Compiling python to a side-channel protected binary

Sergey Frolov
University of Colorado
Boulder, CO, USA
sergey.frolov@colorado.edu

Albert Dayn
University of Colorado
Boulder, CO, USA
albert.dayn@colorado.edu

Abstract

There are numerous side-channel based attacks on protections, guaranteed by Intel SGX trusted execution. Prior research has shown that side-channels may enable a local attacker to extract information from the secure enclave. We address issues with cache, timing, and branch shadowing through a combination of known and brand new techniques to compile Python to a protected binary. We show that proposed protections efficiently eliminate side-channels.

CCS Concepts • **Software and its engineering** → **Compilers**; • **Security and privacy** → *Software and application security*;

Keywords Compilers, Intel SGX, side-channels, secure code generation

ACM Reference Format:
Sergey Frolov and Albert Dayn. 2017. Compiling python to a side-channel protected binary. In *Proceedings of CSCI 4555/5525 and ECEN 4553/5523 (Compiler Construction)*. ACM, New York, NY, USA, 6 pages.

1 Introduction

Intel Software Guard Extensions (SGX) enables trusted execution with a new set of CPU instructions and platform enhancements, allowing a dramatic increase privacy, security and sometimes scalability of applications, when used correctly. Specifically, it allows applications to protect portions of code and data (an enclave) from manipulation, even from a potentially compromised OS. The enclave is protected from disclosure, as data is always encrypted in RAM. Intel SGX also allows users to easily perform operations that are either very computationally expensive in software-only solutions (such as Private Information Recovery [11] and mutually distrusted computation on shared data), or simply impossible (remote attestation, which allows client to prove to a remote entity the integrity of software running inside of enclave).

However, side-channels have proved to be a very serious limitation of SGX. Several attacks on SGX are known, including timing and cache-based side-channels, as well as branch shadowing. While some algorithms may be immune to those attacks due to inherent lack of any branching, there are cases in which side-channels would work extremely effective, e.g. allowing to extract full RSA private key [15] in an automated attack in under 5 minutes.

Whenever application developers are accept the complexities of using SGX, they rely on its protections and assumptions heavily, and when those assumptions are broken even in a minor way, the security of whole application may be compromised. For this reason, it is crucial to find effective means to defeat those attacks in a manner, without making the solution too cumbersome to potential users. Specifically, users need to be able to write idiomatic code and ensure it compiles to a constant time binary. This paper describes known SGX side-channels, and presents effective defences. Description of proposed defences design is followed by performance evaluation, which shows the effectiveness of our suggested timing side-channel protections.

2 Background

Intel SGX allows applications to put portions of code and data into a so-called secure enclave, and protect it from manipulation from any local attacker, including a malicious or compromised operating system. As everything in this secure enclave is encrypted, Intel SGX also effectively protects data from disclosure. Such strong protections may prove challenging to ensure, however, since local attackers have various tools at their disposal to mount attacks. Since the secure enclave shares the processor with the attacker, information may leak through side-channels such as timing, processor cache, Branch Target Buffer, and other shared physical resources. Even though there are usually no direct ways to probe those resources, the attacker may do this indirectly. Also note that it is within a local attackers power to interrupt the secure enclaves execution at will, which makes attacks easier to perform in practice.

Compiler Construction, Fall 2017, Boulder, CO USA

A proper understanding of our side-channel protections requires knowledge of corresponding attacks, and we proceed to describe previously introduced exploitation techniques in this section. Generally, attacks are executed right after interrupting enclave and attempting to infer something about what just happened inside of an enclave. For the attacks that we protect against, the adversary wants to use a particular side-channel to determine which branch (at the assembly level) was taken.

Timing attack.[9] A timing attack is relatively easy and intuitive. Suppose we have 2 or more branches, which take different amount of time to execute. A local attacker may measure time spent on execution and derive which branch was taken, allowing them to infer the value of the tested conditional.

Cache attack.[12] To carry out this attack, the adversary fills the cache with random data, then lets the enclave continue execution. After secure enclave executes the target branch, attacker would interrupt the enclave. Now, the attacker is able to probe the cache, by accessing the same random data and measuring the time it took to fetch it. Form this, they can determine which cache lines were evicted by the enclave. Since the cache line used depends on the memory address, which varies based on the variables used in different branches, the attacker may infer which branch was taken in the enclave.

Branch shadowing attack.[10] This attack allows local attacker to determine whether an enclave branch at a certain address, e.g., `0x0000422410`, was taken, a by crafting shadowing out-of-enclave branch at e.g. `0xffff422410`, which shares lower 31 bits of the address. Since only those last bits are used to locate an entry in Branch Target Buffer table, enclave branch will collide with out-of-enclave branch. After attacker executes the out-of-enclave branch, theyd be able to measure elapsed cycles and checking LBR table, see whether the out-of-enclave was mispredicted, and learn whether corresponding branch was taken in enclave. Attack on memory access patterns. The operating system has access to encrypted RAM, and so may be able to derive memory access patterns from page faults. Suppose the enclave accesses certain index of an array, and will have to go to memory for this. Despite the fact that RAM, used by enclave, is encrypted, operating system still has access to memory accesses and page faults, and may be able to infer the index of an array, enclave tries to access. This will be problematic, since index of an array may be precisely the data application wants to hide, for example this was the case in private contact discovery for Signal messenger [11]. This attack is not covered in this project and left for future work.

3 Implementation

Definitions: *conditional jump instruction* refers to instructions which may or may not perform a jump to a static address (e.g. `jne LOOP`). *indirect jump instruction* refers to instructions that always jump to a dynamic address (e.g. `jmp *%ebx`). *unconditional jump instruction* refers to instructions that always jump to a static address (e.g. `jmp LOOP`).

$$\begin{array}{rcl}
\textit{program} & ::= & \textit{effectStmts} \\
\\
\textit{effectStmts} & ::= & \textit{effectStmt}\ \textit{effectStmts} \\
 & & \mid \epsilon \\
\\
\textit{effectStmt} & ::= & \text{“input” “(” “)”} \\
 & & \mid \text{“print”}\ \textit{expr} \\
 & & \mid \textit{simpleStmt} \\
\\
\textit{simpleStmts} & ::= & \textit{simpleStmt}\ \textit{simpleStmts} \\
 & & \mid \epsilon \\
\\
\textit{simpleStmt} & ::= & \text{“if”}\ \textit{expr}\ \text{“:”} \\
 & & \qquad \textit{suite} \\
 & & \text{“else” “:”} \\
 & & \qquad \textit{suite} \\
 & & \mid \text{“if”}\ \textit{expr}\ \text{“:”} \\
 & & \qquad \textit{suite} \\
 & & \mid \text{“while”}\ \textit{expr}\ \text{“:”} \\
 & & \qquad \textit{suite} \\
 & & \mid x\ \text{“=”}\ \textit{expr} \\
\\
\textit{suite} & ::= & \text{INDENT}\ \textit{simpleStmts}\ \text{DEDENT} \\
\\
\textit{expr} & ::= & \textit{expr}\ \text{“if”}\ \textit{expr}\ \text{“else”}\ \textit{expr} \\
 & & \mid \textit{expr}\ \text{“+”}\ \textit{expr} \\
 & & \mid \text{“not”}\ \textit{expr} \\
 & & \mid \text{“-”}\ \textit{expr} \\
 & & \mid n \\
 & & \mid x
\end{array}$$

Figure 1. The subset of Python implemented in our prototype compiler. Notice the distinction between simple and effect statements, and the type used in a suite.

We have implemented a constant time compiler which hardens a subset of Python against timing, cache, and branch shadowing side-channels. The specific language fragment is defined below. Notice that *effect_statements* (statements which perform observable side effects other than memory access) are allowed only outside of `if` statements, otherwise the semantics of the language would change as a result of our transformations. We feel this is an acceptable trade off for constant time compilation, as conditional side effecting statements expose an unsafe side-channel themselves.

Our side-channel protections consist of three separate phases, which build on the Zigzagger [10] mitigation. Zigzagger is a method of compiling conditional control flow using only conditional move and indirect jump instructions, meaning no branch prediction is done by the cpu, preventing the leakage of branch history outside of SGX through branch shadowing. The first protection we explored used this method unaltered to compile `while` loops to an assembly equivalent without using leaky conditional jump instructions to protect while loops against branch shadowing. The second protection improves upon Zigzagger by transforming `if` statements into an assembly equivalent with no conditional or indirect jump instructions to protect against timing attacks. The third protection prevents memory from being conditionally read in cmov instructions to protect against cache side-channels. Authors of Zigzagger mention that measuring branch prediction/misprediction penalties based on timing is too inaccurate to distinguish fine-grained control-flow changes, but we have decided to close this side-channel none the less, as viable attacks have already been presented and proved viable, which are referenced in Related Work.

3.1 Branch Shadowing defense

Using the Zigzagger mitigation, we perform transformation on while loops, as depicted on Figure 2.

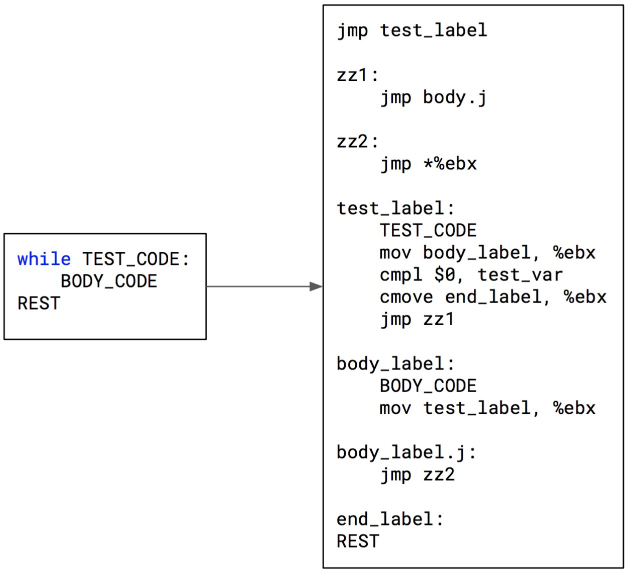
We reserve the `ebx` register to store the address we want the Zigzagger trampoline to jump to next (instead of `r15` as used in Lee, Sangho, et al. [10]). Execution begins by jumping over the inserted Zigzagger trampoline and into the test code. Once the test is performed, we initialize `ebx` to the body code, assuming it will be executed. If the result of the test was `false`, the end label is instead moved into `ebx`. We then jump to the start of the trampoline, and, as described in Zigzagger, we go through a series of unconditional jumps until we reach the final indirect jump at `zz2`. This sends us either to the end label initially put into `ebx`, or to the loop body. In the case of the end label, we continue execution of the rest of the program normally. In the case of the body label, we execute the loop body code and move the test label into `ebx` to go through the loop again, then go through the trampoline which eventually ends with the indirect jump to the address in `ebx` again.

This transformation allows us to protect our produced assembly code from the branch shadowing attack described above, as neither the `cmov` nor the `jmp` instructions leak information into the LBR of an attackers shadow code, while preserving the semantics of the `while` statement.

This still exposes timing side-channels for while loops, however, and while this may not be avoidable in the case of loop based code that necessarily needs to change execution time based on a secret value, we found we could do better for `if` statements.

3.2 Timing Side-Channel defense

The fundamental problem of selectively executing instructions still exists in Zigzagger. The general solution to this is to either pad both branches to have the same number of instructions or to use mathematical equivalents (e.g. `if test == 1: x = y` is equivalent to `x = test * y + (1 - test) * x`) to manually turn specific, known, non-constant time operations into straight-line assembly. It seemed to us that due to small variations in the timing of specific instructions and differences in processor implementations, it was not sufficient to pad either the `then` or `else` branch of an `if` statement to ensure they have the same predicted runtime, we needed to remove conditional control flow all together. Similarly, we found the second approach of using known equivalences problematic as it was a heuristic, not a general purpose algorithm, so we wouldnt be able to guarantee it would always provide a valid transformation to any given



```
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
```

Figure 2. The transformation applied by our compiler on while loops, based on the mitigation on if statements in Lee, Sangho, et al.

input program. We accomplished the removal of conditional control flow by creating a transformation that would always execute the `then` and `else` branches while maintaining program semantics. The specific transformation consists of 2 steps as implemented in our prototype compiler:

1. Rename variables written in the branch to avoid collisions, keeping track of the renamings for step 2. Repeat for both then and else branches.
2. Replace if statements with the following, in sequence:
    a. Save test - Save result of the test expression in case it relies on variables that will be overwritten before conditional writebacks
    b. Then inits - initialize all renamed variables in then branch to their old var
    c. Then code - the code to be executed if test evaluates to true
    d. Else inits - initialize all renamed variables in else branch to their old var
    e. Else code - the code to be executed if test evaluates to false
    f. Conditional moves for then branch - write back renamed variable to old name
    g. Conditional moves for else branch - write back renamed variable to old name

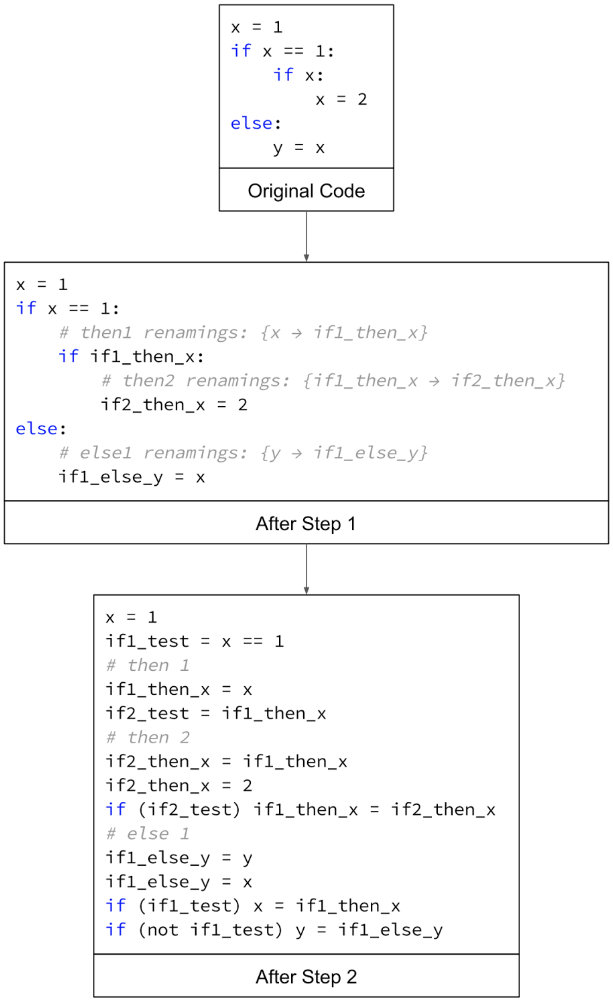
Both steps must be performed recursively to nested ifs after being applied to the outer statement. Example of code, generated by this transofrmation may be seen on Figure 3 Conditional moves are represented by the following pseudo code: `if (test) dest = src`

3.3 Cache Side-Channel defense

Removing conditional control flow results in an almost complete solution to the issue of an attacker knowing what is executing since variables arent selectively read or written based on a test condition. The only stumbling block is when the first operand of a conditional move (the source) comes from memory. In that case, the processor only requests the value from the cache if the condition is met. This can be fixed rather easily with some overhead by always reading the left operand, selectively changing it, and always writing it back.

4 Evaluation

Figure 4 the security of our compiler, we compiled a program consisting of a loop iterating over two if statements (resulting in 4 possible paths of execution). The program would select the test condition for each if statement randomly on each run, independent of any manual input. The above figure shows runtime in nanoseconds on the y-axis of both the output binary from the unsecured Python compiler in blue and the constant time compiler in red. The timings of both binaries have been sorted into the 4 paths, 1 being no branches taken, 2 and 3 being either the first or second if statement executed, and 4 being both branches taken. With no protections, the path taken is clear after only a few iterations of the program, revealing the secret value that determined the test of each if statement. A notable result here is that paths 2 and 3 differ by only one instruction in length. With enough iterations, even these single instruction differences become evident. When the program was compiled with our constant time transformations, there appears to be no timing differences, even between paths 1 and 4, despite the semantics of the language being completely preserved.



```
x = 1
if x == 1:
    if x:
        x = 2
else:
    y = x
```

```
x = 1
if x == 1:
    # then1 renamings: {x → if1_then_x}
    if if1_then_x:
        # then2 renamings: {if1_then_x → if2_then_x}
        if2_then_x = 2
else:
    # else1 renamings: {y → if1_else_y}
    if1_else_y = x
```

```
x = 1
if1_test = x == 1
# then 1
if1_then_x = x
if2_test = if1_then_x
# then 2
if2_then_x = if1_then_x
if2_then_x = 2
if (if2_test) if1_then_x = if2_then_x
# else 1
if1_else_y = y
if1_else_y = x
if (if1_test) x = if1_then_x
if (not if1_test) y = if1_else_y
```

Figure 3. An example of applying our constant time transformation on the Python program in ”Original Code”

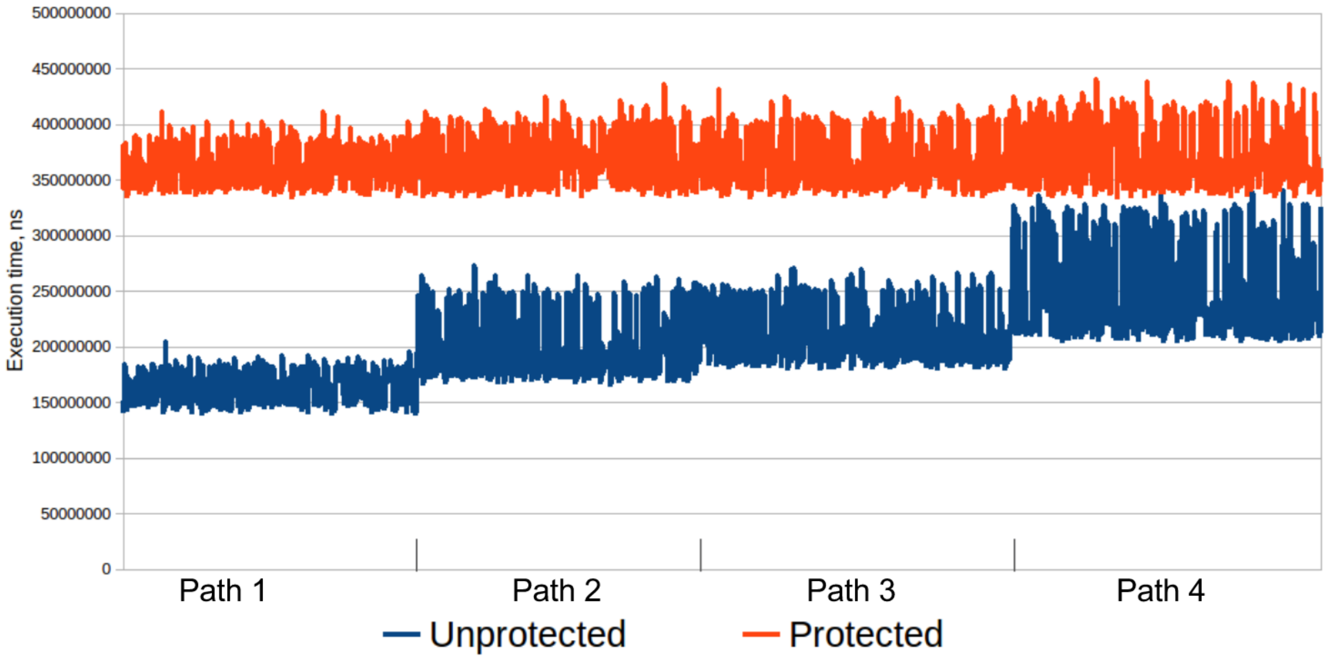


Figure 4. Results from timing multiple executions of two binaries, one produced by an insecure compiler (blue) and one by our constant time compiler (red).

Our implementation introduces a non-negligible overhead. On average in the above figure, our solution is 1.76 x slower, compared to Zigzaggers 1.2 x. If timing attacks are a practical concern, then some overhead will be necessary, as an attacker cant know if the faster or slower computation was performed. However, we still have a 1.4 x slowdown as compared to every path being taken in the unprotected binary (path 4). This is almost entirely due to the excessive safety and naivety of the transformation as defined above. Many renamings and initializations can be removed while preserving the constant time nature of the code, which we do not explore in this work. With optimizations added to the algorithm, runtime should actually be lower than running all branches in the unprotected binary, as we will not run into issues with mispredicted branches and pipeline flushes.

5 Related Work

Timing attacks endanger a wide range of secure applications, including ones that do not utilize SGX, especially cryptographic libraries [3] [6] and allow local adversary to recover private key in under 5 minutes [15]. For that reason, timing side-channels were studied for over 2 decades [9] and many defences have been proposed, such as reducing accuracy with noise [8], spawning multiple threads to execute all branches [5] or closing access to the clock [13] entirely. Another interesting solution to timing side channel was introduced in FaCT [4]—new constant-time, C++ compliant Domain Specific Language—which allows developers to write idiomatic, constant time code, but they do not protect against branch shadowing.

Cache side-channel attacks are also applicable in non-SGX case and has been a threat [12] to cryptographic libraries. There are effective defences against cache-based attacks, including a clever usage of transactional memory [7], proposed in 2017.

There is a research, dedicated specifically to SGX side-channels, including work done by Lee et al, in which a novel Branch Shadowing attack [10] was introduced, as well as defences against it in software and hardware.

Many defences are focused on a particular side-channel and may not always efficiently stack with each other. Still, there are defences that attempt to protect against a very wide range of attacks, such as Racoon [14], which introduces the notion of Digital Side-Channel—defined as a side-channel that carries information over discrete bits—and defends against them, however protections do not include Branch Shadowing.

Verification of constant time implementation is an extensively researched subject. The conventional way to guarantee ground truth on timing is a set of benchmarks, such as one used in to verify fixed-point constant-time math library [2]. Static analysis at various levels also may be effective: one approach analyzes [1] LLVM IR with formalization that reduces the constant-time security of a program P to safety of a product program Q that simulates two executions of P. This work can help ensure that an implementation is constant time.

6 Conclusion

Side channels, such as cache and timing leakage, as well as the newly proposed branch shadowing attack, propose dangerous threats to the otherwise secure SGX solution provided by intel. We were able to extend the safety guarantees that can be provided by the compiler beyond Zigzaggers branch shadowing mitigation, to include both cache and timing side-channel protection. The solution, which renames variables to safely execute both branches of each conditional branch, should be able to be made nearly as fast as the longest execution path of an unsecured program, and has been shown to produce completely constant time binaries. Security sensitive programs, such as cryptographic libraries will see much of their maintenance overhead decrease (due to the ability to write idiomatic code even for constant time implementations) if they adopt compilers that can guarantee their code will always be compiled to a side-channel free binary.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations.. In *USENIX Security Symposium*. 53–70.

[2] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 623–639.

[3] Cyril Arnaud and Pierre-Alain Fouque. 2013. Timing Attack against Protected RSA-CRT Implementation Used in PolarSSL.. In *CT-RSA*. Springer, 18–33.

[4] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *Cybersecurity Development (SecDev), 2017 IEEE*. IEEE, 69–76.

[5] Dominique Devriese and Frank Piessens. 2010. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 109–124.

[6] Cesar Pereida Garcıa and Billy Bob Brumley. 2016. *Constant-time callees with variable-time callers*. Technical Report. IACR Cryptology ePrint Archive, Report 2016/1195.

[7] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. (2017).

[8] Wei-Ming Hu. 1992. Reducing timing channels with fuzzy time. *Journal of computer security* 1, 3-4 (1992), 233–254.

[9] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.

[10] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2016. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952* (2016).

[11] Moxie Marlinspike. 2017. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/. (26 September 2017).

[12] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers Track at the RSA Conference*. Springer, 1–20.

[13] Colin Percival. 2005. Cache missing for fun and profit (2005). *URL: http://www. daemonology. net/papers/htt. pdf* (2005).

[14] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution.. In *USENIX Security Symposium*. 431–446.

[15] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. *arXiv preprint arXiv:1702.08719* (2017).