Constant-Time Code: The Pessimist Case

Thomas Pornin

NCC Group, `thomas.pornin@nccgroup.com`

6 March, 2025

Abstract. This note discusses the problem of writing cryptographic implementations in software, free of timing-based side-channels, and many ways in which that endeavour can fail in practice. It is a pessimist view: it highlights why such failures are expected to become more common, and how constant-time coding is, or will soon become, infeasible in all generality.

1 Introduction

Timing attacks are a class of attacks on cryptographic implementations in which the attacker infers information on secret values (especially private keys) from timing measurements. We consider here the case of *software* implementations, running over a large range of mostly general-purpose CPUs, from small microcontrollers to the largest multi-core server processors. Timing attacks were initially described in the case of RSA and DSS, leveraging the total computation time of a private key operation[20]. Later variants exploit various side-channels, in particular caches[24]: timing measurements can differentiate between a hit and a miss when accessing memory through a cache, which depends on the cache state, which itself is a product of the previous memory access pattern. Cache timing attacks reveal information on secret-dependent addresses used for memory accesses. Conditional branches with secret conditions are also a prime target, both for the effect on the instruction caches, and the effect on overall execution time.

Timing attacks are only one sub-case of side-channels attacks, but they are often considered specially, because they can be exercised remotely[5,6], in contrast to most other side-channels that require the attacker to be physically located in the vicinity of the target. For instance, timing attacks may be attempted between two virtual machines co-located in a data centre (possibly running on the same physical host), using only the inherent abilities of servers at measuring elapsed time (with, in particular, cycle counters present in most CPUs), while the attacker is comfortably operating from a remote place on another continent.

Constant-time code is software specially designed to be immune to timing attacks by not having such side-channels. Specifically, we define it as software which is such that no secret information can be inferred, even statistically, from timing measurements. It is a technically incorrect expression, since it does *not* mean that the code executes in a constant amount of time; but it is traditional and there is no better name in widespread usage for now[1].

[1] One may encounter the adjective *isochronous*, which sure looks fancier; but its Greek roots just mean "constant-time", which implies that this term is exactly as incorrect as the expression it purports to replace. If we want more exact terminology, we need to conjure up some awkward neologism, e.g. *achronognostic* ("there is no knowledge in time").

There are known techniques[3,17] for writing cryptographic software in ways that aim at following strict constant-time discipline; they focus on avoiding memory accesses at secret-dependent addresses, conditional branches with secret conditions, and some operations such as integer division which typically have an operand-dependent execution time. The *hope* is that such techniques are sufficient to obtain constant-time behaviour on a wide variety of hardware platforms, from small microcontrollers to large, high-end multi-core CPUs. The purpose of this note is, in a nutshell, to crush that hope. These techniques are becoming ineffective for reasons which will be developed in the next sections, but can be summarized as follows:

1. Compilers are applying optimization techniques that are heuristically good for performance on general software, but happen to leak information through timing-based side-channels when used on secret data. Since general-purpose CPUs and compilers are applied to a wide variety of tasks, compilers have no reason to stop doing such optimizations.
2. Constant-time coding techniques mostly aim at fooling the compiler, to prevent it from applying these optimizations. Since compilers keep getting smarter, theses techniques lose their efficiency.
3. Just-in-time (JIT) compilers have access to runtime data; they can, and will, use such information to perform extra optimizations that static compilers cannot do, and further destroy the developer's attempt at achieving constant-time code.
4. JIT compilation is becoming pervasive and is now employed in various contexts, in particular *inside* CPUs. Such compilers can do the same optimizations as "external" JIT compilers, but are not publicly documented, and the output of their "machine" code translation cannot be inspected to detect deviations from constant-time execution.
5. Modern hardware and software stacks use a very layered structure, with each layer striving to hide its internal functioning details from upper layers. The socio-economic context which allowed such hardware to exist inherently relies on such abstraction (under the name of "industrial secrecy"). An effective constant-time coding process would require a model of computation with strong guarantees on timing-related characteristics, that would be maintained through all these layers, down to the semiconductors that implement the logic gates. This industry-wide vertical cooperation is unlikely to happen.

2 Compiler Optimizations

It is now customary, for cryptographic libraries, to employ constant-time coding techniques so that information about secret data, in particular cryptographic keys, does not leak through timing-based side-channels. These techniques focus in particular on the handling of Boolean values: secret Boolean conditions, derived from (for instance) key bits, impact which code must execute and what data elements must be read from RAM. A classic case would be a square-and-multiply algorithm for implementing the modular exponentiation at the core of a Diffie-Hellman operation:

– We want to compute $g^x \bmod p$, for some integer g, modulus p, and secret exponent x.
– We process bits of x one by one, in high-to-low order; we maintain a variable r (initialized at 1), and, at each step, we replace r with either $r^2 \bmod p$ (if the corresponding bit of x is 0) or $gr^2 \bmod p$ (if the bit of x is 1).

In order to avoid leaking information on the exponent bits, we must *systematically* compute both r^2 and gr^2, and then select the correct value in a constant-time way, in particular with a memory access pattern that does not depend on the value of the secret bit. An implementation of that conditional selection may use a function similar to the following (here implemented in C):

```
#include <stddef.h>
#include <stdint.h>

void
condmove(uint64_t *restrict a, const uint64_t *restrict b,
         size_t len, uint32_t x)
{
    uint64_t mask1 = -(uint64_t)(((x | -x) >> 31) ^ 1);
    uint64_t mask2 = ~mask1;
    for (size_t i = 0; i < len; i ++) {
        a[i] = (mask1 & a[i]) | (mask2 & b[i]);
    }
}
```

This code is derived from the BearSSL library source code[25]. This example was communicated to me by Moritz Schneider, who recently analyzed similar situations in several cryptographic libraries[27], and found that purportedly constant-time code is outsmarted by modern compilers, and turned into non-constant-time machine code.

This `condmove()` function takes as parameters two arrays `a` and `b`, each containing `len` words, and replaces the contents of `a` with the contents of `b` if and only if the parameter `x` is non-zero. In the source code, two mask values `mask1` and `mask2` are first computed, so that one is the all-ones pattern and the other is the all-zeros pattern, depending on whether the control value `x` is zero or non-zero. The computation of `mask1` is a classic trick: if `x` is non-zero, then either `x` or `-x` (or possibly both in one case) has its highest bit set. The intent is to produce the mask values without using a plain comparison (“`x != 0`”) so that the compiler *does not notice* that we are really making a conditional move.

The compiler is not fooled. Back when BearSSL was originally written (around 2015), such tricks were working. But if we try with a recent enough compiler (Clang 18.1.3, for a 64-bit x86 target, “`-O3`” optimization level, Intel assembly syntax), we get this:

```
condmove:                                   ; @condmove
        .cfi_startproc
        test    rdx, rdx
        je      .LBB0_2
        test    ecx, ecx
        je      .LBB0_2
        shl     rdx, 3
        jmp     memcpy@PLT                  ; TAILCALL
.LBB0_2:
        ret
```

It is worth detailing what happened here. The first test is on register `rdx`, which contains the length of the arrays (`len` parameter); the compiler skips the whole process if the length is zero (this is fine, the length of the arrays is not secret). The second test is on `ecx`, which contains the `x` parameter: the compiler saw through the complicated expression, and understood that it really is an equality comparison with zero, which can be done with a simple `test` opcode, followed by a conditional branch. The compiler *also* worked out that the loop was either a data move or a no-operation, depending on the result of the test on `x`, so that one branch amounted to nothing at all, while the other could be done with the standard function `memcpy()`[2].

This is not a compiler defect. What this example shows is the compiler doing its job, and doing it well. Finding that kind of optimization is exactly what the compiler is designed to do, and what is generally expected from it. This, in fact, could be the ultimate cause of the problem which is explained in this note: general-purpose computers are, by definition, used for many purposes, most of which not being cryptographic tasks; both the hardware, and the software tooling, are optimized for heuristically fulfilling most tasks as well as possible. Preventing timing-based side-channels is in direct opposition to these optimization goals, since heuristic data-dependent shortcuts are a primary source of performance improvement.

This example shows the inherent tension in constant-time coding, as usually practiced: we want the compiler to understand the code enough to produce an efficient translation of it into assembly, but we also want the compiler to *not* understand the code enough, lest it produces a *really* efficient translation of it into assembly. The developer is fighting the compiler. As compilers get better over time, this is a losing battle. In the example above, one could make a more complicated expression for the masks, that will *for now* defeat the compiler; for instance, Clang 18.1.3 does not currently work out that the mask values are a Boolean in disguise, if we use this code:

```
    x |= x >> 16;
    x |= x >> 8;
    x |= x >> 4;
    x |= x >> 2;
    int32_t y = x & 3;
    y *= y;
    y = (y & 3) - (y >> 2);
    y *= y;
    y = (y & 3) - (y >> 2);
    uint64_t mask2 = -(uint64_t)y;
    uint64_t mask1 = ~mask2;
```

One can check that indeed the values of `mask1` and `mask2` are computed properly (this convoluted code leverages the fact that for any integer z, then $z^4 = 1 \bmod 5$ if $z \neq 0 \bmod 5$). However, this sequence is somewhat expensive, both in code size and computational cost, and it moreover relies on the compiler *not* noticing that `x & 3` has a limited range of possible

[2]Part of that analysis leverages the fact that the two arrays do not overlap each other, which the compiler knows thanks to our use of the `restrict` keyword. If we remove that keyword, then the generated code must account for potential overlap between the two arrays, but still has a memory access pattern that depends on whether `x` was zero or not.

values (0 to 3) and making an exhaustive evaluation of what mask values could be obtained for each. If the compiler did make such an exhaustive evaluation, then it could work out that three of the possible values lead to `y` being equal to 1, and the last one sets `y` to 0, and the previous optimizations would apply again. Presumably, as computers get more powerful and compilers correspondingly more aggressive in their optimizations, future versions of Clang will again destroy our attempts at constant-timeness.

The main conclusion here is that trying to achieve constant-time processing through software constructions that hide the true nature of performed operations from the compiler is a fool's errand and doomed to fail. We might try to infer that the solution is to bypass the compiler completely, and write assembly code directly; however, this also will ultimately fail, as will be detailed in the next sections.

3 CPU Structure History

In this section, we propose a simplified, synthetic description of how the structure of CPUs, as used in general-purpose computers, has evolved over the last five decades. Many details are omitted; the point of this description is to highlight how far modern hardware has departed from the abstract model of executing one instruction at a time, and show how such evolution seamlessly blends into more extensive arbitrary, just-in-time compilation inside the CPU itself.

While the description below uses several types of historical and recent CPUs as examples, most of the concepts can be found in action in the well-known line of Intel x86 CPUs. For a detailed analysis of how instructions are executed in recent x86 CPUs, one may refer to the excellent optimization guide by Agner Fog[12].

Abstract Model. The initial abstract model is that the CPU is given individual instructions in an encoded format ("machine code"). The CPU maintains a pointer to the next instruction to execute (often called "program counter" or "instruction pointer"). The CPU operation is an unending loop, each iteration consisting in the following steps, executed in due order:

1. Fetch the next instruction from memory at the address indicated by the program counter; the program counter value is also incremented.
2. Decode the instruction into its core operation specification and operands.
3. Perform the operations specified by the instruction. This may involve computations, updates to internal registers, and extra memory accesses. The program counter itself may be modified.

Once all steps have been performed to completion for an instruction, the whole cycle starts again, fetching the next instruction, and so on. Branch and call instructions work by modifying the program counter, the new value being naturally used at the start of the next cycle to fetch the next instruction. This model corresponds to how early computers worked, e.g. the DEC PDP-8, in the late 1960s[10].

Pipelining. Strictly adhering to the abstract process was a necessity in these early computers, since they had very few processing resources (i.e. logic gates) and most of them were reused for several sub-tasks; for instance, instruction and data fetching would use the same internal registers[3]. As technology improved with the invention of the microprocessor, more transistors could be stored in a chip, and this allowed specializing sub-units, which in turn unlocked the optimization technique of *pipelining*: the individual execution procedures for successive instructions may partially overlap. Instructions still execute in their abstract order, but the execution of the next instruction starts while the current one (and possibly some previous instructions) is still ongoing.

One early example is the MOS Technology 6500 family of CPUs[23], which powered 8-bit home computers in the late 1970s and early 1980s, such as the Apple II and Commodore 64. This CPU tries to maximize its use of the available memory bandwidth: the bus width is 8 bits, and the CPU can only read or write a single byte per clock cycle. For instance, the instruction expressed in assembly language as "`LDA #$2A`" (which loads the constant value `0x2A` into the 8-bit A register) is encoded over two bytes (`A9 2A`):

1. During the first clock cycle, the CPU fetches the first instruction byte (`0xA9`).
2. As the second cycle starts, the CPU fetches the second instruction byte (`0x2A`). *In parallel*, the CPU decodes the first byte. Note that the fetching of the second byte starts before the first byte decoding has been completed, so that the CPU does not know at that point whether the second byte is really part of this instruction, or the first byte of the next instruction.
3. In the third cycle, the CPU knows that the instruction is `LDA`, with an immediate one-byte operand which has just been fetched. The CPU immediately starts the fetching of the next instruction byte (which is part of the next instruction). At the same time, the CPU performs the operation, i.e. moves the value of the second instruction byte (the immediate operand) into register A.

As this example shows, the instruction following the `LDA` instruction in program order starts executing two clock cycles after the `LDA` instruction; one may thus say that the runtime cost of that `LDA` is two clock cycles. However, its complete execution procedure really spanned over three cycles. This rudimentary pipelining thus increased processing speed by 50%.

Later CPUs increased the number of individual steps over which the execution of an instruction splits. The Intel 80386, for instance, has six functional units that can operate in parallel: bus interface, code prefetch, instruction decode, execution, segmentation, and paging. The 80386 hardware manual contains figure 1, which illustrates how instructions start in quick succession but their processing greatly overlaps. Note how, by the time instruction 1 has finished executing (with a final memory store), instruction 2 has done most of its job (save for a final memory write), instruction 3 is decoded, and the bytes encoding instruction 4 have already been fetched from memory.

In modern CPUs, the complete execution process of even a simple instruction can span over more than a dozen cycles.

Branch Prediction. Pipelining allows achieving a high execution bandwidth, close to the memory bandwidth limit for instruction fetching. It is, however, a bit of an illusion: as

[3]In the PDP-5, a predecessor to the PDP-8, the program counter was not even in the CPU, but stored in the main memory; instruction fetching then required extra memory operations to fetch and store back the program counter value.

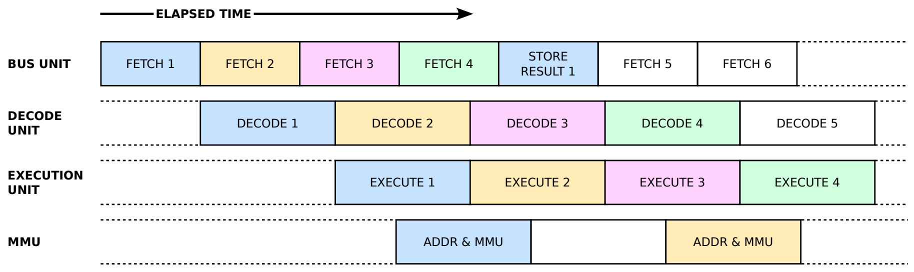

Fig. 1: Instruction pipelining in the Intel 80386 (inspired from the 80386 hardware manual, figure 2-1[14]).

the 6502 example shows, instruction execution really takes more clock cycles than the execution bandwidth suggests. The illusion breaks down in the presence of a branch instruction: when a branch is performed, the next instruction is *not* located immediately after the branch instruction in memory, but elsewhere[4]. Thus, the instruction bytes which have been fetched from immediately after the branch, and were already partially decoded, must be discarded, while none of the bytes that encode the next instruction in program order have been obtained yet, let alone decoded. This condition is known as a *pipeline flush* and can have a high cost, proportional to the pipeline depth, as the instruction targeted by the branch is fetched and decoded.

Various ways to reduce the cost of pipeline flushes have been employed. In some historical instruction sets (e.g. MIPS and SPARC), *delay slots* are used to, basically, foist the problem unto the developer: the instruction that follows the branch, and which was partially decoded and executed, is completed anyway, regardless of whether the branch is taken or not. In effect, this changes the semantics of the branch from "continue execution at that address" into "execute the next instruction, *then* continue execution at that address".

Static branch prediction is a heuristic optimization in which the CPU tries to predict whether a conditional branch will be taken based on a simple rule. Rudimentary prediction may simply expect conditional branches not to be taken, thus incurring the cost of the pipeline flush only if the branch is indeed taken (this is how the Intel 80386 operated). A more advanced form of static branch prediction heuristically assumes that a backward branch is taken (since this is the classic case of loops), while a forward branch is not; this may be further informed by explicit hints encoded in some instruction fields and provided by the developer. Such static branch prediction can reduce the cost of a pipeline flush, assuming that the prediction is correct most of the time; it also requires some specialized circuitry in the instruction decoder to provide early recognition and processing of branch instructions.

Dynamic branch prediction is used by modern large CPUs (in recent Intel x86 CPUs, only dynamic branch prediction is used; hint prefixes are ignored). A specialized cache unit

[4]Depending on the source, branch instructions may also be called *jumps*. Some microprocessor manuals use both terms with some specific semantic difference, e.g. jumps using absolute addresses while branches are relative to the current program counter value. Here, such distinctions do not matter and we use the term "branch" exclusively.

remembers, for the most recently seen branch instructions, that the instruction *is* a branch, and its kind (conditional branch, absolute branch, indirect branch to a computed address, ...) and how it previously fared. Various and increasingly complex pattern recognition methods are used to properly predict common cases, in particular tight loops with a fixed number of iterations, and the call/ret sequences.

An inherent consequence of branch prediction is *speculation*: based upon the correctness of the prediction, some partially executed instructions may have to be abandoned. Speculated execution may include some externally visible effects that remain even if the execution was cancelled in the formal model. In particular, speculated memory operations may have altered the contents of caches, which will impact the timing characteristics of subsequent accesses[21].

Register Renaming. Since pipelined operations execute in a partially overlapping way, they may be subject to interdependencies that constrain their execution. For instance, consider the two following x86 instructions (with Intel notation; the left operand receives the result of the operation):

```
    imul    eax, ebx    ; multiply eax by ebx, result in eax
    add     ecx, eax    ; add eax to ecx, result in ecx
```

The integer multiplication is relatively expensive and will need several clock cycles for the mathematical operation alone (not counting instruction fetching and decoding, and data movement within the CPU). The second instruction performs an addition that uses the result of the first instruction as one of the operands; that addition cannot mathematically start until the multiplication has completed. This is known as a *true dependency*, and at execution time, in a pipelined (in-order) CPU, it must imply a delay (*pipeline stall*) while the CPU waits for the multiplier unit to complete its work. *A contrario*, consider this sequence:

```
    imul    eax, ebx    ; multiply eax by ebx, result in eax
    mov     ebx, [edi]  ; load word at address edi, into ebx
```

These two instructions use the same register (`ebx`), which induces a *false dependency*: though the two instructions share the same resource, there is no mathematical necessity that the multiplication completes before the memory load may occur.

Register renaming is a technique which avoids pipeline stalls from false dependencies. In a nutshell, the CPU contains more internal registers than are visible to the developer at the assembly level; when an instruction is decoded, internal registers are allocated for each operand, and the in-CPU allocation table maintains the mapping from "external" register (such as `eax`) to the "internal" register that contains its value, as per a given instruction in the executed code sequence. In the example above, `ebx` in the `mov` instruction would be mapped to a different internal register than `ebx` in the `imul` instruction, breaking the false dependency and avoiding the stall.

Register renaming is also an essential tool of speculative execution, as is needed for any branch prediction with a deep pipeline. Suppose that the `imul` instruction is executed speculatively, and is later on found to be spurious, i.e. the actual execution path (in the abstract programming model) did not include that operation, due to a previous mispredicted conditional branch. The register renaming unit, seeing the `imul` instruction with two operands (`eax` and `ebx`), really used *three* internal registers: one for each of the input operands, and a distinct third register to receive the result of the multiplication. For instance, suppose that `eax` and `ebx` are mapped to internal registers `r1` and `r2`, respectively, when `imul` is encountered; the output, which formally replaces the contents of `eax`, is redirected to a newly allocated register `r3`, so that it does *not* overwrite the contents of `r1`. If the instruction is later found to have indeed been part of the instructions to execute in the abstract model[5], then `r1` is discarded (i.e. marked free for reallocation) and `eax` is now mapped to `r3`. On the other hand, if the instruction must be abandoned, then `r3` is discarded, and the register allocation will keep pointing to `r1` for `eax`.

Micro-operations. Instruction encoding is subject to various constraints. The initial design of individual instructions and their encoding is a careful trade-off between expressivity of operations, performance of decoding by the CPU, ease of use by developers and compilers, and encoding size. Code compacity is highly desirable (since memory bandwidth is often a performance bottleneck) but may complicate the instruction decoding, or prevent good code generation by compilers. Moreover, this trade-off is relative to the technology level at the time the instruction set is designed, but a given encoding may have to be supported for a long time for backward compatibility reasons, even though the available technology has changed. For a famous example, modern x86 CPUs from 2025 still use an instruction encoding that harks back to the Intel 8086 CPU, launched in 1978. Even though registers have grown (from 16 to 32 to 64 bits) and various new instructions have been added, the essential features of the instruction encoding in these modern, large CPUs used in big servers and laptops, have been in place for close to five decades.

Backward compatibility is an essential consequence of the economic forces that allow the multi-billion dollars development process for new, faster CPU models. It must thus be maintained, even though it has a cost in terms of a larger, more complex instruction decoding unit, with a higher latency (hence a longer pipeline and cost of pipeline flushes). Modern CPUs cope with it by breaking instructions into *micro-operations* (µops). For instance, an instruction such as `add eax,[ebx]`, which performs a memory read followed by an addition that uses the read value as input operand, will be split into two µops that will perform, respectively, the memory read and the addition. These µops do not exist outside of the CPU; they are invisible to the developer. However, the CPU may keep them in an internal encoded format; for instance, on Intel x86 CPUs since the Pentium IV, the innermost cache for instructions does not contain a copy of the most recently accessed instruction *bytes*, but a representation of the µops resulting from the instruction decoding; the instruction decoder, and beyond it the level 1 code cache and outer cache layers, are used only when the requested µops were not found in that innermost cache.

The transform of instructions into µops can also go into the other direction, i.e. merging several developer-level instructions into a single µop, e.g. to combine a comparison and a conditional branch into a single compare-and-branch operation. Some other instructions may be elided, resulting in no µop at all, because their effect can be ignored. For instance, recent x86 CPUs will elide register copies (`mov eax,ebx`) since they can be done implicitly by

[5] Instructions that have been executed speculatively and are confirmed are said to *retire*; the *retirement buffer* stores such instructions whose final side effects must be propagated.

the register renaming unit; similarly, a `xor eax,eax`, which sets register `eax` (and the flags) to zero, is handled by a mapping to a special always-zero internal register, and yields no µop, making it "free"[6].

Out-of-Order Execution. Some instructions may have a variable latency that depends on a large number of external parameters; in particular, every memory read may have to go through multiple levels of cache, and its latency can take about all values from an almost immediate response (e.g. 4 clock cycles on recent x86 CPUs, for accesses to data which is in the L1 cache) to thousands of clock cycles[7]. Moreover, the latencies of individual instructions can vary depending on the exact CPU model on which the code runs. For these reasons, developers (and compilers) cannot in general find, for a given task, a unique sequence of instructions that will be optimal or close to optimal in all situations where the code will run. To help achieve better performance, modern CPUs can dynamically reorder instructions, using a method first described by Tomasulo[29].

In a CPU that uses Tomasulo's algorithm, the µops which are expected to execute are stored in a generalized pipeline called the *re-order buffer*, whose capacity can range up to hundreds of µops. Each instruction in the ROB has some dependencies (mainly on register contents) and becomes eligible for execution when these dependencies are fulfilled. At each clock cycle, the CPU will select some eligible µops for execution. When an instruction has been executed and is confirmed (i.e. that instruction was indeed part of the sequence to execute in the abstract programming model), then the instruction is moved out of the ROB, into the *retirement buffer*, which handles definitive application of the instruction side effects. Speculation is general and heavily relies on branch prediction, as well as assumptions such as memory accesses succeeding and not triggering a memory protection exception. A large number of internal registers is required to support rollbacks efficiently.

Figure 2 illustrates the process of program execution in a CPU with support for out-of-order execution. The six steps through which a (micro) instruction goes through are shown:

1. Using branch prediction, the next predicted to execute instructions are fetched, decoded, and translated into µops with renamed registers.
2. Depending on what values each µop needs, these instructions become eligible for execution.
3. At any point, the CPU selects some eligible instructions (usually but not necessarily following the program formal order) and executes them.
4. Executed instructions are still speculative, and wait for confirmation.
5. When the confirmation boundary advances past an executed instruction, that instruction retires, and goes into the retirement buffer. Its side effects are applied.
6. When all side effects of a retired instruction have been applied, the instruction is removed from the retirement buffer.

In figure 2, the confirmation boundary cannot advance until the instruction that immediately follows it is executed; at this point, that instruction is predicted to execute, but cannot

[6] An elided instruction still occupies some resources in memory and needs to be decoded at some point; its cost is thus minimal, but not exactly zero.

[7] Accesses to memory pages not present in the TLB may imply a cascade of extra memory accesses to load the page characteristics, each of them being amenable to more cache misses, even if no CPU exception ultimately occurs.

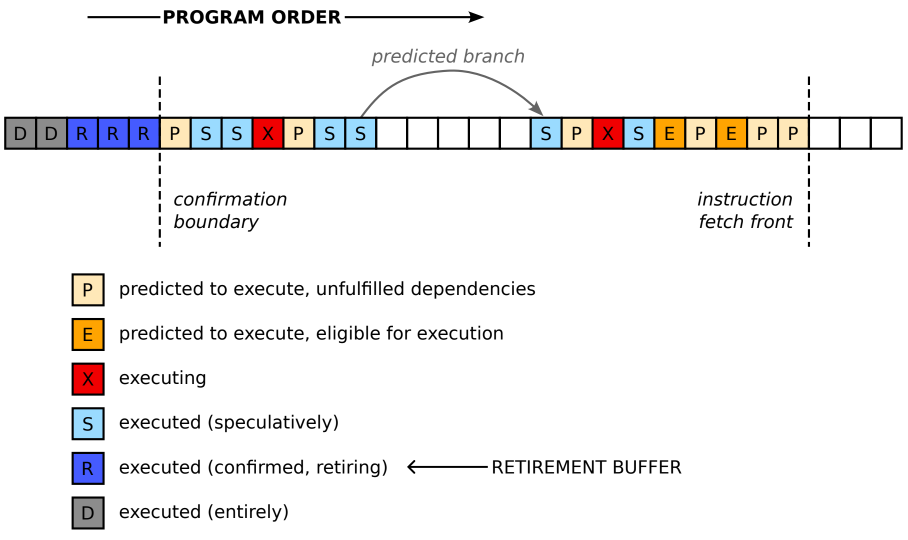
PROGRAM ORDER
predicted branch
D D R R R P S S X P S S S P X S E P E P P
confirmation boundary
instruction fetch front
P predicted to execute, unfulfilled dependencies
E predicted to execute, eligible for execution
X executing
S executed (speculatively)
R executed (confirmed, retiring)
RETIREMENT BUFFER
D executed (entirely)

Fig. 2: Re-order buffer for out-of-order execution.

do so yet because it is waiting on one of its dependencies (e.g. a data value that is fetched from memory, as part of the side effects of one of the instructions in the retirement buffer). Instead of being merely stuck, the CPU can execute further instructions for which the dependencies are already fulfilled. The *instruction fetch front* marks the latest instructions that have been fetched by the CPU; this front can advance as long as the re-order buffer is not full.

Branch prediction and out-of-order execution explain why compilers tend to use conditional branches: with out-of-order execution, branches are mostly free, as long as they are correctly predicted. In old, in-order CPUs, especially in the early RISC-inspired designs in the 1980s and early 1990s, branches were considered very expensive, since branch prediction was not very effective at that time (for lack of silicon resources to implement good pattern detection and caching), and the cost of mispredicted branches was becoming quite large as pipelines where lengthened. CPU designers were preferring *predication*, i.e. making execution of most instructions conditional. An extreme example was the original ARM, in which almost all instructions included a 4-bit condition field that indicated what combinations of the CPU flags were required for execution of an instruction. For instance, consider the following C expression, which adds `y` to `x` only if `c` is non-zero (all values have type `unsigned int`):

```
x += c ? y : 0;
```

Compiling this code with GCC (11.3.0) for an ARMv4 target leads to the following code (with `x`, `y` and `c` being in registers `r0`, `r1` and `r2`, respectively):

```
cmp     r2, #0        @ compare r2 with zero
addne   r0, r0, r1    @ add r1 to r0 only if r2 was not zero ('ne')
```

We see here predication being used on the `add` instruction, which is skipped unless the "`ne`" (not-equal) condition is true (i.e. the `Z` flag is cleared). However, in later versions of the instruction set architecture, most predication fields were removed; on ARMv7-M, GCC produces the following:

```
    cbz     r2, .L2       @ branch to .L2 if r2 is zero
    add     r0, r0, r1
.L2:
```

GCC uses a conditional branch, which avoids the dependency on `r2`: in an out-of-order CPU pipeline, and under the assumption that branch prediction is effective, the addition will become eligible for execution *before* the value of `r2` becomes available. We may note that the ARMv7-M architecture still includes a (somewhat limited) predication support, and the conditional addition could still have been expressed in a branchless way:

```
    cmp     r2, #0
    it ne                 @ predication prefix: one conditional instruction
    addne   r0, r1        @ conditional addition of r1 to r0
```

GCC elected *not* to do that, because the conditional branch version is faster in newer CPUs. This of course relies on the heuristic assumption that most branches are correctly predicted, which is the case in practice; some cryptographic code is an exception here, with its usage of (purposely!) unpredictable, uniformly selected random values. We see again that general-purpose CPUs are not optimized solely for cryptography, but for a much larger spectrum of tasks, in most of which branch prediction works well.

Other Optimizations. Many other optimization techniques are being developed, some already implemented. The microprocessor industry is highly competitive; no great gains are expected from further increases in transistor density, since current gates are already close to the minimum size beyond which electrons leaking between neighbouring gates through tunnel effect imply too large a parasite current. There is therefore high market pressure to find and implement new ways to make code run faster. We list here a few such tricks; this is not an exhaustive list.

Branch prediction focuses on branches; when the branch is conditional, this is equivalent to making a prediction on the Boolean value of the condition that the branch instruction uses as input. *Value prediction* (not to be confused with predication) was proposed in the 1990s to extend this notion to arbitrary values, especially when loaded from memory[22]: if the CPU detects that memory loads from a given location tend to return the same value repeatedly, then subsequent loads can be predicted to keep returning that value, allowing instructions that depend on that value to execute (speculatively) earlier than the completion of the load operation. Such value prediction has not seen much use yet in CPU designs, but at least the recent Apple models (M3, M4, A17 Pro) implement it, which allows some microarchitecture-based attacks[19]. In a more general way, value prediction may allow all sorts of timing attacks, since it provides a detectable behavioural change (longer execution time due to a pipeline rollback on misprediction) when an internal value is distinct from what the CPU heuristically expected.

Data-dependent prefetching consists in using the data that the CPU sees (e.g. in registers or in cache memory) to optimistically prefetch other data elements, so as to lower the latency cost of further memory accesses. Prefetching in the context of processing data elements in increasing address order is common enough, but recent CPUs go further by following values that "look like" pointers; this speeds up many common constructions such as linked list walking, structure field access, and virtual method dispatch in object-oriented languages. Of course, anything data-dependent is a prime target for side-channel attacks, and this prefetching mechanism has been demonstrated to be exploitable in the case of the Apple M2 and M3 CPUs[7]. Recent Intel x86 CPUs also implement data-dependent prefetching[15] (though no working exploit was published so far).

Silent stores (also called *store elimination*) avoid the cost of performing a memory store when the new data bytes are identical to the byte values that they replace. In a normal write operation, the new value is written in the corresponding cache line in L1 cache, which must be ultimately flushed to the outer cache levels and the main memory; in multi-core systems, other cores need to be informed of the modification so that their own caches are updated. If the new value is not different from the previous one, then this background memory flush can be avoided, which preserves the memory bandwidth for other operations. Some Intel CPUs exhibit use of silent stores in the case of writing zeros over zeros[9].

Summary. In modern CPUs, as found in today's smartphones, laptops and servers, what the CPU actually executes is quite removed from what the developer may see at the assembly level. The instruction sequence is translated into a different, internal instruction set (the µops), operating on a different (and larger) set of internal registers, and stored in an internal cache structure. The execution process may run the instructions in a variable order, informed by dynamically harvested information about the handled data and the past behaviour of the same piece of code. These are most of the essential features of just-in-time compilation; it is thus just a matter of quantitative expansion of resources, which will happen over time (and in many respects has already done), before CPUs process the developer-visible machine code through a full virtual engine that leverages all JIT compilation techniques.

4 JIT Compilation

Just-in-Time compilation is the general name for compilation techniques which are applied during the program execution rather than in a prior step. The separation of pre-execution and execution is somewhat arbitrary and many variants of JIT have been employed in various situations, since the concept first emerged in the 1960s[4]. Here, we mostly envision "full" JIT systems featuring characteristics that were pioneered by Smalltalk implementations in the 1980s[8], and later refined in other systems:

– Translation is performed on-demand, for parts of the program (e.g. a specific function) at a time, only when invoked.
– Possibly, the JIT engine may support several execution mechanisms, e.g. a simple interpreter, and a more complex translation to machine code only for parts of the code which are invoked repeatedly. More than two mechanisms may be present, with increasingly aggressive optimization techniques.

- When translation occurs, it may use heuristic data gathered from previous invocations of the translated part, in particular detection of constant or mostly-constant input values.
- Translation granularity does not necessarily correspond to the formal subdivision of the program into functions. A *tracing JIT compiler* tries to detect repeated execution paths, e.g. often-invoked loops, and optimizes only these paths, not necessarily full functions.
- Translation output is cached; an eviction policy is used to remove older translated code which seems unlikely to be used again during execution (this is used in particular to cope with CPU-intensive initialization code which is no longer used after having been performed once).

JIT Compiler Types. For the purposes of the present discussion, we can consider three types of JIT compilers.

Language-specific JIT compilers are tied to a specific programming language. A prime example is the implementation of JavaScript within Web browsers; the JavaScript code is obtained dynamically, not only through downloading, but also as computed data (through a language feature such as `eval()`). Several implementations of JavaScript exist; the Mozilla Firefox browser uses an engine called SpiderMonkey[28], which supports three translation levels. Other languages that often integrate JIT compilation include Python, Ruby, Smalltalk...

In general, language-specific JIT compilers can leverage the specificities of the language, in particular:

- Source-level type information, if present, may be used by the JIT compiler to better optimize its output.
- The JIT compiler may be tailored to better support constructs which are idiomatic to the language, for instance a specific type of dynamic method dispatch in object-oriented systems.

Such JIT compilers may use all the optimization techniques enjoyed by static compilers, as well as dynamically gathered information; the latter is especially true for JavaScript, since the language's only formal number type is a floating-point format (IEEE 754 binary64 type), and an important part of the JIT compiler is to identify which values are actually integers that may use the CPU's general-purpose registers and operations.

Virtual machines present themselves as non-physical CPUs with their own instruction set architecture. The source code produced by the developer is compiled into instructions for that virtual CPU; an emulator for that CPU is used to actually run the code. The emulator will typically use JIT compilation techniques to enhance the performance. A well-known example is the Java Virtual Machine (JVM), originally developed in the mid 1990s by Sun Microsystems to support the specific execution model of the Java language; the JVM has been used later on as target for other non-Java programming languages, such as Kotlin, Scala or Clojure. The use of a virtual machine by Java was meant to ensure a high level of portability ("write once, run anywhere", as the original slogan went) while enforcing a strict typing and memory-safe execution model that could be used to run potentially hostile code in a sandboxed way, typically as an "applet" integrated in a Web-based service.

In modern Web browsers, the JVM is rarely encountered, but a conceptually equivalent mechanism is present under the name WebAssembly (Wasm)[30]. The Wasm virtual machine offers a general-purpose 32-bit architecture, and can be used as a target for various languages, including classic statically-typed compiled languages such as Rust or C. Wasm is not necessarily tied to Web browsers, but its features, in particular portability and sandboxing, are most useful in a Web integration context.

Another example of a JIT-compiled virtual machine is the translation layer implemented by Apple under the name Rosetta, to help with the transition of Apple's Mac computers from a PowerPC to an Intel x86 architecture, in the 2006-2011 period. In 2020, Rosetta 2 was introduced to handle another architecture transition, from x86 (64-bit) to ARM64. In both cases, the goal was to be able to run existing software on the new machines, without needing any support by the software vendors[8]. In both cases, the transition was successful[9]. That kind of translation technology highlights that even when writing code for a very concrete, physical architecture such as x86 CPUs, a JIT-powered virtual machine may still be involved.

JIT-compiled virtual machines can use most of the optimizations that are available to language-specific JIT compilers, though they are usually slightly limited with regard to static type information; type annotations visible in the developer-level source code are not necessarily encoded into the instructions for the virtual machine[10].

In-silicon JIT compilation is the same concept as the virtual machine, except moved beyond the "hardware boundary". Classically, a developer writes code that gets translated in some way to machine code, i.e. instructions for the CPU. How the CPU executes these instructions is in general poorly documented, mostly for intellectual property reasons. The machine code, as defined in the *instruction set architecture*, marks the boundary of what the developer can inspect; anything beyond is the hardware vendor's realm. Nothing prevents a CPU from containing a JIT compiler to help it run the instructions in an efficient way. This concept was applied to the Transmeta processors in the early 2000s[13]: the Crusoe and later Efficeon CPUs outwardly presented themselves as x86 CPUs, able to correctly run x86 code (including system code such as the operating system kernel), but internally translating the x86 code into instructions for a RISC-like architecture with explicit support for parallelism[11].

Transmeta was not commercially successful. However, the concept remained available. The more recent "Project Denver" by Nvidia developed microarchitectures (nicknamed Den-

[8]Apple computers of the Macintosh/Mac line have gone through six architectures along the years: Motorola 68k, PowerPC (32-bit), PowerPC (64-bit), x86 (32-bit), x86 (64-bit), and now ARM64. The 32- to 64-bit transitions for PowerPC and x86 CPUs were handled by the hardware itself, since the 64-bit CPUs also supported 32-bit mode. For the m68k to PowerPC transition, software vendor support was needed, Apple providing only some OS support for "fat binaries" that included compiled code for both architectures. In later transitions, the Mac software ecosystem was too diverse to be able to efficiently pressure software vendors into recompiling and redistributing versions for the new architecture, hence the need for a dynamic binary translation mechanism with good performance.

[9]JIT-powered virtual machines have been used for cross-architecture transitions in other systems, albeit with less commercial success; examples include the ill-fated Itanium ("IA-32 EL") and Alpha ("FX!32"), both for running unmodified 32-bit x86 code.

[10]Many dynamically typed languages such as JavaScript do not have source-level type annotations anyway.

[11]The internal hardware was of the VLIW kind, for "Very Long Instruction Word" (128-bit and 256-bit instructions, for Crusoe and Efficeon, respectively). Explicit parallelism is easy to support by the hardware since all complicated decisions about what parallelism may happen are taken by the JIT compiler, not the hardware. VLIW implies very large binaries, but this is tolerable in a JIT system since only the most used routines are translated and cached.

ver 1, Denver 2, and Carmel) that outwardly implement the ARM64 (64-bit ARMv8-A) architecture, but internally translate the instructions to undocumented µops, and additionally use JIT compilation techniques to automatically translate and optimize the most frequently used ARM64 routines into the internal representation. CPUs using these microarchitectures include the Nvidia Tegra K1, Tegra X2 and Tegra Xavier. Compared to a CPU using a more traditional out-of-order pipeline, the use of JIT techniques has some advantages:

- Since translation and optimization are applied only on the most used code paths, more aggressive optimization techniques can be used.
- Information gathered from previous executions of a given code path can be leveraged to inform the optimizer.
- To some extent, some optimization decisions can be moved from the hardware into the JIT software layer (e.g. register renaming), thus reducing the number of required gates and the overall CPU power usage.
- The non-optimized baseline interpreter can also be used as a low-power CPU for background tasks, especially for mobile devices such as smartphones, for whom battery efficiency is of paramount importance, and some background processing must still happen, at a possibly low pace, when the device is otherwise idle.

Challenges to Constant-Time Code. JIT compilers make it especially challenging to write constant-time code for the following reasons:

- JIT compilers can do all that static compilers can, which is already enough to cause trouble (see section 2), but their gathering of runtime information on data further improves their ability to annihilate attempts at constant-time processing. For instance, if an encryption routine uses the same key repeatedly, then the JIT engine may heuristically assume that this particular key value is a constant, and use it to further optimize computations that use it[12].
- JIT compilation is based on heuristics, similar to value prediction, but with a much higher cost of misprediction (since a wrong hypothesis on the value, type or range of an input implies a recompilation or at least a switch back to interpretation). This can plausibly turn into a timing-based side-channel that is much easier to detect and leverage by attackers.
- JIT compilation, occurring at runtime, is out of reach of developers. In a traditional economic model, the developer produces some source code which is either provided to the customers/users (open-source model), or compiled by the developer, and the resulting machine code is distributed. A constant-time-conscious developer may try to inspect the output of static compilation to see whether the machine code, at least, seems constant-time; this is not possible in the context of JIT compilation, since that happens *after* the delivery of the software to the user.

In-silicon JIT compilers make the problem especially insolvable since the actual sequence of executed µops is hidden away. There are strong market forces that lead to this situation; namely, the development of a new large CPU design is a capital-intensive process, that can happen only because the resulting hardware can be sold in large quantities and generate profits that will cover the development costs. A cornerstone of this model is that the design is not

[12]Thanks to Frank Denis for scaring me with this particular scenario.

easily copyable by third parties (in a revealing way, such circuitry designs are called “IP”, as in “intellectual property”). There is thus a strong push for *not* documenting how the CPU internally works.

This is less true in the context of small CPU designs, e.g. for microcontrollers; such designs can be small enough to have low development costs (the commercially important secrecy applies more at the integration level, where the CPU core is adjoined with some RAM and Flash, and I/O abilities). Hobbyists regularly write and publish open-source CPU designs for either custom instruction set architectures, or for standard ones that do not require a license (typically RISC-V).

5 Partial Solutions

The point of this note is to show that writing constant-time code, in a general and portable way, is not really feasible with modern hardware. However, in some specific situations, there are known methods, or at least tricks, that can lower the risk of timing-based side-channels. We list a few here. We do not aim at making an exhaustive survey; rather, we want to highlight how most of these attempts at coping with the problem revolve around avoiding, countering, and validating the developer-visible parts of the system, i.e. the language compiler and its assembly output. For lower parts, in particular the hardware (or emulation thereof, for virtual machines), some specific behaviour must be assumed as part of a given model, usually without any clear guarantee that any physical CPU follows that model.

Intel DOIT, ARM DIT. Recent Intel x86 CPUs *do* offer some guarantees of constant-time execution, for a subset of the available instructions, subject to some conditions[16]. The subset includes arithmetic and logic operations (including integer multiplications, but excluding integer divisions); floating-point operations are *not* included[13]. The “conditional move” instructions such as `cmovz` are part of the list of constant-time instructions.

As per Intel documentation, the constant-time guarantees hold for CPU microarchitectures earlier than Ice Lake (in the “Intel Core” line) and Gracemont (in the “Atom” line). On Ice Lake, Gracemont, and later cores, the guarantees are *not* provided by default, but must be enabled explicitly by setting the data operand independent timing (DOIT) flag, which is bit zero in the model specific register (MSR) called `IA32_UARCH_MISC_CTL`. Reading or setting a MSR requires a high privilege level (i.e. kernel mode, not user mode). In practice, even without this flag, the listed “constant-time” instructions are still constant-time by themselves, even in newer CPUs (as of early 2025); however, setting the DOIT flag will also disable some other in-CPU optimizations that can exhibit data-dependent timing differences, e.g. data-dependent prefetching.

In the ARM ecosystem, the ARMv8.4-A architecture defines the `PSTATE.DIT` register ([2], section C.5.2.4), which is very similar to the Intel `IA32_UARCH_MISC_CTL` register. The Apple M-series processors support DIT, but most other ARM CPUs (as of early 2025) do not. On the bright side, the DIT flag can be read and set by user code without any special privileges.

[13]Experimentally, floating-point arithmetic operations are mostly constant-time for non-exceptional cases, but extra delays are induced when infinites, NaNs or denormalized values are involved[1].

In situations where one can ensure that the used CPU is an Intel x86 and the DOIT flag is set, or an ARM-compatible CPU with enabled DIT, then it is again conceivable to try to achieve constant-time operations, at least if compiler-induced issues can be avoided.

Inline Assembly. Using inline assembly is a simple trick that can be used to prevent some unwanted data-dependent optimizations in some specific compilers, e.g. GCC and Clang for the C language. Going back to our original "`condmove()`" example in section 2, this trick may look like this:

```
#include <stddef.h>
#include <stdint.h>

void
condmove(uint64_t *restrict a, const uint64_t *restrict b,
         size_t len, uint32_t x)
{
    x = (x | -x) >> 16;
    __asm__ ("" : "+r" (x) : : );   // inline assembly to fool the compiler
    uint64_t mask1 = -(uint64_t)((x >> 15) ^ 1);
    uint64_t mask2 = ~mask1;
    for (size_t i = 0; i < len; i ++) {
        a[i] = (mask1 & a[i]) | (mask2 & b[i]);
    }
}
```

Inline assembly is a language extension which allows the inclusion of some explicit assembly code in a program. In the syntax used by GCC (and later implemented by Clang), the assembly code is provided as a literal string used as parameter to the `__asm__` construction. For historical reasons, GCC and Clang do *not* understand the contents of that string[14]; they dump it mostly unmodified in the assembly code they generate, with only textual replacement for plugging input and output operands.

In this case, one operand is specified (`x`), that the compiler is instructed to map to a register, and will be used for both input and output (`"+r"`). The compiler replaces the sequence "`%0`", where it appears in the assembly code, with the name of the register that it has chosen for that input/output operand; here, the assembly code is empty, so there is no actual replacement. The net effect is that the inline assembly yields no instruction at all, but the compiler cannot assume anything on the value of `x` after that assembly chunk. Prior to that assembly code, the compiler knows that only the low 16 bits of `x` can be non-zero (since this is the output of a right shift by 16 bits), but afterwards `x` could have any value (from the point of view of the compiler), hence "`x >> 15`" is not necessarily a disguised Boolean value (as far as the compiler knows). In effect, this prevents the previously observed optimizations. Note that

[14]GCC started as a quest to replace the closed-source, OS vendor-provided "Unix compiler"; for easier integration, it had to generate code in a textual assembly representation, as expected by the vendor-provided assembler, which limited the compiler's options. Understanding the contents of inline assembly chunks would have required the compiler to support the vendor syntax for assembly, which could have led to legal difficulties.

since the assembly chunk is empty, it is also portable: no architecture-specific part is involved. For 64-bit x86, Clang's output for the computation of `mask1` and `mask2` is the following (Intel syntax):

```
        mov     eax, ecx
        neg     eax
        or      eax, ecx
        shr     eax, 16
        #APP
        #NO_APP
        shr     eax, 15
        xor     eax, 1
        mov     rcx, rax
        neg     rcx
        dec     rax
```

The inline assembly code (the empty string) goes between `APP` and `NO_APP`. `mask1` is computed in `rcx`, and `mask2` in `rax`. We see here a nifty optimization from the compiler: `mask2` is nominally a bitwise complement of `mask1`, but is instead computed by subtracting 1 (with the `dec rax` instruction) from the value before the negation that yielded `mask1` (this works because in these machines using two's complement, negation is the same thing as a bitwise complement followed by adding 1).

Use of inline assembly to prevent compiler optimizations is a fragile technique; not only does it work only for the developer-level compiler (it cannot cover the case of an in-silicon JIT compiler or a virtual machine), but it also relies on the compiler not trying to understand inline assembly. This is not true of all compilers; for instance, the SUNWspro compiler (on Sun SPARC systems) supported inline assembly as "inline templates" and integrated them in its code generation pipeline, thus re-optimizing the assembly code in all the ways that the C compiler could do with C code. Conversely, Microsoft's Visual C compiler does not support inline assembly at all for x86 in 64-bit mode.

More Compiler Fighting. Classic constant-time coding techniques can be described as trying to fight or fool compilers, to undo their optimization efforts. Maybe the solution is to fight harder? Many methods and tools to do so have been proposed so far; a list of such tools is available on:

https://crocs-muni.github.io/ct-tools/

with no fewer than 55 listed tools (at of early 2025). Most tools are about verifying whether a given piece of code is constant-time, but some modify code generation to (try to) enforce constant-time behaviour.

All such tools rely on a set of assumptions about what instructions and conditions are constant-time in the hardware. For instance, they may assume that the target is a recent Intel x86 or ARM with DOIT/DIT enabled; in that case, the enemy is the developer-visible compiler, and once "constant-time" assembly has been generated, the job is done. Some other works try to handle optimizing hardware, based on a model of what shortcuts and tricks the CPU will use, and what conditions are sufficient to prevent the unwanted optimization to happen (e.g. the recent [11] can work around a CPU optimizing away multiplications by 0 or 1, but does not try to avoid other kinds of multiplication shortcuts, which nonetheless exist in some CPUs[15]). The main conceptual problem of such assumptions is that the exact behaviour of the hardware is rarely, if ever, documented; surveys rely on scanning existing research papers and patents to get an idea of what *could* be implemented in newer CPUs[26], but it is mostly impossible to ascertain what is implemented in a specific processor, let alone what will be implemented in newer versions in the future. Instruction set architectures can only specify, at best, some expected timing characteristics, and do so only grudgingly (even Intel DOIT and ARM DIT list the constant-time behaviour as optional, not enabled by default, and likely to be slower than the default setting). In the presence of a virtual machine or an in-silicon JIT compiler, all these constant-time guarantees disappear.

Another common characteristic of constant-time analysis tools is that they must proceed from a notion of secret data: it must be specified, somewhere, that a given data element (e.g. a cryptographic key) is secret, and operations involving it must not leak information through timing-based side-channels. Most data in a program is not secret; e.g. most loops have a non-secret number of iterations, and the tool should not highlight the loop counter as being potentially leaked through the conditional branch that terminates each iteration. Moreover, secret data can become non-secret; for instance, when symmetric encryption is applied, this is because the plaintext and the key are secret, but the ciphertext is not (this is the whole point of encryption, really). A tool that follows secret data through an instruction graph must be instructed that, at some specific points, secrecy has been "cured" and non-secret data is obtained. Tagging secret and non-secret data is normally done with developer-provided annotations, that extend the language; this often requires modifying the compiler itself, and thus adds an additional hurdle: if the tool is a patch on the compiler, then it must be either adopted by the compiler maintainers, or maintained separately as new compiler versions are produced. Separate maintenance is a significant long-term time investment that academic authors typically do not have the resources to provide, especially since compiler development is fast-paced; as for upstream adoption, compiler authors have so far shown little appetence for it.

6 Conclusion

In this note, we have tried to explain and illustrate the problem of writing constant-time code. The pessimistic conclusion is that ensuring constant-time behaviour through coding practices does not in general work, because compilers keep getting smarter, and new compilers keep popping up in many places and especially deep inside the hardware, where we cannot even see them and control what they do. Moreover, a pervasive lack of documentation throughout the whole computing stack makes it hard to even know the current situation with some degree of precision; for the in-silicon parts, this obscurity is part of the economic model that allows modern, complex and expensive CPUs to exist at all. Thus, the situation is not good and unlikely to do anything else than worsening in the future.

Constant-time coding can still be achieved in some situations where the target hardware is narrowly defined. For instance, when writing code specifically for a microcontroller running

[15]PowerPC 7xx and 74xx, aka the G3 and G4 lines, computed multiplications faster when one operand was short, with thresholds on 8, 16 and 24-bit sizes; more recently, the ARM Cortex-A53 and A55 support 64-bit multiplications but return earlier when one of the operands fits on 32 bits.

with an ARM Cortex-M4 core, and using explicit assembly routines (to avoid any compiler-induced issue), then it is possible to achieve constant-time processing, mostly because that CPU core has a relatively simple execution model (only some limited pipelining) and does not have much data-dependent timing behaviour beyond the obvious effect of conditional branches. Memory accesses should still be performed only at non-secret addresses, because interactions with DMA-able peripherals may reveal information on such addresses. Another example is when using extensive constant-time analysis in conjunction with a CPU that offers explicit guarantees at the hardware level (e.g. Intel DOIT). In the introduction of our analysis, we stated that vertical cooperation for guaranteeing constant-time behaviour was unlikely; Intel DOIT and ARM DIT are the exception, and such endeavours should be encouraged.

Cryptographic libraries should still aim for constant-time coding: while it is not feasible to obtain constant-time behaviour generically, i.e. with plain source code that will be constant-time on most platforms with most compilers, following constant-time coding practices still has benefits:

– Constant-time coding highlights the secret data elements and the operations which are most likely to induce information leakage through timing-based side-channels. This will help with application of constant-time analysis tools in situations where the target software and hardware stack is known.
– In many usage contexts, attackers are limited in their measurements to some granularity, and might be unable to detect a timing difference of a dozen cycles or so. By adhering to constant-time practices, in particular avoidance of branching based on secret conditions, a library might avoid some of the most egregious information leaks, and the remaining side-channels could prove unexploitable by attackers.

Generic cryptographic libraries should thus aim for at least "best effort" constant-time coding, even though side-channel leaks cannot be considered eradicated independently of the actual hardware target.

Cryptographic protocols can improve the situation by, for instance, focusing on short-term secrets. An example is key exchange within a TLS-like protocol: if a server uses only ephemeral use-once key pairs for the key encapsulation, and a key pair is never reused for another handshake, then side-channel leaks about that key are less of a concern. Most side-channel attacks yield only a small amount of information from each measurement, and must be repeated and statistically analyzed in order to lead to a successful breach; ephemeral keys can prevent many attacks in practice. Of course, not all cryptographic keys can be made ephemeral (in a TLS-like protocol, a server will typically use a non-ephemeral signature key pair in addition to the ephemeral key encapsulation key pair), but reducing the range of potential targets can only help.

The problem is not only cryptographic: the question of constant-time coding arises from the processing of secret data. We started this note by asserting that timing attacks are a class of attacks on cryptographic implementations, but this is not entirely true. Cryptographers tend to focus on cryptographic keys, which, by design, concentrate the secrecy in a cryptographic scheme (as per Kerckhoffs' principles[18]). However, information leaks logically apply to any processing done on any secret data. If a system encrypts some plaintext, it is because that plaintext is confidential, and thus everything that is done with that plaintext should aim for constant-time processing. The problem is merely more acute for cryptographic algorithms, which often happen closer to the input of external data, and thus more amenable to repeated experiments and measurements by attackers. We should nonetheless remember that the problem is larger than cryptography. A really comprehensive framework for confidential constant-time computing is still lacking; what this note underscores is that such a framework must encompass the entire software and hardware stack and punch through layers of jealously preserved obscurity by the involved actors.

Acknowledgments

We thank Paul Bottinelli and Gérald Doussot, who reviewed this paper.

References

1. M. Andrysco, A. Nötzli, F. Brown, R. Jhala and D. Stefan, *Towards Verified, Constant-time Floating Point Operations*, ACM SIGSAC Conference on Computer and Communications Security - CCS '18, pp. 1369-1382, 2018.
2. Arm Limited, *Arm Architecture Reference Manual*, ARM DDI 0487, 2024.
3. J.-P. Aumasson (ed.), *Cryptocoding*, `https://github.com/veorq/cryptocoding`
4. J. Aycock, *A Brief History of Just-In-Time*, ACM Computing Surveys (CSUR), vol. 32, issue 2, pp. 97-113, 2003.
5. D. Brumley and D. Boneh, *Remote Timing Attacks Are Practical*, 12th USENIX Security Symposium (USENIX Security 03), 2003, `https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical`
6. D. Brumley and N. Tuveri, *Remote Timing Attacks Are Still Practical*, Computer Security - ESORICS 2011, Lecture Notes in Computer Science, vol. 6879, pp. 355-371, 2011.
7. , B. Chen, Y. Wang, P. Shome, C. Fletcher, D. Kohlbrenner, R. Paccagnella and D. Genkin, *GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers*, 33rd USENIX Conference on Security Symposium - SEC '24, pp. 1117-1134, 2024, `https://gofetch.fail/`
8. P. Deutsch and A. Schiffman, *Efficient Implementation of the Smalltalk-80 System*, 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '84, pp. 297-302, 1984.
9. T. Downs, *Ice Lake Store Elimination*, 2020, `https://travisdowns.github.io/blog/2020/05/18/icelake-zero-opt.html`
10. Digital Equipment Corporation, *Small computer handbook*, 1970, `https://bitsavers.org/pdf/dec/pdp8/handbooks/SmallComputerHandbook_1970.pdf`
11. M. Flanders, R. Sharma, A. Michael, D. Grossman and D. Kohlbrenner, *Avoiding Instruction-Centric Microarchitectural Timing Channels Via Binary-Code Transformations*, 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '24, pp. 120-136, 2024.
12. A. Fog, *The microarchitecture of Intel, AMD, and VIA CPUs*, 2024, `https://www.agner.org/optimize/microarchitecture.pdf`
13. L. Geppert and T. Perry, *Transmeta's magic show*, IEEE Spectrum, vol. 37, issue 5, pp. 26-33, 2000.

14. Intel Corporation, *80386 – Hardware Reference Manual*, 1986, https://www.dosdays.co.uk/media/intel/1986_80386_Hardware_Reference_Manual.pdf
15. Intel Corporation, *Data Dependent Prefetcher*, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html
16. Intel Corporation, *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance*, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html
17. Intel Corporation, *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html
18. A. Kerckhoffs, *La cryptographie militaire*, Journal des Sciences Militaires, vol. 9, pp. 5-38 (Jan. 1883) and pp. 161-191 (Feb. 1883).
19. J. Kim, J. Chuang, D. Genkin and Y. Yarom, *FLOP: Breaking the Apple M3 CPU via False Load Output Predictions*, to appear at USENIX Security 2025, https://predictors.fail/files/FLOP.pdf
20. P. Kocher, *Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems*, Advances in Cryptology - CRYPTO' 96, Lecture Notes in Computer Science, vol. 1109, pp. 104-113, 1996.
21. P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, *Spectre Attacks: Exploiting Speculative Execution*, 2019 IEEE Symposium on Security and Privacy (SP), pp. 1-19, 2019.
22. M. Lipasti, C. Wilkerson and J. P. Shen, *Value locality and load value prediction*, ACM SIGPLAN Notices, vol. 31, issue 9, pp. 138-147, 1996.
23. MOS Technology, MCS 6500 Microcomputer Family – Hardware Manual, 1976, https://web.archive.org/web/20221106105459if_/http://archive.6502.org/books/mcs6500_family_hardware_manual.pdf
24. D. Osvik, A. Shamir and E. Tromer, *Cache Attacks and Countermeasures: The Case of AES*, Topics in Cryptology - CT-RSA 2006, Lecture Notes in Computer Science, vol. 3860, pp. 1-20, 2006.
25. T. Pornin, *BearSSL: a smaller SSL/TLS library*, https://www.bearssl.org/
26. JR. Sanchez Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison and D. Kohlbrenner, *Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data*, 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 347-360, 2021.
27. M. Schneider, D. Lain, I. Puddu, N. Dutly and S. Capkun, *Breaking Bad: How Compilers Break Constant-Time Implementations*, arXiv:2410.13489, 2024, https://doi.org/10.3929/ethz-b-000700923
28. *SpiderMonkey*, https://spidermonkey.dev/
29. R. Tomasulo, *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal of Research and Development, vol. 11, issue 1, pp. 25-33, 1967.
30. World Wide Web Consortium (W3C), *WebAssembly Core Specification*, https://www.w3.org/TR/wasm-core-2/