Compiling python to a side-channel protected binary

Albert Dayn, Sergey Frolov

Intel SGX

https://newsroom.intel.com/news/intel-microsoft-enterprise-blockchain-service/

Intel SGX features

- Allows applications to protect portions of code and data(an “**enclave**”) from manipulation, even from potentially compromised OS.
- Code and data are encrypted in RAM and protected from disclosure. This also enables (mutually) distrusted computation.
- Provides the ability to prove that a remotely executing application does what it says it does (hasn’t been tampered with)

Intel SGX features

- Allows applications to protect portions of code and data(an “**enclave**”) from manipulation, even from potentially compromised OS.
- Code and data are encrypted in RAM and protected from disclosure. This also enables (mutually) distrusted computation.
- Provides the ability to prove that a remotely executing application does what it says it does (hasn’t been tampered with)

But there are serious issues with SGX: side-channels.

- Many are known: timing, cache, branch shadowing.
- Were shown in prior research to be very problematic: *“We extract the full RSA private key in an automated attack from 11 traces within 5 minutes”*.

Related Work

Many papers are exploring compiler defenses against specific side-channels.

[1] Gruss, Daniel, et al. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory." (2017).
[2] Lee, Sangho, et al. "Inferring fine-grained control flow inside SGX enclaves with branch shadowing." *arXiv preprint arXiv:1611.06952* (2016).
[3] Schwarz, Michael, et al. "Malware guard extension: Using SGX to conceal cache attacks." *arXiv preprint arXiv:1702.08719* (2017).
[4] Cauligi, Sunjay, et al. "FaCT: A Flexible, Constant-Time Programming Language." *Cybersecurity Development (SecDev), 2017 IEEE*. IEEE, 2017.
[5] Rane, Ashay, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution." *USENIX Security Symposium*. 2015.
[6] Garcıa, Cesar Pereida, and Billy Bob Brumley. *Constant-time callees with variable-time callers*. IACR Cryptology ePrint Archive, Report 2016/1195, 2016.
[7] Almeida, José Bacelar, et al. "Verifying Constant-Time Implementations." *USENIX Security Symposium*. 2016.

What we did

We implemented subset of Python language (forked homework and reworked if and while statements to work without polymorphism) that is

- Protected from shadow branching attack. Technique was borrowed from previous work.

- Protected from timing and cache side-channels. Expansion on top of previous technique.

Branch shadowing attack [2017]

Generally attacks are executed right after interrupting enclave and attempt to infer something about what just happened inside of an enclave.

Branch shadowing attack infers whether a particular branch in enclave was taken by “shadowing” its branch history.

Branch shadowing attack [2017]

To determine whether an enclave branch at, e.g., `0x0000`**`400530`** was taken, local attacker crafts an out-of-enclave branch at e.g. `0xffff`**`400530`**.

Since only lower 31 bits of branching instruction address are used to locate an entry in Branch Target Buffer table, those 2 branches collide. After measuring elapsed cycles and/or checking LBR table for out-of-enclave branch, one can learn whether corresponding branch was taken in enclave.

Timing side-channels

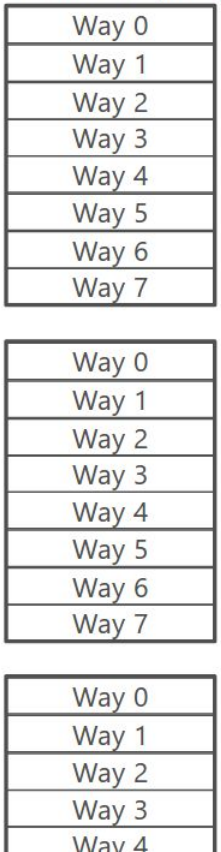
Underlying reason is very simple.

```
if secret-value:
    do_things()
else:
    // can take more or less time
    // difference will be noticeable to local adversary
    do_other_things()
```

Attack example: Prime+Probe

Slide borrowed from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

Cache sets (L1-I)



Attacker

L1 caches

64 cache sets

8 ways per cache set

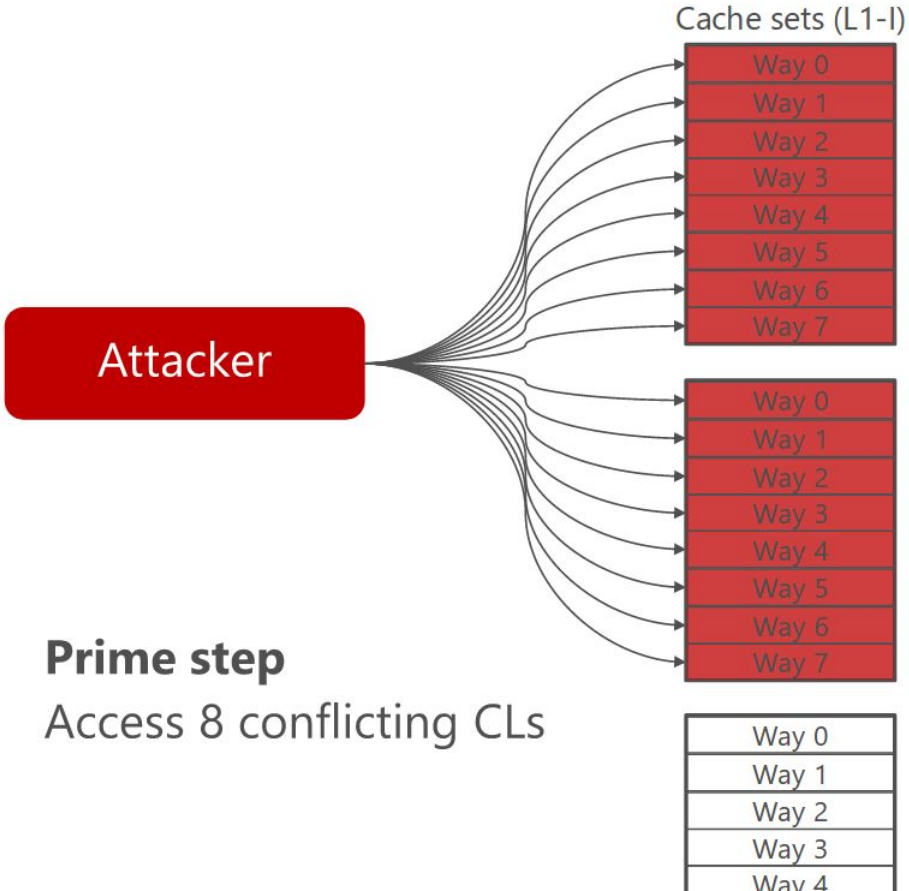
64 bytes per way/cache line

Victim

```
if (secret) {
    // ...
}
else {
    // ...
}
```

Attack example: Prime+Probe

Slide borrowed from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

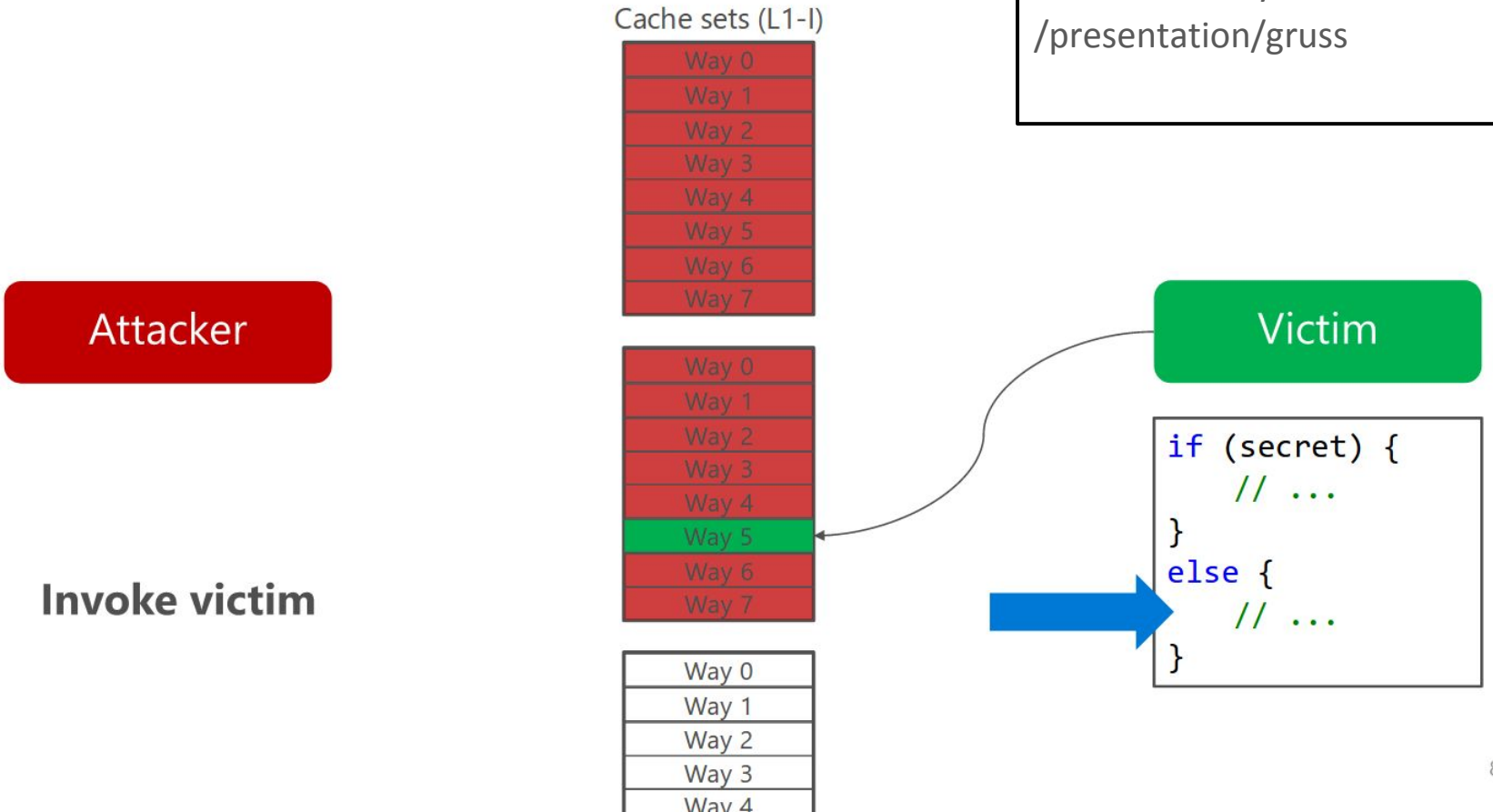


Victim

```
if (secret) {
    // ...
}
else {
    // ...
}
```

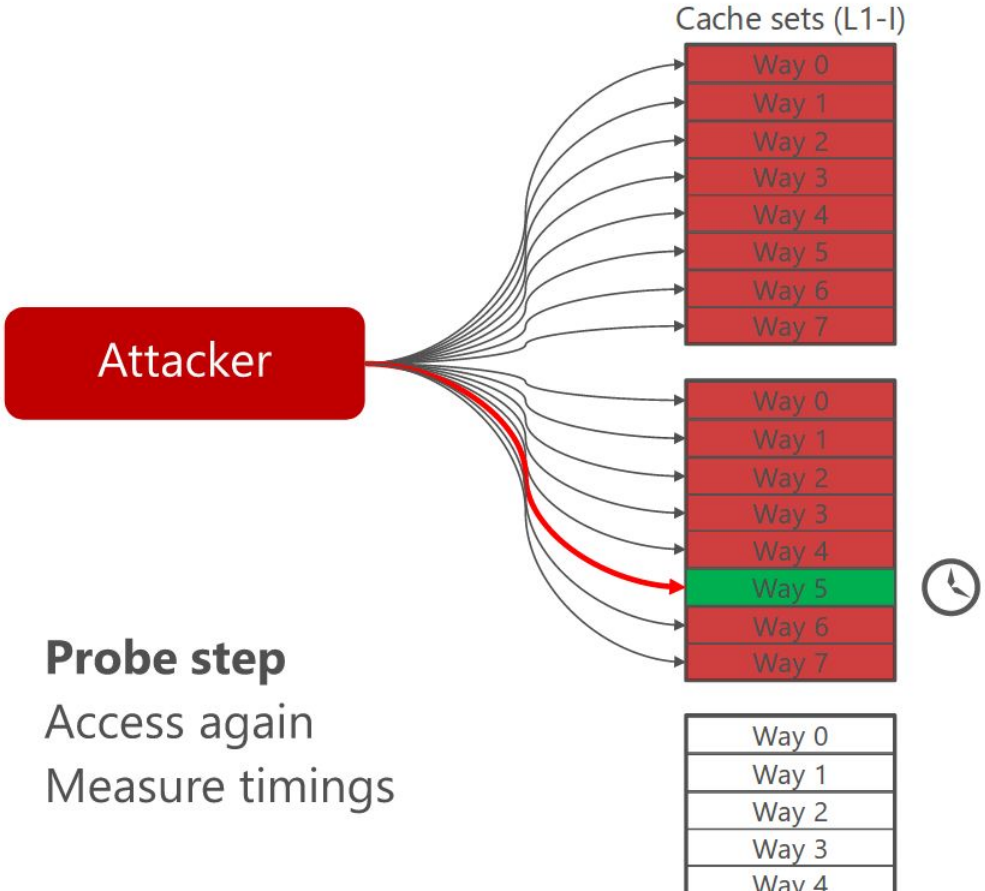
Attack example: Prime+Probe

Slide borrowed from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss



Attack example: Prime+Probe

Slide borrowed from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss



Probe step

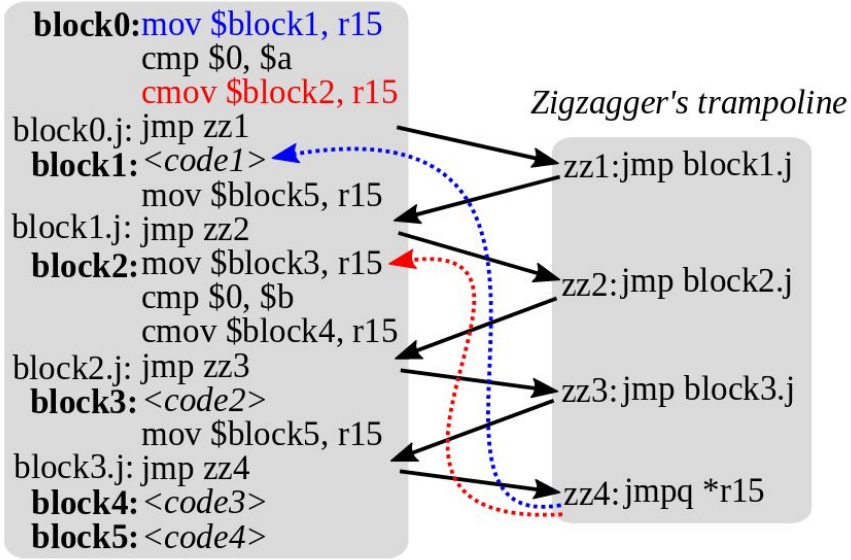
Access again

Measure timings

Victim

```
if (secret) {
    // ...
}
else {
    // ...
}
```

Zigzagger [2017]



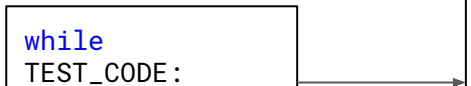
(b) The protected code snippet by Zigzagger. All branch instructions are executed regardless of a and b variables. An indirect branch in the trampoline and CMOV instructions in the translated code are used to obfuscate the final target address. Note that r15 is reserved in Zigzagger to store the target address.

Protects against shadowing attack since indirect jumps are always reported as correctly predicted in LBR

Protecting against Branch Prediction Side-Channels

- We implemented Zigzagger for Python on loop control flow
- Only unconditional and indirect jump instructions used for looping
  - LBR always shows indirect jumps as correctly predicted.

```
while
TEST_CODE:
    BODY_CODE
REST
```



```
jmp test_label

zz1:
    jmp body.j

zz2:
    jmp *%ebx

test_label:
    TEST_CODE
    mov body_label, %ebx
    cmpl $0, test_var
    cmove end_label,
%ebx
    jmp zz1

body_label:
    BODY_CODE
    mov test_label, %ebx

body_label.j:
    jmp zz2

end_label:
REST
```

Improving on Zigzagger

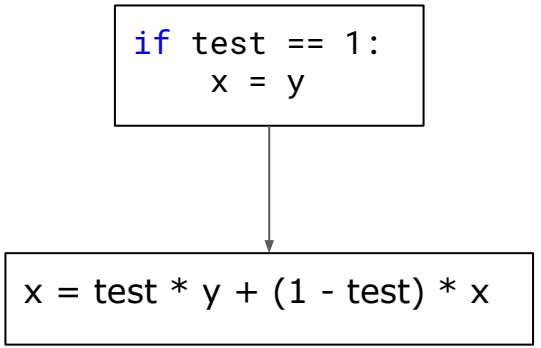
- Zigzagger only protects against Branch Shadowing Attacks
- Timing and cache side-channels remain a problem
- Solved branch prediction leakage by turning conditional jumps (e.g. `jne`) into a trampoline consisting only indirect jumps
- To solve time leakage, we have removed even indirect jumps, using the constant time `cmov` instruction.

Contemporary protections vs Timing Side-Channels

- Manually execute same amount computation on both branches, using a variable to indicate if it's valid
  - Not guaranteed to work due to compiler optimizations
  - Bug prone

```
if (mbedtls_mpi_cmp_abs(A, N) >= 0) {
  mpi_sub_hlp(n, N->p, A->p);
  i = 1;
}
else { // dummy subtraction to prevent timing attacks
  mpi_sub_hlp(n, N->p, T->p);
  i = 0;
}
```

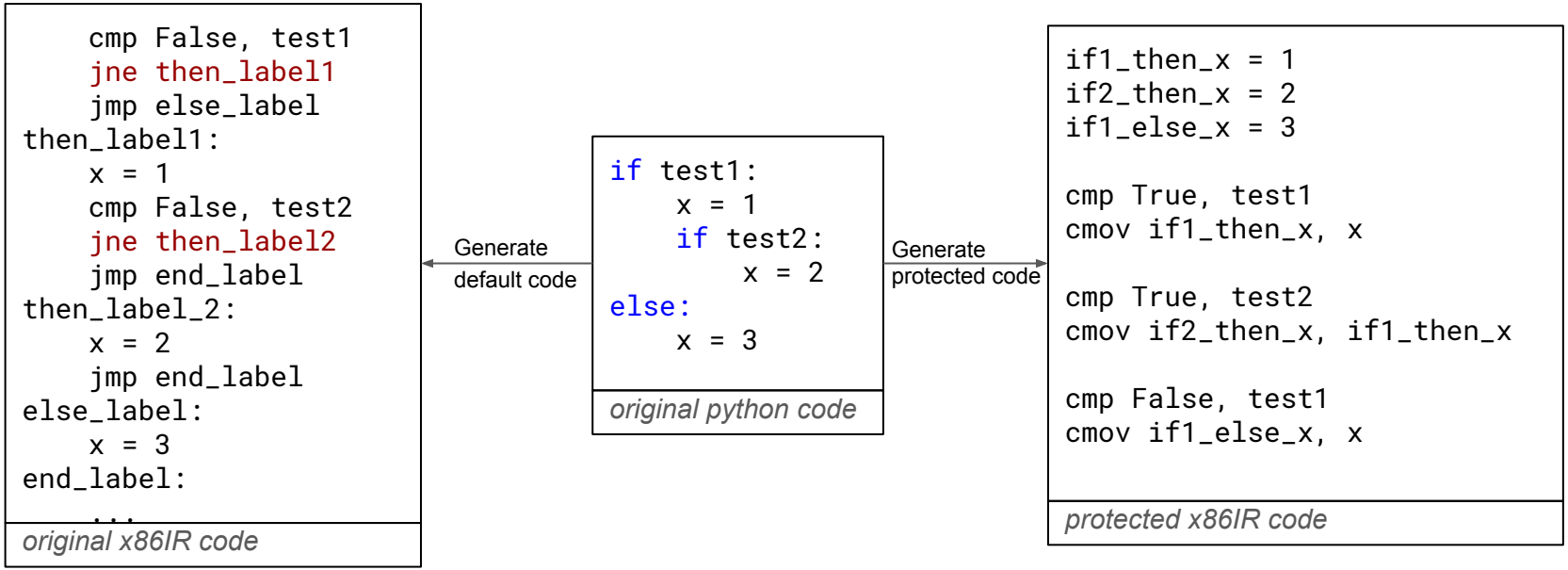
- Restructure algorithm to entirely eliminate all branches
- Math tricks help sometimes
  - More robust than manually executing both branches
  - Not always possible



Protecting against Timing Side-Channels

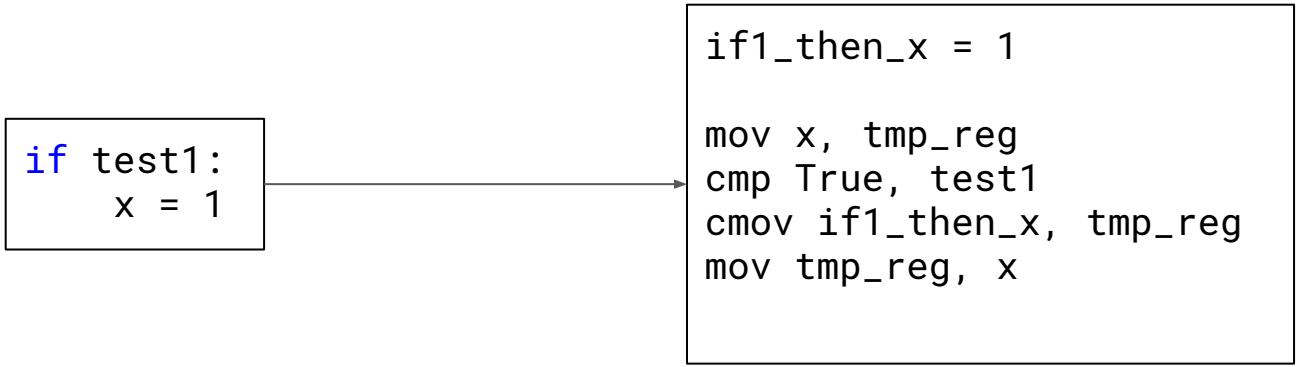
- Instead of indirect jumps, we use only conditional move, (cmov) instructions for branching
  - Not sensitive to branch shadowing since cmov isn’t predicted
  - Not sensitive to timing attacks since we produce straight line assembly code
- Required 2 new passes on the x86 IR
  - Pass 1: Removing variable collisions due to executing both branches
    - Inject new variables at each level of a nested if statement, recursively
    - Rename variables in each branch to their temporaries
  - Pass 2: Remove conditional control flow
    - Replacing conditional jumps with straight line assembly code and conditional moves

Protecting against Timing Side-Channels

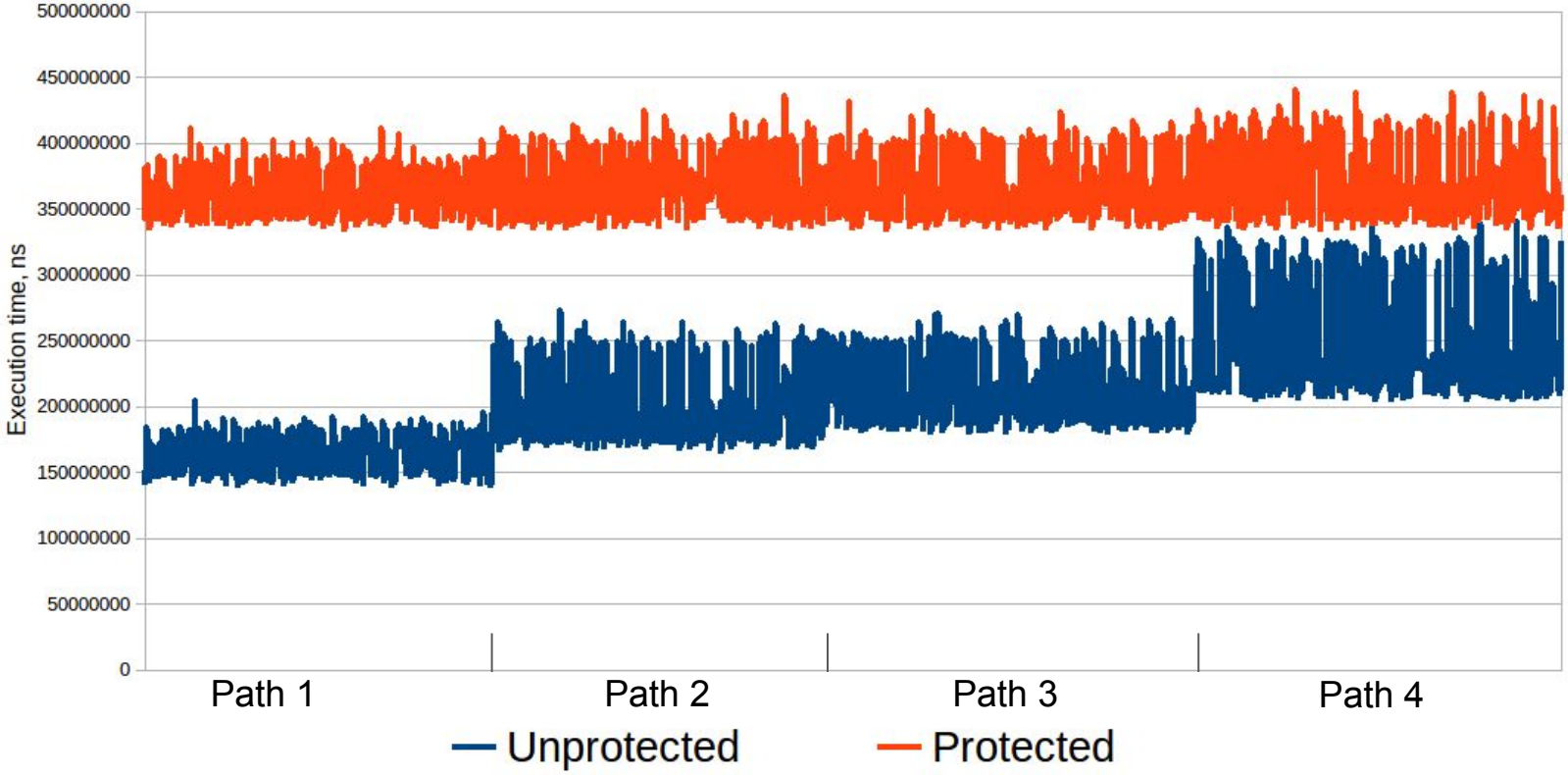


Protecting against Cache Side Channels

- Always read and write memory
- Selectively change result before writeback using only registers



Evaluation


500000000
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
Execution time, ns
Path 1
Path 2
Path 3
Path 4
Unprotected
Protected

Thanks!

Intel SGX usage

Used for

- Secure cloud computation on untrusted servers
- Private data recovery (Signal messenger app)
- Shared computation on distributed data, where all parties distrust each other
- Password Managers
- AVs
- “To add new levels of privacy and confidentiality to blockchain transactions” © Intel

Limitations

- Slower, since all branches must be executed
- No functions with side-effect (such as print) can be performed in branches while maintaining semantics

Future Work

Extend Python with this work as a type of function using decorators

Add ability to distinguish between public and private vars

Array[i_secret] = 123 <- Protect with SMT solver, like in FACT

Formally Verify it

Do a wider study on `cmov` to check whether it's constant.

Reduce amount of reserved registers