



# Malware Guard Extension: Using SGX to Conceal Cache Attacks

170050068 - Killari Ramprasad  
170050081 - Sailendra Bathi Babu  
170050083 - Bonela Mahith  
170050100 - Ramya Narayanasamy



# Summary

- We analyse a software-based side-channel attack from a malicious SGX enclave targeting co-located enclaves.
- In this attack the malware runs on the SGX hardware, abusing SGX protection features to conceal itself. A Prime+Probe cache side-channel attack is performed on a co-located SGX enclave running RSA implementation that uses a constant time multiplicative primitive.
- The attack intends to extract the RSA key using single or multiple traces.
- Demonstrate key recovery through data generated artificially



## Attack Overview

- The attack runs in an SGX enclave and mounts prime+probe attack on the victim running RSA on another enclave.
- The victim provides an API for RSA signature calculation, which the attacker uses for getting private key from the attack.
- The attack can be divided into two stages, online stage and offline stage.
- According to the reference paper, the attack can be completed in around 4 minutes.



## Attack Details - Attack Primitives

- The victim is an unprivileged program that uses SGX to protect an RSA signing application using **mbedTLS** from both software and hardware attackers and provides an API to compute a signature for provided data. Both the RSA implementation and the private key reside inside the enclave.
- The attacker runs an unprivileged program on the same host as that of the victim. The goal of the attacker is to stealthily extract the private key from the victim enclave. Therefore, the attacker uses the API provided by the victim to trigger signature computations.



## Attack Details - Attack Primitives

- To prevent information leakage from function calls, **MBEDTLS** uses the same function for both the square and multiply operation. The function takes two parameters that are multiplied together. For square operation the function is called with the current buffer as both arguments. For the multiply operation, the current buffer is multiplied with a buffer holding the multiplier (base of the exponentiation).
- This buffer is allocated in the calling function using **calloc**. Due to the deterministic behaviour of the **calloc** implementation, the used buffers always have the same virtual and physical addresses. Thus the buffers are always in the same cache sets. The attacker tries to mount a prime+probe attack on the cache sets containing the buffer.



## Attack Details - Attack Primitives

**High Resolution Timer:** Cache side channel attacks usually use rdtsc or rdtscp instructions. These instructions may cause a VM exit in an enclave. A new mechanism is proposed where a global variable is used as the timing source and a separate counter thread is employed to increment the global variable, which works faster.

**Eviction Set Generation:** Accessing two virtual addresses that map to the same DRAM bank but a different row is significantly slower than any other combination of virtual addresses. The algorithm for generating eviction sets exploits this DRAM timing differences that are due to DRAM organization



## Attack Details - Online Phase

### Identifying and Monitoring Vulnerable Sets

We consecutively mount a Prime+Probe attack on every cache set while the victim is executing the exponentiation step. This allows us to log cache misses due to a victim's activity inside the monitored cache set.

Once we have identified a cache set which is used by the exponentiation, we collect the actual traces. The measurement method is the same as for detecting the vulnerable cache sets, i.e., we again use Prime+Probe. As the behavior of heap allocation algorithm is deterministic, the address of the attacked buffer does not change on consecutive exponentiations. And hence, we can collect multiple traces of the signature process.



## Attack Details - Offline Phase

**Pre-processing:** High values raw measurement data correspond to cache misses whereas low values indicate cache hits. Timing measurements have varying sample rate as a cache miss delays the next measurement while cache hits allow more frequent measurements. For simplicity we convert the measurements to a constant sampling rate.

**Partial Key Extraction:** We run a peak detection algorithm and delete duplicate peaks (where the corresponding RSA multiplications would overlap in time) and also delete peaks that are below a certain adaptive threshold, as they do not correspond to actual multiplications.





## Attack Details - Offline Phase

### Final Key Recovery:

Instead of recovering the RSA key from combining the traces (traces contain cache hit and miss data using timestamps), the attack first recovers multiple partial keys from individual traces and then recovers final key using these partial keys. This is because the traces are affected by multiple noise sources and thus, making the trace alignment difficult. This approach is quicker than trace alignment since key recovery is performed on partial keys. We quantify partial key errors using the edit distance.



## Implementation Details

- We realized that implementation of the full attack would not be possible according to the deadlines. Hence, we focussed on finding a feasible step to code.
- The offline part is simpler to implement and demonstrate but the data from the online step is necessary. The online part does not require such data, but needs setting up a lot of parameters.
- The basic step we could do was eviction set generation. Even this requires knowledge on the DRAM configuration, assembly instructions and address mappings and moreover, the same code might not work on all our team's machines due to different configurations.
- We went through a few github repositories like hoping that we could get some idea on how to proceed with eviction set generation. But these were not so easy to understand and not SGX specific



## Implementation details


- Next option was to somehow create our own noisy data to demonstrate the key recovery from partial keys.
- This involved using a python program to randomly corrupt bits of the binary sequence corresponding to the secret key. This artificial trace data was then used as inputs to the key recovery step.
- Full key is recovered bitwise, starting from the most-significant bit. The correct key bit is the result of the majority vote over the corresponding bit in all partial keys. Before proceeding to the next key bit, we correct all wrong partial keys which did not match the recovered key bit. We compute the edit distance over a lookahead window of a few bits and perform the necessary actions to transform one key to another.



## Implementation Details

- We ran our algorithm with 1024, 2048 and 4096 bit keys as input. The results are shown in the table below. The noise rate of the partial keys was 4% in all the cases

Key size	No. of traces	Lookahead size	Time Taken
1024 b	10	10	0.6 seconds
2048 b	15	10	2 seconds
4096 b	20	20	12 seconds



```

input : keys: boolean[], lookahead: int
output: key: boolean[]

key  $\leftarrow$  [];
i  $\leftarrow$  0;
while True do
  | keybit  $\leftarrow$  majority(keys, i);
  | if keybit =  $\perp$  then
  | | return key;
  | end
  | key[i]  $\leftarrow$  keybit;
  | correct  $\leftarrow$  {};
  | wrong  $\leftarrow$  {};
  | foreach k in keys do
  | | if k[i] = keybit then
  | | | correct  $\leftarrow$  correct  $\cup$  k;
  | | else
  | | | wrong  $\leftarrow$  wrong  $\cup$  k;
  | | end
  | end
  | foreach kw in wrong do
  | | actions  $\leftarrow$  {};
  | | foreach kc in correct do
  | | | actions  $\leftarrow$  actions  $\cup$ 
  | | |     EditDistance(kw[i : i + lookahead],
  | | |                    kc[i : i + lookahead]);
  | | end
  | | ai  $\leftarrow$  0;
  | | while kw[i]  $\neq$  keybit do
  | | | action  $\leftarrow$  majority(actions, ai);
  | | | apply action to kw[i];
  | | | ai++;
  | | end
  | end
  | i++;
end

```

```

function majority(set, idx) begin
  | counter[]  $\leftarrow$  0;
  | foreach array in set do
  | | element  $\leftarrow$  array[idx];
  | | increment counter[element];
  | end
  | return element with max. counter;
end

```

Algorithm for recovering final key from partial keys.



# Countermeasures

The countermeasures can be classified into 3 categories based on the modification required

A.Source level:

- Exponent Binding: Signing the message as  $m^{(d+k \cdot \phi(N))}$  where  $k$  is a random binding number
- Bit Slicing : Use only bit operations for computations throughout the algorithm. No lookup tables or branches are used in these algorithms and thus, they are not vulnerable to cache attacks

B.OS level(assuming the OS is benign):

- Enclave Coloring: the cache is partitioned into multiple smaller parts. Each of the parts spans over multiple cache sets, and no cache set is included in more than one part.

However this requires trusting the OS, which is against the core philosophy of SGX



# Countermeasures

## C. Hardware Level:

- Combining Intel with CAT SGX: Any cache sharing between SGX enclaves and the outside as well as co-located enclaves could be eliminated. It would protect co-located enclaves as well as the operating system and user programs against malicious enclaves.
- Secure RAM: An additional secure memory element that resides inside the CPU with data stored within not cacheable. As data from this element is only accessed by one program and is never cached, cache attacks and DRAM-based attacks are not possible anymore.



## Trials and Tribulations

- This attack involves many intricate details on various topics like assembly instructions and DRAM configuration knowledge. Hence, getting a basic understanding of the attack procedure took a lot more time than expected.
- Algorithms mentioned in the research paper skips out on a few helper functions which are considered trivial by the authors. Hence, we had to understand what the missing functions were doing to get a clear understanding of the attack.
- We referred to a lot of git repositories regarding the exact function calls for eviction, timing thresholds etc. But they were not much useful for us as they were not SGX related and use few instructions which are not compatible with SGX.





## Trials and Tribulations

- Keeping track of what works and what does not work among the team was difficult because different members of the team have different machine configurations and hence the code arguments and parameters need to be configured differently.
- Since, the attack is so complicated, we have decided not to implement a full attack and focus on parts which are feasible before the deadline. Offline part of the attack is not so complicated as that of the online part. But requires data from the online part to be provided as inputs. For this we e-mailed two of the authors asking if they are willing to provide the trace data for the project. Thankfully, one of the authors replied with the trace files. But we did not have enough time to go through the data as the reply arrived on the day of project submission(i.e 19th April 2021). We had already started working on creating artificial trace data for the sake of demo.



## Demo

We demonstrate the key recovery process from the partial keys that are normally obtained by removing noise and detecting peaks on the data obtained through online step of the attack. Here, we create the data artificially.



## References

- [A Survey of Published Attacks on Intel SGX](#)
- [Malware Guard Extension: Using SGX to conceal cache attacks](#)
- [Intel SGX explained](#)
- [CacheZoom: How SGX amplifies the power of cache attacks](#)
- [Timing Side-Channel attack on RSA](#)
- [Finding Eviction Sets](#)
- [Trace data obtained from Online Step](#)

---

**Thank You**