The Center for Cyber Defenders

Expanding Computer Security Knowledge

SAND2015-6162C

Flush and Reload: an L3 Cache Timing Attack

Project Mentor: Severiano Sisneros, 05635

Raewyn Duvall, Tufts University

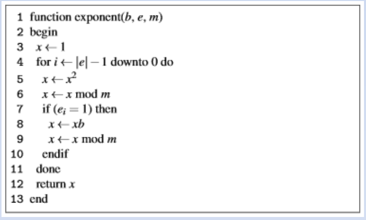
Problem Statement:

The goal of this project was to recover the secret key in GnuPG RSA encryption/decryption by timing the memory access.

Objective and Approach:

The object of this project was to mimic "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack" by Yarom and Falkner.

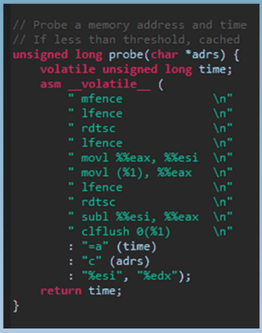
RSA generates an encryption by selecting two random prime numbers p and q and calculating $n = pq$. GnuPG uses $e = 65537$ as it's public exponent; then RSA calculates a private exponent $d \equiv e^{-1} (mod(p-1)(q-1))$. The encryption then consists of the public key, (n,e); the private key, (p,q,d); the encrypting function, $E(m) = m^e mod(n)$; and the decrypting function, $D(c) = c^d mod(n)$. To compute the encryption and decryption functions, GnuPG uses the square and multiply exponentiation algorithm that is shown in Figure 1. It shows how for every bit of e, a square and reduce are performed, and if the bit is a 1, a multiply and reduce are also performed. Therefore, if it can be determined which patterns of square, reduce, and multiply are called, then it is possible to get all the bits to the private key.



```
function exponent($b$, $e$, $m$)
begin
  $x \leftarrow 1$
  for $i \leftarrow |e| - 1$ downto 0 do
    $x \leftarrow x^2$
    $x \leftarrow x$ mod $m$
    if ($e_i = 1$) then
      $x \leftarrow xb$
      $x \leftarrow x$ mod $m$
    endif
  done
  return $x$
end
```

Figure 1

In the x86 architecture, all processors share the Last-Level L3 Cache, therefore all memory will be copied in the L3 Cache when accessed. With this information, we timed memory access during GnuPG RSA decryption to see if those square, reduce, and multiply functions had been cached recently and therefore used. The timing code is pictured in Figure 2.



```
// Probe a memory address and time
// If less than threshold, cached
unsigned long probe(char *adrs) {
    volatile unsigned long time;
    asm __volatile__ (
        " mfence             \n"
        " lfence             \n"
        " rdtsc              \n"
        " lfence             \n"
        " movl %%eax, %%esi  \n"
        " movl (%1), %%eax   \n"
        " lfence             \n"
        " rdtsc              \n"
        " subl %%esi, %%eax  \n"
        " clflush 0(%1)      \n"
        : "=a" (time)
        : "c" (adrs)
        : "%esi", "%edx");
    return time;
}
```

Figure 2

Results:

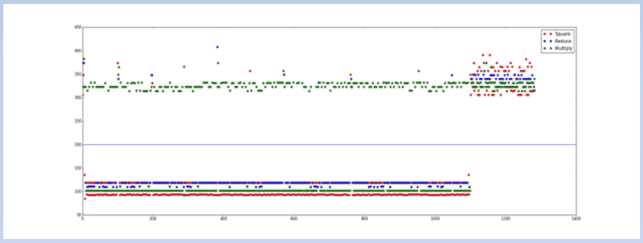


Figure 3

Figure 3 shows a visualization of the timings of each function over the course of the attack. Looking at the points below the threshold (blue solid line), when there was a Square-Reduce-Multiply-Reduce pattern, the key contained a 1; when there was only a Square-Reduce pattern, the key contained a 0. Going through all the output this way, we determined the binary of the secret key used in GnuPG RSA encryption/decryption. Unfortunately there has been a lot of noise that skews the results, so we are currently creating a process to eliminate the noise by averaging a lot more data.

Impact and Benefits:

This is a proof of concept that this attack is viable. It can be used to create other cache side-channel attacks that we can use to find and fix vulnerabilities before they're exploited.