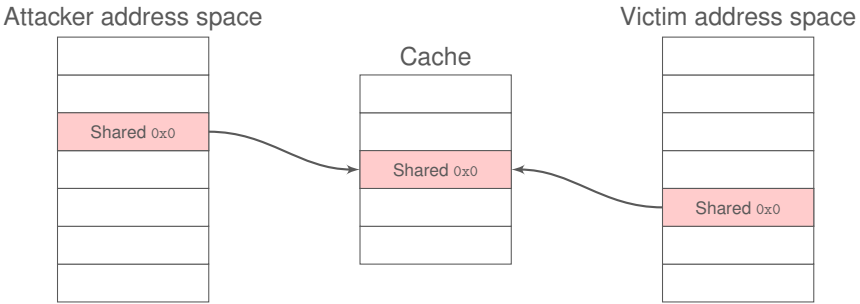
Attacker triggers an event

Profiling Phase



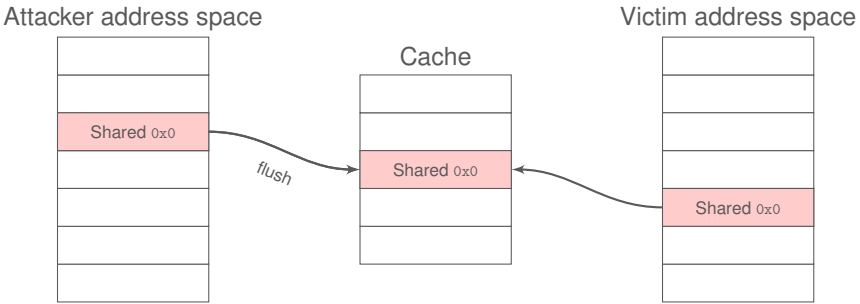
Attacker checks one address for cache hits (“Reload”)

Profiling Phase



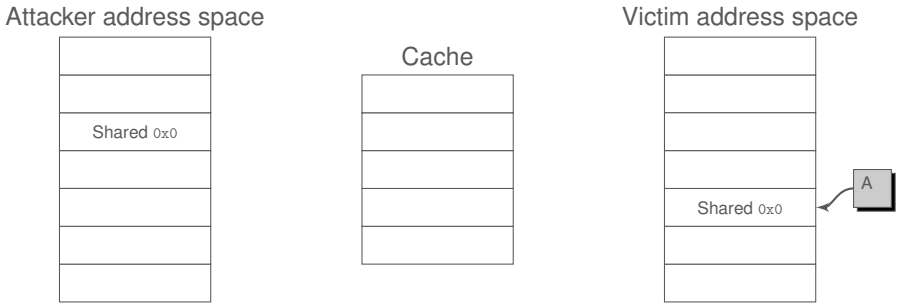
Update cache hit ratio (per event and address)

Profiling Phase



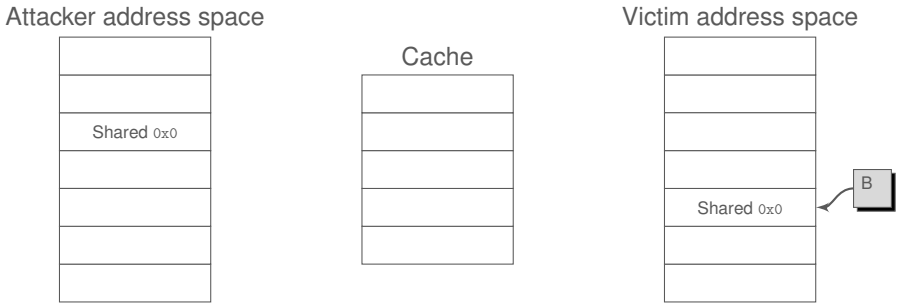
Attacker flushes shared memory

Profiling Phase



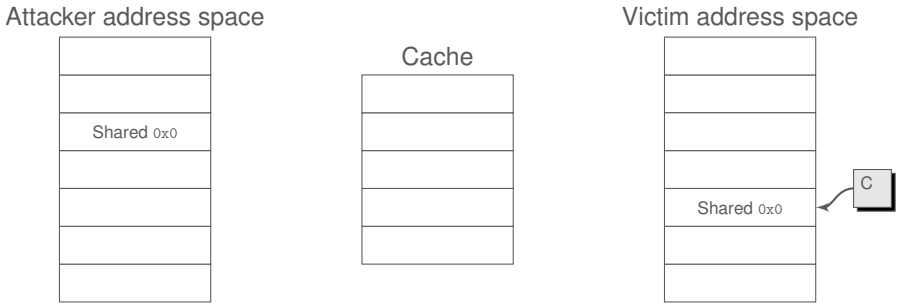
Repeat for higher accuracy

Profiling Phase



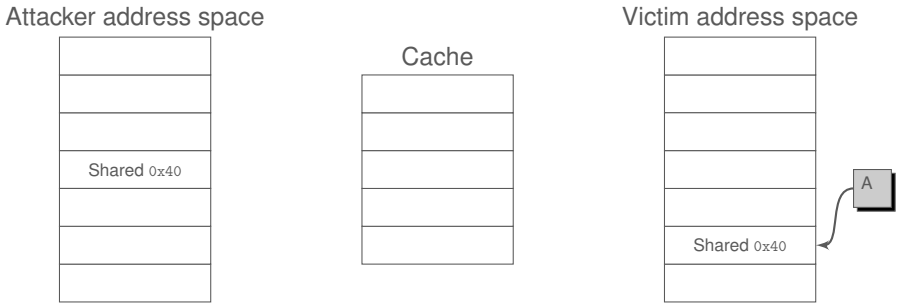
Repeat for all events

Profiling Phase



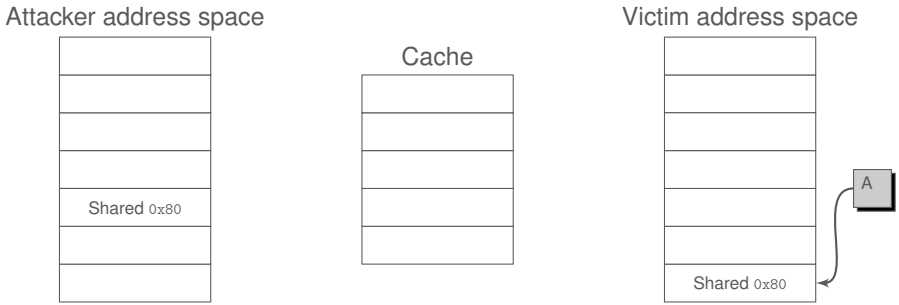
Repeat for all events

Profiling Phase



Continue with next address

Profiling Phase



Continue with next address

Profiling summary

- Trigger events, save event trace and cache-hit trace per address
- Extract cache hit-ratio
  - Can be time dependent/independent
- Pruning - limitation in the number of exploited addresses
  - For some addresses, the hit ratio may be independent of the event

Exploitation Phase

- Monitor addresses from Cache Template
- Compute similarity of hit-trace between profile-time and exploitation-time to identify the occurred event

Exploitation Phase

- Monitor addresses from Cache Template
- Compute similarity of hit-trace between profile-time and exploitation-time to identify the occurred event
- Report to log file / attacker

Exploitation Phase

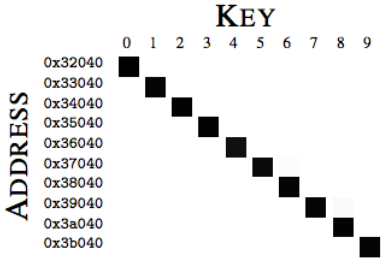
- Monitor addresses from Cache Template
- Compute similarity of hit-trace between profile-time and exploitation-time to identify the occurred event
- Report to log file / attacker
- Manual analysis of log file
  - Find password in keypress log, etc.

Example Artificial Attack

```
1 int map[130][1024] = {{-1U},...,{-130U}};
2 int main(int argc, char** argv) {
3   while(1) {
4     int c = getchar(); // unbuffered
5     if (map[(c % 128) + 1][0] == 0)
6       exit(-1);
7 } }
```

Listing 1: Victim program with large array on Linux

Cache Template Matrix



Example Attacks

Keylogging Attack

- Linux with GTK: monitor keystrokes of specific keys
- Detect groups of keys
- Some keys distinct

Discussion of Countermeasures

- Removal of clflush does not help - there are alternatives
  - Enforce eviction by accessing congruent physical addresses
  - Might incur false positives (w.r.t cache hits)
- Disable cache line sharing
  - Can turn off page deduplication by the hypervisor
  - Disable shared memory by the OS? Have to modify OS.
  - Use virtually tagged caches
- Cache set associativity

Enhancing the Prefetcher

- Spy may not be able to simultaneously exploit consecutive addresses
- Good! Increase prefetch trigger distance
  - Will effectively shrink attack granularity
- If only prefetching were to take temporal locality into attack, attacks will be too hard

Conclusion

- Novel technique to find any cache side-channel leakage
  - Attacks
  - Detect vulnerabilities

Conclusion

- Novel technique to find any cache side-channel leakage
  - Attacks
  - Detect vulnerabilities
- Works on virtually all Intel CPUs
- Works even with unknown binaries

Conclusion

- Novel technique to find any cache side-channel leakage
  - Attacks
  - Detect vulnerabilities
- Works on virtually all Intel CPUs
- Works even with unknown binaries
- Marks a change of perspective:

Conclusion

- Novel technique to find any cache side-channel leakage
  - Attacks
  - Detect vulnerabilities
- Works on virtually all Intel CPUs
- Works even with unknown binaries
- Marks a change of perspective:
  - Large scale analysis of binaries
  - Large scale automated attacks

Cache Template Attacks:

Automating Attacks on Inclusive Last-Level Caches

Daniel Gruss, Raphael Spreitzer, and Stefan Mangard

September 26, 2017