EuroSec'17
Belgrade, Serbia

Cache Attacks on Intel SGX

Johannes Götzfried*, Moritz Eckert*, Sebastian Schinzel†, and Tilo Müller*

*Department of Computer Science
FAU Erlangen-Nuremberg, Germany

†Department of Electrical Engineering and Computer Science
Münster University of Applied Sciences, Germany

April 23, 2017

Motivation

SGX aims to guarantee confidentiality and integrity of applications running inside untrusted environments

- Secure containers to protect against higher privileged software
  - including the operating system
- In fact: Only the CPU package is considered trusted

→ SGX assumes a very strong attacker model (local root-level attacker)

Main applications of SGX so far have been cloud-related solutions

- Protect against potentially malicious cloud providers
- Maintain confidentiality and integrity of customers code and data
- Example: Haven and VC3

→ Any information leak violates the security goals of SGX

Related Work

Side channels for SGX enclaves are part of current research

- Xu et al.: Controlled-Channel Attack
  - track memory accesses of enclaves on per-page basis
- Weichbrodt et al.: AsyncShock
  - exploit synchronization bugs such as use-after-free and time-of-check-time-of-use

$\rightarrow$ No publication about cache attacks against SGX so far

We present an access-driven cache attack against a vulnerable AES implementation running within an SGX enclave.

Gladman AES: Initial State

AES Parameters:

- 128 bit input plaintext p
- 128 bit round key $k^{(r)}$ for each round r
- 128 bit internal state s
- 128 bit ciphertext c (state after last round)

Initial state corresponds to plaintext p:

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,0} & p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,0} & p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix}$$

Gladman AES: Round Operation

State after one round can be expressed as follows:

$$\begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[s_{0,j}] \\ S[s_{1,(j+1) \bmod 4}] \\ S[s_{2,(j+2) \bmod 4}] \\ S[s_{3,(j+3) \bmod 4}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j}^{(r)} \\ k_{1,j}^{(r)} \\ k_{2,j}^{(r)} \\ k_{3,j}^{(r)} \end{bmatrix}$$

Includes the four AES steps:

- *SubBytes*
- *ShiftRows*
- *MixColumns*
- *AddRoundKey*

Gladman AES: Tables

First three steps can be replaced by table-lookups and XOR-operations:

$$T_0[x] = \begin{bmatrix} S[x] * 02 \\ S[x] \\ S[x] \\ S[x] * 03 \end{bmatrix} \qquad T_1[x] = \begin{bmatrix} S[x] * 03 \\ S[x] * 02 \\ S[x] \\ S[x] \end{bmatrix} \qquad \cdots$$

The state is calculated as follows:

$$\begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = T_0[s_{0,j}] \oplus T_1[s_{1,(j+1) \bmod 4}] \oplus T_2[s_{2,(j+2) \bmod 4}]$$

$$\oplus T_3[s_{3,(j+3) \bmod 4}] \oplus \begin{bmatrix} k_{0,j}^{(r)} \\ k_{1,j}^{(r)} \\ k_{2,j}^{(r)} \\ k_{3,j}^{(r)} \end{bmatrix}$$

Gladman AES: Last Round

Within the last round, *MixColumns* is missing:

$$T_4[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \\ S[x] \end{bmatrix}$$

Thus, the ciphertext is computed as follows:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = (T_4[s_{0,j}])_{\vec{1}} \oplus \left(T_4[s_{1,(j+1) \bmod 4}]\right)_{\vec{2}} \oplus \left(T_4[s_{2,(j+2) \bmod 4}]\right)_{\vec{3}}$$

$$\oplus \left(T_4[s_{3,(j+3) \bmod 4}]\right)_{\vec{4}} \oplus \begin{bmatrix} k_{0,j}^{(10)} \\ k_{1,j}^{(10)} \\ k_{2,j}^{(10)} \\ k_{3,j}^{(10)} \end{bmatrix}$$

Neve and Seifert's Elimination Method

Access-driven approach against the *last round* of AES:

$$c_{i,j} = k_{i,j}^{(10)} \oplus \left(T_4[s_{i,(i+j) \bmod 4}] \right)_i$$

If we would know which part of the table has been accessed, we could deduce key bytes:

$$k_{i,j}^{(10)} = c_{i,j} \oplus \left(T_4[s_{i,(i+j) \bmod 4}] \right)_i$$

$\rightarrow$ Difficult to deduce exactly accessed bytes

Neve and Seifert's Elimination Method

Elimination Method:

$$k_{i,j}^{(10)} \notin c_{i,j} \oplus \neg[T_4 \text{ outputs}]$$

Key candidates are excluded:

- *Prime&Probe* to get non-accessed cache lines
- $\neg[T_4$ outputs$]$ refers to all T_4 byte values within such lines
- Discard all candidates for $k_{i,j}^{(10)}$ which map to such lines

$\rightarrow$ Repeat with different plaintexts until only one or few key bytes remain
$\rightarrow$ Due to AES key schedule redundancies $k^{(10)}$ is sufficient to get k

L1-Cache Associativity

L1 cache is split into data and instruction cache (Intel Core i7-6700HQ):

- size of 32KB
- 8-way associative
- 64 byte cache lines

Priming 8-way associative cache:

- Neve and Seifert describe their approach for direct-mapped caches
- Two cache lines of T_4 could be stored within the same cache set (unlikely, because T_4 needs 16 sets and 64 sets are available)
- Access to a cache set can be treated like an access to a cache line
- Need to access every cache set 8 times to fill every line within the set

Identifying Evictions using PMC

Our attacker model includes local root-level attackers:

- Use *Performance Monitoring Counters* (PMC) to count cache misses
- More accurate and reliable than timing information
- Used from attacker thread outside of the enclave

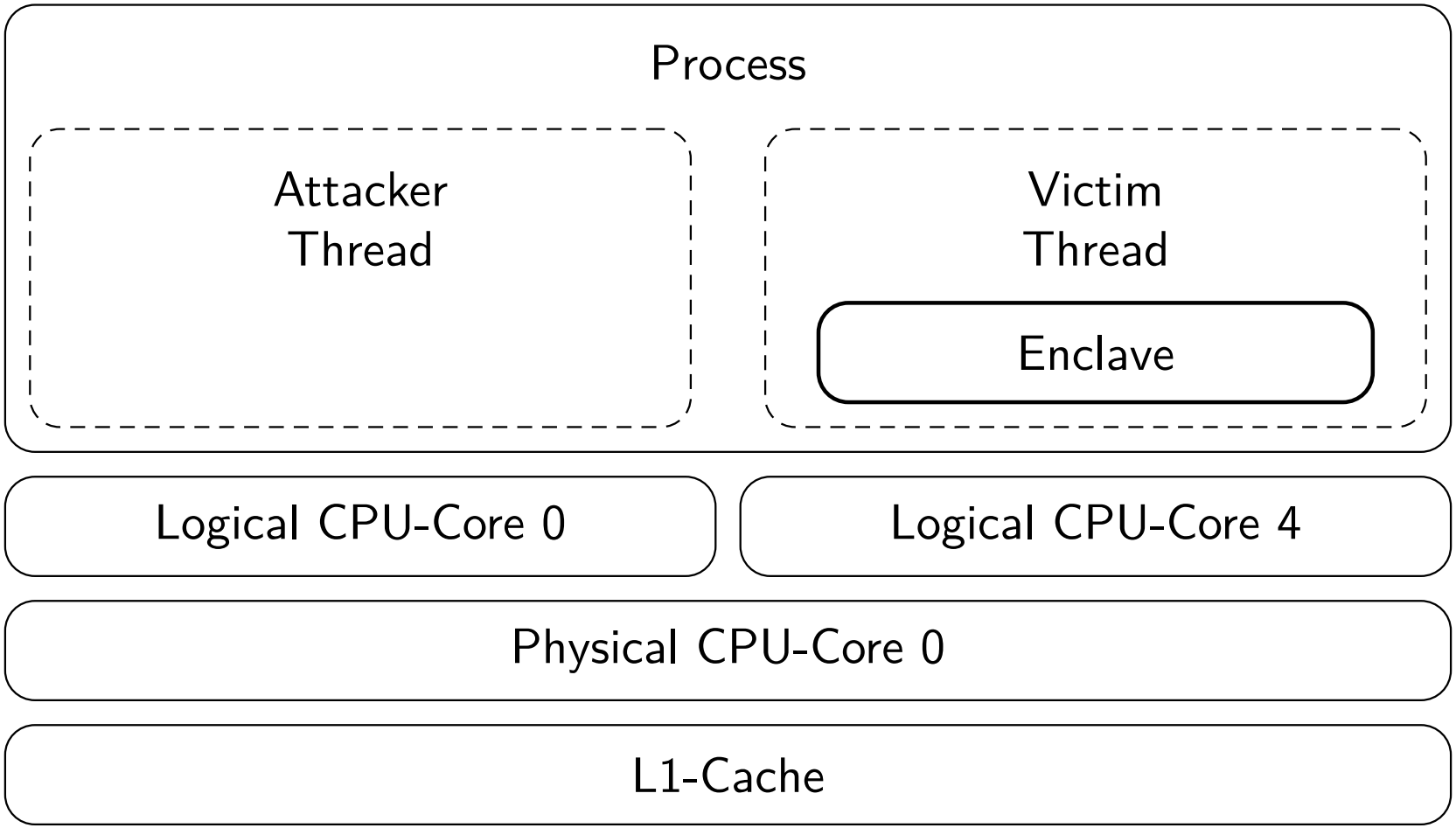
Probing 8-way associative cache (for each ciphertext byte):

1. Read PMC count with `readpmc`
2. Access desired line
3. Read PMC count again and return difference

$\rightarrow$ Repeat 8 times to catch all evictions

$\rightarrow$ If one difference is > 0 the corresponding line for T_4 has been accessed

Attack Setup



Attack Details

Attacker and Victim Thread:

- Process context switches would trigger L1 cache flushes
- Attacker and victim thread share the same process
- Threads are pinned to different logical CPUs mapped to the same physical core (hyperthreading)
- Easily possible with `sched_setaffinity()` system call

Communication with Shared Memory:

- ECALLs and OCALLs introduce noise
- Shared memory for plaintext and ciphertext
- Control flags to start and stop the encryption

Performance

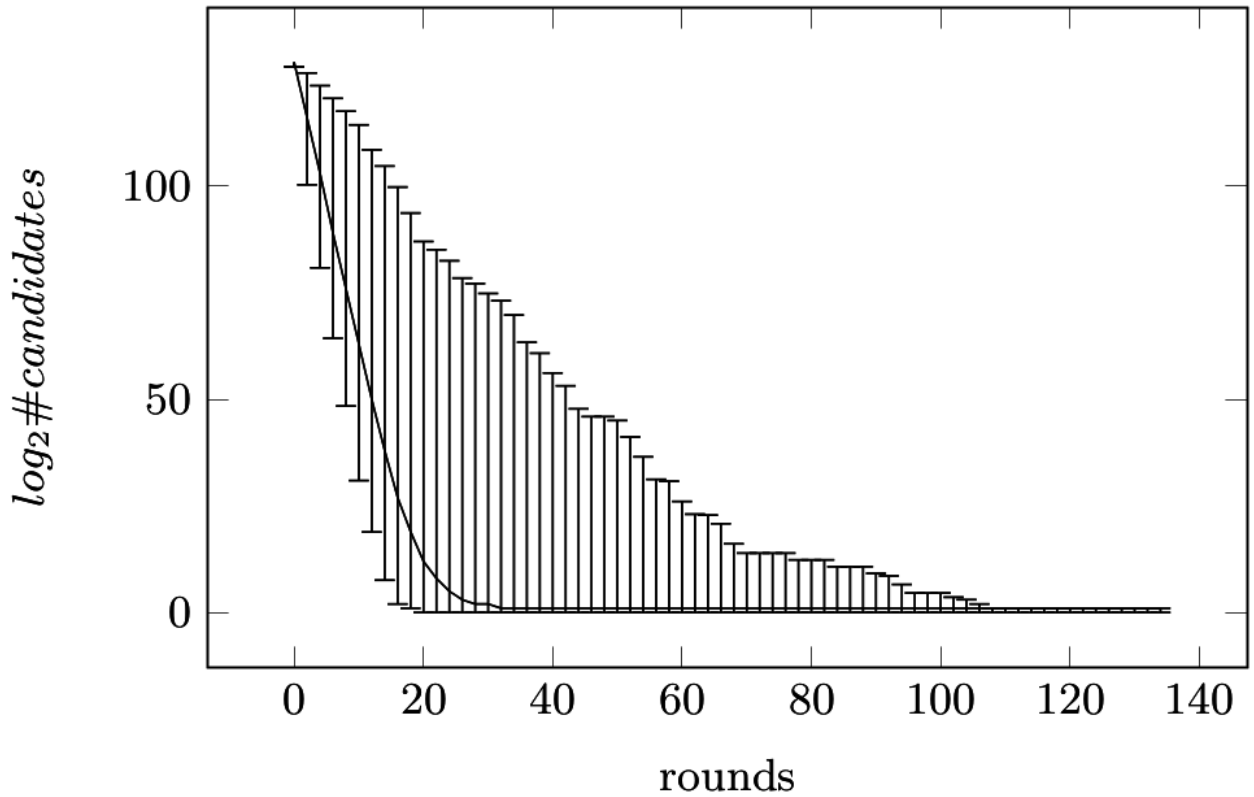
System:

- Intel Core i7-6700HQ CPU running at 2.60GHz
- 16GB of RAM
- Ubuntu Linux 14.04 LTS (Trusty Tahr)

Evaluation Setup:

- 5000 runs with different keys
- Measure the required time
- Measure the amount of required elimination rounds (number of needed ciphertext blocks)

Amount of Required Elimination Rounds



→ On average 30 elimination rounds are needed

→ On avarage $30 \cdot 16 = 480$ encryptions are necessary

→ Average time of less than 10 seconds

Practicability

Cipher implementation:

- Needs (of course) to be vulnerable
- We use Gladman AES of an old version of OpenSSL (version 0.9.7a)
- Interestingly the Intel SGX SDK for Linux does not use AES-NI (but textbook AES is hardened against cache attacks)

Anti Side-channel Interference (ASCI) bit:

- Our attack is run in debug mode
- Intel provides possibility to disable PMC counters
- Only affects threads running in enclave mode

$\rightarrow$ Attack should still work

Practicability

Artificial isolation of last round:

- Control flags are not practical
- Process context switches and enclave exits cause too many evictions
- Need to go to higher-level cache (L2 or L3)

$\rightarrow$ Practical problem of our attack

Conclusion

First cache attack on software running within an Intel SGX enclave

- Access-driven cache attack
- Deterministically derives the key within an average of < 10 seconds

$\rightarrow$ SGX does not protect against cache attacks

$\rightarrow$ Developers need to take care themselves

Thank you for your attention!

Further Information:

https://www1.cs.fau.de/sgx-timing