Attacking SHARP through Cache Side-Channels with Multiple Spies

Joseph Reeves and Nuno Sabino

December 6, 2021

1 Introduction

Modern computer architectures support parallelism at multiple levels, providing many cores that can each interleave many threads. The parallel execution of independent processes relies on shared hardware to maintain efficiency. While a malignant program cannot directly observe the instructions executed by other programs, it can take advantage of shared hardware to learn information indirectly. In this work we focus on attacks that leverage eviction protocols in a shared L3 cache to learn when certain instructions are used by another process.

Algorithm 1: Square-and-Multiply exponentiation.

```
   Input  : base b, modulo m, exponent e = (e_{n-1}...e_0)_2
   Output : b^e mod m
1  r = 1
2  for i = n - 1 downto 0 do
3  |   r = sqr(r)
4  |   r = mod(r, m)
5  |   if e_i == 1 then
6  |   |   r = mul(r, b)
7  |   |   r = mod(r, m)
8  |   end
9  end
10 return r
```

Figure 1: Square and Multiply algorithm. Image from the SHARP paper [4].

As an example, take the square and multiply algorithm [2] used to compute encrypted messages for RSA encryption. The program executing the algorithm in Figure 1, which we call the *victim*, will execute line 6 only when the current bit of the private exponent is 1. Regardless of the bit value, the victim will execute line 3 after each iteration. Note that the time between iterations is large because the instructions are often operating on big numbers (> 2000 bits). If a malignant process, which we will call the *spy*, knows the victim's execution pattern of the lines 3 and 6, it can recover the encryption key used by the victim.

Many styles of cache side-channel attacks to recover the encryption key have been researched, including *flush and reload*, *evict and reload* and *prime and probe*, to name a few.

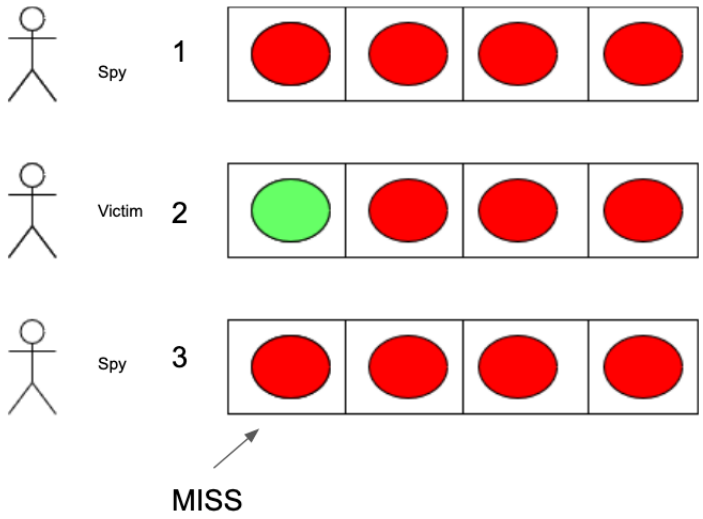

Figure 2: Prime and probe attack showing content of a set in a 4-way L3 shared cache.

We will focus on prime and probe attacks, which we exemplify in Figure 2. This attack assumes the victim and spy are running on different cores but share an L3 cache. In addition, the spy must know which cache set the address of mul (called in instruction 6 in Figure 1) is mapped to. The attack begins with the spy priming the cache set through several load instructions. In step 1 the spy makes four loads to fill up the cache set with an associativity of four. Then the victim may execute instruction 6 and call the mul function, loading it (represented as the green dot) into memory if there is an exponent in the encryption key. During step 2 the spy waits. At step 3 the spy reloads all four memory addresses, and if there is a miss the spy assumes the victim had an exponent bit. Alternatively, if the victim did not have an exponent bit the spy should have all hits. Ideally, the spy could run this attack on multiple uses of the victim's encryption key and combine the information found at each iteration. This is important because the process is inherently noisy and other cores may be accessing memory in the same L3 cache set. However, there are ways to extract a full encryption key from a partial key [1].

A real-world implementation of this attack requires several pieces of information that can be found through experimentation. The address of the desired instruction in memory and the wait time of the spy in step 2 can be determined emperically [5]. Additionally, an L3 cache miss can be inferred by finding average L1, L2, and L3 miss times, assuming there is a significant and measurable difference between L3 misses and the rest of the cache. We do not focus on these platform specific numbers, and instead model our attacks at a high-level with these assumptions baked in.

The *SHARP* [4] protocol presents one way of mitigating cache side-channel attacks by modifying the eviction policy for the shared L3. Our project involved implementing high-level simulations of the SHARP protocol, then deploying two styles of prime and probe attacks that bypassed the protocol. One attack uses multiple spies on multiple cores (*Multiple Spies*) and the other uses spies sharing the victims core (*Shared Core*). We ran experiments on a deterministic python simulator as a proof of concept. We then constructed a pintool to show that our attacks work even when some noise is added to the experiment.

2 SHARP

The Secure Hierarchy-Aware cache Replacement Policy (SHARP) is a cache eviction policy designed to work in a shared L3 cache and mitigate cache side-chanel attacks [4]. The general idea of SHARP is to prevent *inclusion victims*. An inclusion victim is created when data evicted in the shared L3 is active in a core's private L2 cache. That data must then be evicted from the private L2 to preserve inclusivity. Preventing inclusion victims should mitigate attacks of the sort in Figure 2 because the attack relies on the prime step which evicts the victims instruction from memory. This would cause an inclusion victim and is therefore prevented. So, when the spy tries to prime its four pieces of data into the shared L3, it will not be able to fill the cache set and evict the victim.

It is infeasible to prevent inclusion victims in every case, e.g., a cache set may contain cache lines that all create inclusion victims if evicted leading to a deadlock scenario. Therefore, the SHARP design provides a multi-step process for implementing the eviction policy when a core accesses memory:

1. If the cache set contains an unused cache line use it. Otherwise go to 2.
2. If the cache set contains a cache line owned by the calling core, evict the cache line owned by the calling core based on the existing replacement policy ordering. Otherwise go to 3.
3. Evict some cache line at random and increment the alarm counter for the calling core.

The alarm counter helps identify when a core is causing frequent random evictions in step 3. This information can be used by the operating system to investigate processes running on a certain core. The authors recommend using a threshold of 2000 alarm triggers per core every one billion cycles to detect attacks. Legitimate executions should not surpass that value. Core Valid Bits (CVBs) are used to determine when a cache line is owned by a core, i.e., the data is active in the core's private L2 cache. SHARP describes three methods for maintaining the CVBs, involving lazy updates or snoopy monitoring. In our implementation, we assume the CVBs are accurate under a snoopy protocol, meaning the CVB is set iff the data is active in the core's L2 cache. This is an important assumption for the shared core attack.

Since the SHARP paper was published in 2017, there has been research into problems with the protocol [3]. This work shows how the replacement policy can be exploited with denial of service attacks. The attacker owns many cache lines and constantly forces the victim into the step 3 random eviction. It also investigates how useful the alarm counter is in detecting spies, and how the coarse-grained alarm counter lacks information on the thread level. The paper also described a prime and probe attack to bypass SHARP, but the details are insufficient for replication.

We propose two attacks that bypass SHARP. The first attack uses multiple spies each accessing a single cache line in the shared L3. This takes advantage of the random eviction in step 3 of the protocol. The second attack uses a spy on the same core of the victim to invalidate the victim's data in the private L2 cache. This takes advantage of the snoopy CVB monitoring, allowing a second spy to complete the basic prime and probe attack.

3 Multiple Spy Attack

The multiple spy attack takes advantage of the random eviction in step 3 of SHARP by placing spies on n cores assuming the associativity of the L3 cache is n. Figure 3 shows the basic setup

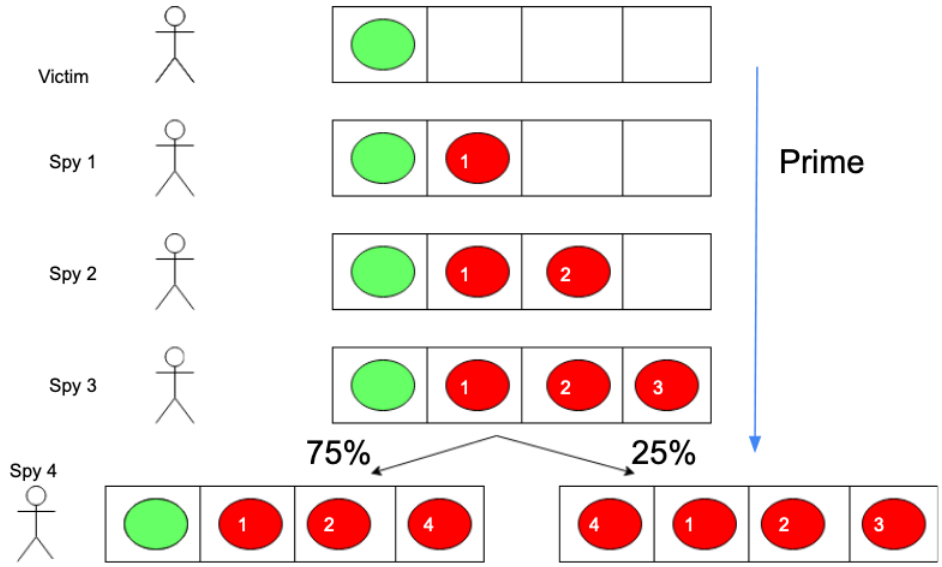

Figure 3: Example of a multiple spy attack at the level three shared cache with a set size of four.

for the attack. Assuming the victim has active data in the L3, $n-1$ spies will load independent data into a single cache line. Then the n^{th} spy loads data with a $\frac{1}{n}$ chance of evicting the victim (shown by the 25% case in the example). The spies then wait for some time and all reload their data, recording any misses at the L3 cache. It is important that the n^{th} spy reload its data first, as this allows more information to be gained. The exponent bit can be inferred in the following cases for each iteration:

1. The exponent bit is 0 if all spies have cache hits. This means the victim never put data into the cache set.

2. The exponent bit is 1 if all spies had cache hits in the previous iteration and some spies have cache misses in the current iteration. This means the victim may have put data into the cache set causing a cache miss.

3. The exponent bit is 1 if the n^{th} spy which loads first after the victim has a cache miss. The n^{th} spy was the last to load during the prime stage, so no other spy could have evicted its data. This means the victim may have evicted it. This happens with probability $\frac{1}{n}$ as the victim would evict some spy at random.

4. Otherwise, the exponent bit is unknown (we represent this a ? in our simulations).

Of course, for case 2 and 3 it may be another program that puts data into the cache set. Additionally, the chances that you can learn information are slightly better than $\frac{1}{n}$ (due to the order of spy loads). So, there will be many -1's in the recovered key, and potentially some 1's that are incorrectly labelled. We account for this by running the attack over many victim's executions. Then the set of partial keys are combined. Since the eviction is random, the chances of at least one of the spies' primes working (right-hand side of Figure 3) for a specific bit in the key become more likely with each additional execution. As an aside, case 3 requires timing between the loads of the n^{th} spy and the other spies, which may be unrealistic for spies acting across several cores.

In this attack, the alarm counter for both the victim and the n^{th} spy would be incremented during the random evictions. However, the alarms can be spread across all spies by rotating which spy makes the n^{th} load. This rotation can be based on a predefined static schedule, so the spies would not need to communicate.

4 Shared Core Attack

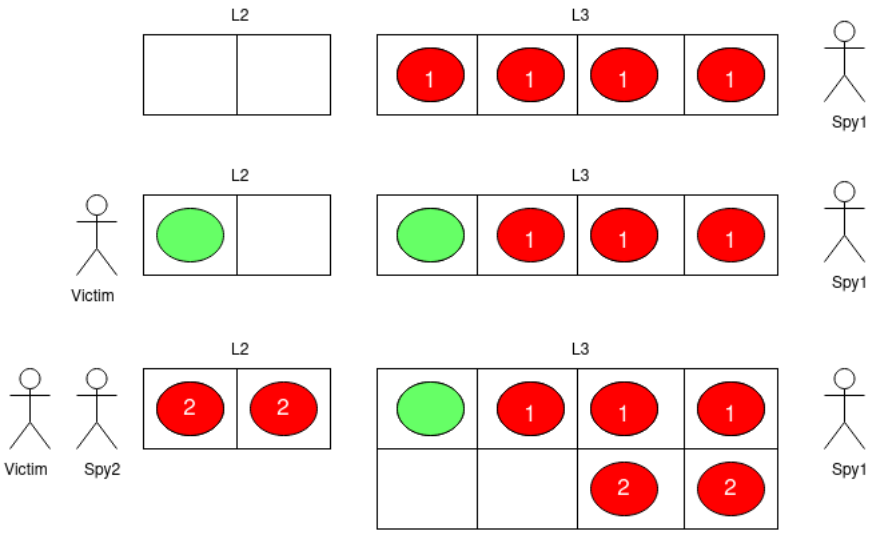

Figure 4: Example of a shared core attack at the level two private and 4-way level three shared cache.

The shared core attack relies on the assumption that the CVBs operate using a snoopy protocol. This allows a spy on the same core as the victim to invalidate memory at the private L2. Figure 4 shows an example of this attack, where the L2 is a private cache shared between Spy2 and the victim. In this attack Spy1 is tasked with priming the shared L3 cache set and collecting information about the encryption key exponents. To do this, Spy1 first primes the cache set with n loads. Then Spy1 waits as the victim accesses the exponent bit instruction, bringing that instruction into the L3 and L2. After the victim has accessed the instruction, Spy2 fills the L2 set containing the victim's data. If the L2 uses an LRU-based protocol, this will evict the victim's data. Note that this data should be mapped to a separate cache set in the shared L3. It is crucial that at this point the snoopy monitoring invalidates the CVB for the victim's data in the L3. Finally, Spy1 reloads n times encountering one miss if the victim has accessed data. This miss allows the spy to infer a bit of value 1, and all hits corresponds to value 0. Importantly, on the miss Spy1 will evict the now inactive victim cache line and once again fill the entire cache set. This would not be possible if the victim data was still in the private L2.

This attack benefits from avoiding step 3 in the SHARP protocol, so no random evictions will occur for the spies. In addition, Spy1 will never increment its alarm counter because it never loads more than n independent cache lines. On the other hand, the victim will increase its alarm counter with each access to the exponent instruction. This inverts the intended affect of the alarm counter. For this attack it may be difficult to consistently evict data in the L2 with Spy2. Since the processes share the same core it is not clear how thread scheduling and blocking on a cache miss would affect Spy2.

5 Evaluation

We implemented two simulators to model our attacks. The first was a high-level simulator built in Python. This simulator implements the SHARP protocol and both attacks, along with a representation of the L2 and L3 cache lines in question. It abstracts away instructions and timing, meaning the encryption key either has an exponent bit and the victim accesses memory or it does not. The square and multiply algorithm is not implemented. The spies are also assumed to know the appropriate wait time and recognize cache misses correctly.

This was simply a prototype to quickly evaluate the two attacks under near perfect conditions. The tool can be used with the following options:

- -s Cache set size (spawn same number of spies for multiple spies attack)
- -k RSA key exponents, e.g. 011000110
- -i Iterations of RSA key detection
- -a 1 for multiple spies attack, 2 for shared core attack

We sought to further evaluate the attacks in a more robust simulation framework that included noise. To do this we decided to use a pintool. This was the best option, as the simulators used in [4, 3] had poor documentation and the authors of SHARP were unable to give us their source code. Pintool does not have support for inserting analysis code into multiple executable files that are running on multiple cores simultaneously. To get around this, we used the victim program [1] as the executable. The spies, cache hierarchy, SHARP protocol, and global timing were implemented in the pintool itself.

5.1 Multiple Spy Attack Evaluation

In the shared core attack the spy on the separate core can determine through repeated probes when the square instruction has been used by the victim, prompting the spy to check if the multiply instruction was used. This is not possible in the multiple spy attack. Assuming the spies cannot communicate in real-time, which would require significant overhead and coordination, the spies combine information about hits and misses after execution to generate the leaked key. Under this framework, the spies will only collect information about the multiply instruction. Each spy acts independently with the following operation:

1. load (multiply address + line size $\times$ L3 set number $\times$ spyId)
2. if load time is greater than L3 hit time, record miss
3. wait for t amount of time then repeat

The difficulty of this approach is determining a good t value. We experimentally found the instruction count in between iterations of the RSA algorithm with an exponent is 7244 and without an exponent is 4980. If $t > 7244$ the spies will skip some iterations as they collect the leaked bits and if $t < 7244$ the spies will collect the same leaked bit twice for some iterations. We cannot adjust t dynamically since the spies cannot determine if the multiply instruction was used until all spy information is combined.

[1] RSA implementation found at
`https://gist.github.com/Thelouras58/a3b04a3df0d167743084ff94442f52d8`

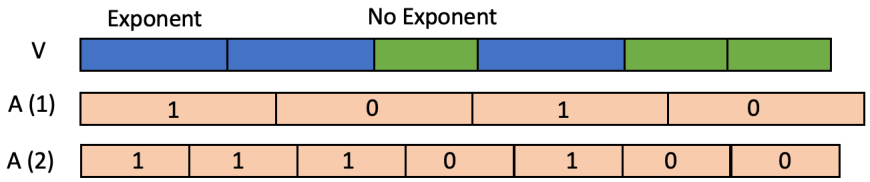

Figure 5: Example scheduling with victim (v) on the top line where operation time when key has an exponent is longer (blue) than if the key does not have an exponent (green). The attackers wait time can be long (A1) but some iterations will be missed, and if the wait time is short (A2) some iterations will be duplicated.

For our experiments we choose t to be 4980 ensuring we collect at least one bit each iteration. This corresponds with A2 of Figure 3. We run multiple executions and combine the partial keys in an attempt to recover the original key. Recall each iteration can only learn approximately $\frac{1}{n}$ bits because of the random eviction in step 3 of SHARP. The random values are computed dynamically so we learn a different $\frac{1}{n}$ bits with each execution.

```
10??01???
?00???1??
1????1??1
...
```

The partial keys after each execution will look like the strings above. We line them up and generate the full key when all positions in the key have been determined. Conflicting bits between partial keys could happen if for example another program was loading data into the cache set. Since $t < 7244$ the full key will be longer than the original key with some duplicated bits where the spies probed twice with a single square and multiply iteration. This could be solved by varying t values across executions to determine when bits have been duplicated in the full key, but we did not have time to investigate this feature.

Figure 6 shows the results of combining keys after each execution for a set size of 16 with t as 4980. The keys were combined by lining them up based on the first leaked bit (they all found the first exponent bit because they primed the sets before the victim made its first access). They were matched against the original key by sweeping through the combined key and determining if the bit at the current position matched the original bit or matched the original's previous bit in the case of a duplicate. If neither values matched or there was an unknown bit in the combined key, the iteration continued at the next position. The number of unknown bits after each execution decreases by approximately $\frac{1}{16}$ after each execution as expected. The number of correct bits and duplicates should have an inverse relation to the unknown bits but this is not the case. These results may have been affected by conflicting bits in the partial keys. The spies loaded in n consecutive instructions, and if the victim loaded within that window it could cause conflicting information between executions. In addition, the combined key could be matched with the original key in a more sophisticated way. Using some other metric for comparing strings may show better results.

These results can be duplicated using the script:

```
sh multi-spy-attack.sh <iterations> <square addr> <multiply addr>
```

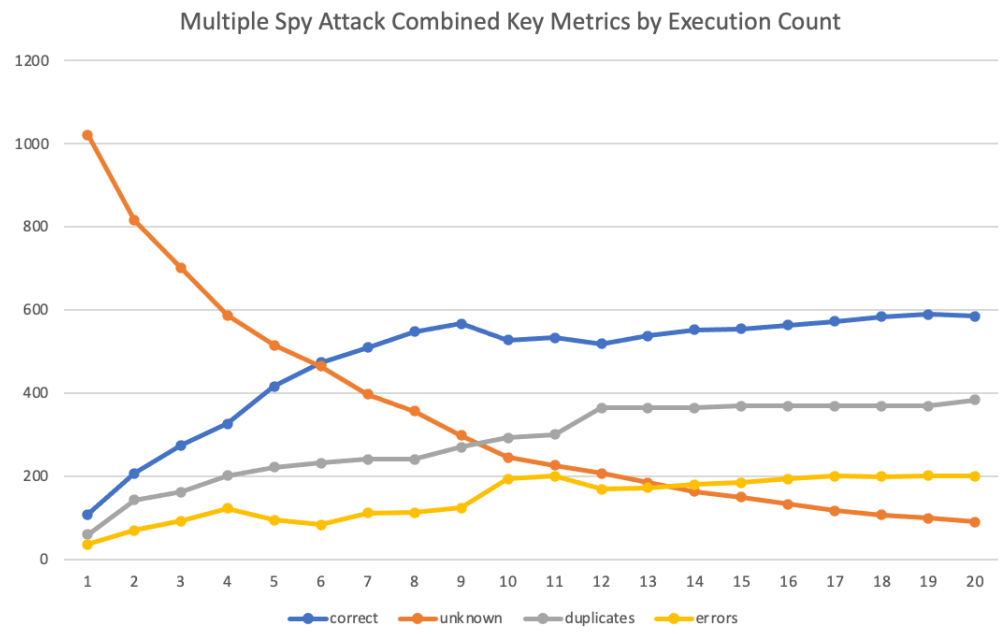

Figure 6: Combining keys after each iteration and measuring the number of unknown bits, the number of duplicated iterations, the number of incorrect matches, and the number of correct matches.

5.2 Shared core Spy Attack Evaluation

The spy that shares the core with the victim only has to repeatedly evict the multiply and square addresses from its L2 cache, without evicting them from the L3 cache. Thus, it does:

1. load (multiply address + line size $\times$ L2 set number $\times$ i) for $i \in \{1..L2\_ASSOCIATIVITY\}$
2. load (square address + line size $\times$ L2 set number $\times$ i) for $i \in \{1..L2\_ASSOCIATIVITY\}$

Probabilistically, the loaded addresses will correspond to a completely different set in the L3.

The other spy is the one that actually leaks the private key. It can be seen as a state machine. In the $initial\_state$, it tries to detect whether a new iteration has started. If so, it moves to the $iteration\_found$ state and waits t_1 cycles. Otherwise, it waits t_2 cycles. In the $iteration\_found$ state, it checks whether the exponent bit is a 1. It waits t_3 cycles and moves to the $initial\_state$ regardless, but if exponent is really a 1 it ways an extra t_4 cycles. This is because if the exponent is a 1, the multiply will be called and therefore the number of cycles until we start the next iteration will be greater.

The way we found t_1, t_2, t_3 and t_4 is similar to the multiple spy attack we described earlier.

Even in a simulator, we had to overcome a series of technical difficulties due to the difficulty of implementing this kind of attacks. We enumerate a few of the challenges:

1. Other instructions executed by the victim (or accessed data) can map to the same set as the multiply / square address. For example, if the square and multiply addresses are very close to eachother, they might be mixed up during iteration/exponent bit detection. To fight this, we can choose addresses from the start or end of each of the functions to maximize their distance. Thus, we might choose to not pass the exact start addresses of

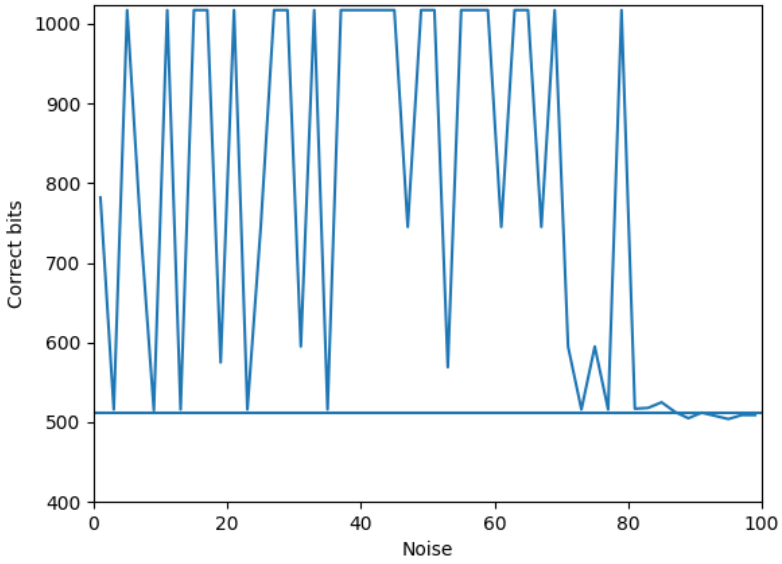

Figure 7: Comparing number of correctly leaked bits of private exponent depending on the noise in cache loads.

the square/multiply address to our pintool. We just need instructions that will be executed if and only if the target function is executed.

2. Depending on what is the current state, the second spy might need to wait a different number of cycles until its next action. If the exponent bit is a 1, the multiply function will be called and therefore there will be more cycles executed during that iteration. We ended up splitting that in four different times and empirically determine them.

3. If one or more iterations are not detected correctly (i.e. we either missed an iteration or incorrectly detected an extra iteration) the combined key will be hard to compare to the leaked key, as it will have a different number of bits. It is hard to combine the result of multiple executions if we are not sure when some iterations started. Thus, we should always prioritize the correct identification of the square address hit.

Figure 7 represents how many bits of the private exponent can we correctly identify, relatively to how much noise we have during cache loads. The x-axis represents the number of cycles that a load will vary from its baseline. The L3 cache miss penalty is 120 cycles. L3 baseline hit time is 40 cycles (situation where L1 and L2 miss but it hits L3). This will have an impact on the spy hit detection, both for the iteration start detection or the exponent value for the current iteration.

As we can see, if the overall noise is smaller than 80 cycles we can recover the private exponent, even if we need multiple executions to account for possible errors. From that point on, either the attack is not effective, or we are not comparing the resulting leaked key in the best way. Imagine that we missed the first iteration (so we have one more bit than the actual key). Then the bits of the leaked key will look incorrect, but actually they are just shifted to the right by 1 bit. We leave as future work the study of a good way to compare the results of this attacks against the real key and how to go from that to a real key recovery attack.

The alarm clock for the first core is usually around 2600 after this attack, which would be detected by SHARP. We noticed though that the alarm triggers increase linearly with the number of bits. We only need to correctly identify 75% of the bits to successfully recover the private exponent, so we can actually stop before reaching the threshold of 2000 alarm triggers and still have a successfull attack.

The results of executing this attack for varying amounts of noise during cache loads can be replicated by running

```
python3 process-shared-spy-attack.py
```

It assumes address for multiply is 0x4016dc and 0x4014e3 for square.

Results can be processed with

```
python3 process-shared-spy-attack.py
```

6 Conclusion and Future Work

In this work we implemented the SHARP protocol for a high-level deterministic simulator and also as a pintool. We implemented two cache side-channel attacks that recover the bits of an encryption key for the RSA algorithm. The experimentation shows a proof-of-concept for the attacks, with some level of noise added to the pintool. To make the attacks more enticing, it would be important to implement them on a multi-core cycle-level simulator. This would add more noise to the system and bring the attacks closer to real-world implementation. This requires significant additional work, including empirically determining spy wait times, instruction addresses in the shared L3, and L2/L3 approximate miss times.

In the project proposal 75% was to implement SHARP in a simulated environment and evaluate the protocol, 100% was to design attacks to bypass SHARP, and 125% involved evaluating more mitigation techniques beyond SHARP. We feel we landed comfortably in the 100% range. We were very happy with the two attacks we came up with, and decided to spend more time on them than implementing SHARP in a more sophisticated simulator. The largest setback was realizing the simulators used in the relevant papers [4, 3] had little developer support. We reached out to the author of SHARP but they were not able to share the source code they used for experimentation. As such, we opted for a solution that hacked spies into pintool. It added some noise, taking the experimentation a level above the deterministic simulator we implemented in Python.

Source code can be found at
`https://github.com/icemonster/mitigatingCacheSideChannels`.

References

[1] Matthew Campagna and Amit Sethi. Key recovery method for crt implementation of rsa. *IACR Cryptology ePrint Archive*, 2004:147, 01 2004.

[2] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[3] Dixit Kumar, Chavhan Sujeet Yashavant, Biswabandan Panda, and Vishal Gupta. How sharp is SHARP ? In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

[4] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360, 2017.

[5] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.