The `cryptoint` library

Daniel J. Bernstein[1,2]

[1] Department of Computer Science, University of Illinois at Chicago, USA
[2] Institute of Information Science, Academia Sinica, Taiwan
djb@cr.yp.to

Abstract. C/C++ code is often designed to run in constant time so that secret information is not leaked through timings. This code relies on a variety of replacements for secret branches, secret comparisons, and secret `bool`. However, this protection has been undermined by various "optimizations" in `gcc` and `clang` that sometimes introduce branches and timing variations into the assembly for C/C++ code where earlier compiler versions had generated constant-time assembly.

The `cryptoint` library provides functions such as `crypto_int64_max` with implementations designed to defend against such "optimizations". Some previous work aims at stopping compilers from introducing branches for conditional selection; `cryptoint` aims at stopping compilers from internally introducing any `bool` conditions in the first place. The `cryptoint` defenses include (1) usage of a global `volatile` zero variable for portable code and (2) assembly for various platforms.

The library is an almost-header-only library with automatic decisions between assembly and portable code, allowing simple inclusion in other libraries or in applications. C/C++ software for a wide range of cryptographic primitives has already been adjusted to use `cryptoint`, with the resulting binaries passing a variety of correctness tests and constant-time tests. Each function in `cryptoint` is subjected not just to conventional tests but also to equivalence checking via symbolic execution and SMT solving.

This paper surveys challenges and progress in producing constant-time code, and then explains the design and implementation of `cryptoint`.

1 Introduction

This paper introduces `cryptoint`, a software library providing various `{int,uint}{8,16,32,64}` operations to use in C/C++ code. For example,

This work was funded by the Intel Crypto Frontiers Research Center, and by the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP). "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s)." Permanent ID of this document: `c371d5977da0a47bdf31ee2b62c1316a3bf42122`. Date: 2025-04-24.

including `crypto_int64.h` and then calling `crypto_int64_max(x,y)` returns the maximum of two signed 64-bit integers `x` and `y`.

Why not write `x>y?x:y` or, as a C++ alternative, `std::max(x,y)`? Answer: `crypto_int64_max(x,y)` takes responsibility for avoiding data flow from `x` and `y` into timings, and tries hard to make sure that there is no such data flow, whereas a C/C++ compiler often produces such data flow and certainly does not take any responsibility for avoiding it.

The importance of avoiding such data flow is demonstrated by many attacks in the literature exploiting code that *isn't* constant-time: attacks that work backwards from attacker-visible timings to secrets. For example, [57], [94], and [79] exploit timing variations stemming from secret branch conditions; [16], [116], and [129] exploit timing variations stemming from secret array indices; and [44] exploits timing variations stemming from secret inputs to a division instruction. See [45] for an introduction to timing attacks and further references.

It is clear from various defense papers such as [108], [58], [9], and [54] how a compiler can explicitly support constant-time code, producing constant-time assembly given suitable source code. Why, then, do mainstream compilers such as `gcc` and `clang` not provide any such support? The answer appears to be a belief that constant-time code would consistently be a performance disaster so nobody would want to use it.

This belief is incorrect. Two decades ago, I used constant-time software to set new speed records for secret-key cryptography and public-key cryptography; see Section 2. Subsequent work has identified many more examples where constant-time software provides the best performance available—or at least close enough for all practical purposes.

Within today's software ecosystem, software that handles secret data is often designed to run in constant time. Sometimes the code is written in assembly (see Section 3); sometimes it is written in C or C++; sometimes it is written in other languages that can be viewed as competing with C and C++. But `gcc` and `clang` still do not explicitly support constant-time code. Even worse, "optimizations" in recent versions of these compilers sometimes create timing variations in code that was previously running in constant time; see Section 4.

The point of `cryptoint` is to provide—and, perhaps more importantly, centralize—defenses of C/C++ software against these compiler "optimizations". Many cryptographic implementations in the SUPERCOP benchmarking package from [46] have already been adjusted to use `cryptoint`; quantitatively, the implementations currently have a total of 7026 lines calling the `cryptoint` functions. Each version of `cryptoint` is released as part of SUPERCOP. The `cryptoint` library is also used in lib25519 [39], libmceliece [40], libntruprime [41], and the `sntrup761` component of OpenSSH [110].

Section 6 explains steps that have been taken to avoid timing variations and bugs in `cryptoint`. For comparison, Section 7 looks at subroutines in OpenSSL, BoringSSL, BearSSL, and Botan designed to be constant-time.

1.1 Recommendations

For any existing C/C++ code designed to run in constant time, I recommend switching to the `cryptoint` functions wherever the functions are applicable. For example, existing code saying `x>>31`, where `x` is a signed 32-bit integer, should be rewritten as `crypto_int32_negative_mask(x)`. Similarly, new C/C++ code designed to run in constant time should use `crypto_int32_negative_mask(x)` rather than `x>>31` or `-(x<0)`.

This is not a recommendation to use C and/or C++ and/or `gcc` and/or `clang` in the first place. This is also not a guarantee of safety. On the contrary: C and C++ have many dangerously sharp edges. The development processes for `gcc` and `clang` are approximately 180 degrees away from what I would recommend for high assurance. In such an environment, guarantees would be hard to justify.

What I find much more convincing is applying security tests to the resulting binaries. Many tools are available for testing constant-time behavior of code (see generally [87] and [71]), and some of the tools take binaries as input. In particular, SUPERCOP includes TIMECOP 2, which uses `valgrind` to scan for timing variations in large volumes of compiled cryptographic software; see Section 5. Similar tools are built into the test suites for lib25519, libmceliece, and libntruprime. These tests are not perfect—their path coverage is limited to whatever appears in dynamic runs; also, `valgrind` does not prohibit all of the variable-time instructions discussed in Section 3—but it seems feasible to close these gaps.

If binaries are subjected to sufficiently comprehensive tests then a compiler "optimization" that produces a timing variation will be caught before the variable-time binaries are deployed. This does not remove the value of `cryptoint`. Imagine a package maintainer running tests, finding a timing variation, and then being faced with the problem of fixing the variation, perhaps with a tight package-release deadline.[3] It is reasonable to expect that proactively using `cryptoint` will make any necessary patches easier, and will make these problems less frequent in the first place.

1.2 Keeping up with the compilers

A recent paper by Pornin [122] starts from the same problem: compilers adding some "optimizations" that turned constant-time code into variable-time code. That paper then claims 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" since "compilers get better over time" and "data-dependent shortcuts are a primary source of performance improvement".

Here is a data point that appears to support the idea that compilers evolve over time to produce faster code. In 2022, Kapoulkine [88] reported the following

[3] It may be possible to use a previous compiler version without the "optimization". However, my experience is that maintaining older compiler versions runs into difficulties because of instabilities in the surrounding software environment.

conclusion from running various benchmarks: "LLVM 11 tends to take 2x longer to compile code with optimizations, and as a result produces code that runs 10–20% faster (with occasional outliers in either direction), compared to LLVM 2.7 which is more than 10 years old."

Perhaps this somehow implies that compilers will eventually turn `cryptoint` into variable-time code—but, again, this won't be a security problem if binary testing is sufficiently comprehensive by then. Rolling out `cryptoint` addresses the current compiler threat, and simplifies the handling of any problems that might develop in the future. This is not a fool's errand: it is common-sense engineering.

1.3 Incentives for CPU designers

Pornin also appears to claim in [122, page 2] that "industrial secrecy" makes it unlikely that CPU manufacturers will provide "guarantees on timing-related characteristics". However, ARM's "data-independent timing" [10] (DIT) and Intel's "data-operand-independent timing" [85] (DOIT) are examples of such guarantees, as [122, page 21] admits.

The underlying picture of CPU design drawn in [122] is driven entirely by speed. In fact, CPU manufacturers also invest some resources in security. For example, most CPUs provide memory protection, even though this has a cost. As another example, Intel didn't ignore the attack demos from [16], [116], and [129]: it invested in ways to stop those attacks, such as (1) writing RSA software that carried out table lookups within cache lines (see Section 3.1) and (2) adding AES instructions to hardware.[4] These responses dealt with only a corner of the problem, but they still show Intel explicitly targeting goals beyond speed.

I commented twenty years ago, in [16], that CPU manufacturers "need to highlight every variation in their instruction timings, and to guarantee that there are no other variations. As this paper demonstrates, hidden CPU performance information is a security threat". The value of these guarantees becomes clear when there is an ecosystem of constant-time software—something that was essentially nonexistent back then but that is much more visible today. So it is unsurprising that ARM and Intel have started making guarantees. DIT and DOIT were announced in 2020 and 2022 respectively. The value of these CPU features is very easy to see from newer timing attacks blocked by DIT, such as [59] and [90].

Are we done yet? No: we still need better alignment between what CPUs are providing and what software is relying upon. But the overall shape of the solution is clear; the necessary software is mostly in place; and supporting the solution is turning into a competitive advantage for CPU manufacturers.

[4] See [77]: "Beyond improving performance, the AES instructions provide important security benefits. By running in data-independent time and not using tables, they help in eliminating the major timing and cache-based attacks that threaten table-based software implementations of AES. In addition, they make AES simple to implement, with reduced code size, which helps reducing the risk of inadvertent introduction of security flaws, such as difficult-to-detect side channel leaks."

2 Speed

There is nothing new about hyping the tension between constant-time software and performance: this was already brought up three decades ago as a reason to not even try building such software. This section reviews what happened.

One of the attack papers mentioned in Section 1 was an influential 1996 paper [94] by Kocher.[5] That paper also considered defenses. As part of this, the paper listed various obstacles to building constant-time software, and claimed without quantification that such software is "likely to be slow; many performance optimizations cannot be used since all operations must take as long as the slowest operation".[6]

I challenged this claim several years later, introducing a new cipher Salsa20 that used constant-time software to set new speed records for encryption on a variety of CPUs. I explicitly avoided any data flow from secrets to branches, table lookups, and other instructions that (at least on some CPUs) take variable time; I paid attention to this instruction-selection constraint as part of algorithm selection. See [24, Section 2]. The original postings were [16, 2004 version, Section 5], [17, Section 2], [18], and [20] ("Why not switch to a cipher that avoids these problems?").[7]

I then used constant-time software to set new speed records for elliptic-curve cryptography. See [19] and [21]. I again avoided "all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings" (to quote [21]); I again paid attention to this constraint when I selected higher-level algorithms.

There are, as noted in Section 1, many newer examples of software designed to run in constant time. For example, Amazon's s2n-bignum library includes fast constant-time software for Curve25519 (the elliptic curve introduced in [21]) on current 64-bit CPUs from AMD, Intel, and ARM. A notable feature of s2n-bignum's Curve25519 code is that it is accompanied by a computer-checked proof [81] of producing the correct output for all inputs, assuming a particular (plausible) specification for how each CPU instruction behaves.

[5] Timing attacks had been previously described in, e.g., [57], but cryptographers were generally unaware of the threat until the attack examples in [94]. Subsequent literature often miscredits the entire concept of timing attacks to [94].

[6] The same paper also claimed that a "better solution" was to randomly mask (or "blind") secrets: for example, storing a secret `uint32` s as a random `uint32` r along with $r + s$. The hope is that timings of computations will then look random, not revealing the actual secrets. However, the words "better" and "solution" here are both controversial. Masking has its own software-engineering complications, its own performance issues, and an unclear level of security; masking has not removed the interest in constant-time software. On the other hand, masking might help against other side channels; constant-time software has not removed the interest in masking. Masking and constant-time software can also be used together. For more information about masking, see, e.g., [103], [66], and [97].

[7] These postings also pointed to earlier ciphers as having the same security feature in retrospect—but those ciphers were slower, so they did not challenge Kocher's claim.

2.1 The importance of reoptimizing algorithms

Algorithm designers and programmers accustomed to using variable array indices will naturally think that avoiding such indices is a performance disaster. For example, inside the RC4 stream cipher, there are frequent accesses to unpredictable positions in a 256-byte array. Rewriting each access as arithmetic on all 256 array positions would make RC4 much slower.

Similar comments apply to a wide range of algorithms outside cryptography. Imagine the consequences of rewriting each array access as arithmetic on all possible array positions inside heap sort, for example, or inside radix sort.

How, then, did I use constant-time code to set new speed records in 2018 [30] for sorting `int32` arrays in cache on Intel CPUs? How does this code remain competitive with newer sorting libraries?

A typical Intel CPU core can read two `int32` inputs from variable array positions each cycle—but the same core can carry out two 256-bit `VPMAXSD` operations per cycle, each of which computes $(\max\{a_0, b_0\}, \ldots, \max\{a_7, b_7\})$ where a_i and b_i are each `int32`, for a total of 16 max operations on 32 `int32` inputs per cycle. What I did in [30] was arrange "sorting networks", which are constant-index sorting algorithms, in a way that takes advantage of this parallel arithmetic. Insisting on constant indices incurs extra comparisons, but this expense is outweighed by the extra parallelism.

In other words, while sorting networks are much slower than other sorting algorithms in a naive count of `int32` operations, they are competitive with other sorting algorithms on Intel CPUs. This is an example of a common phenomenon in the literature on algorithms: the answer to the question "Which of these algorithms is better?" is often "This depends on the model of computation and the choice of cost metric".

Why does Intel invest so much hardware area in parallel max operations? The critical point here is that reading an array at a variable index has much higher inherent hardware cost than simple integer arithmetic, especially as the arrays grow. For example, a single read from a 4KB cache bank is an operation not just on the address but also on the 32768 bits stored in the cache bank. For comparison, computing the max of two `int32` inputs is an operation on just 64 bits, and computing 8 of those in parallel is an operation on just 512 bits.

One shouldn't extrapolate from the sorting example to conclude that *all* software will gain performance from avoiding variable array indices. But one also shouldn't use algorithms optimized for variable array indices as predictors of the performance of algorithms optimized for constant array indices.

Similarly, conditional branches have low cost in a naive model of computation but are resource-intensive in hardware. A single hard-to-predict branch can cost several cycles on average, time that could otherwise have been used to run many arithmetic instructions. One shouldn't use algorithms optimized for a naive model as predictors of the performance of algorithms optimized for constant instruction flow.

3 Machine instructions

Fast constant-time software is often written in assembly; this applies, for example, to [24], [21], and [81]. The fact that this software avoids data flow from secrets to timings relies on the same fact for the machine instructions used in the software. But wait: how do we know which machine instructions have timing variations?

The traditional answer to this question starts with documentation of various ways that CPUs make software run faster by taking shortcuts for common inputs. For example:

- A simple CPU might incur the cost of accessing DRAM on each load/store instruction, but CPUs typically add an SRAM cache (or multiple layers of such caches). The shortcut here is that, in the common case of `x[i]` being in cache, access to `x[i]` is a fast SRAM access instead of a slow DRAM access. That's why [21] forbids "input-dependent array indices".
- Branch timing is generally not balanced between branches, whether or not the number of instructions in each branch is balanced. Some specific reasons for branch conditions to leak into timings are explained in [69, page 13]. More fundamentally, secret instruction pointers are examples of secret array indices and thus have to be assumed to leak. See, e.g., [7] for an attack extracting instruction pointers from timings.

So we have a list of instructions to avoid. But what if this list is incomplete, because of an incomplete understanding of speed features in current CPUs? What if the list is rendered obsolete by changes in CPUs? (See, e.g., the potential future changes analyzed in [14].) One can build all computations from just logic operations and constant-distance shifts (as noted in, e.g., [23, page 30]), but what happens if this produces speed complaints?

The reader of [122] is led to believe that these are new questions. The rest of this section illustrates that these questions appeared many years earlier—and that we are finally seeing constant-time promises from ARM and Intel, the aforementioned DIT and DOIT.

3.1 Timing variations within cache lines

In May 2005, in response to one of the attack papers [116] mentioned in Section 1, an OpenSSL patch [131] took the following "consttime" approach to table lookups `x[i]` for situations where `i` is between 0 and 63: (1) align arrays to 64-byte cache lines; (2) decompose lookups in larger-than-64-byte arrays into lookups within cache lines, by transposing array contents. Over the next several years, various Intel personnel recommended this approach:

- [75] (which tweaked the software to look up 2 bytes at a time rather than 1 as in OpenSSL; in context, `i` was limited to 31) labeled the approach as "constant run-time".

- [78] (which tweaked the software to look up 4 bytes at a time, with `i` limited to 15) said that "modular exponentiation code need to be written in a way that its memory access patterns (at the granularity of a cache line) do not leak secret information".
- [55, page 9] recommended avoiding secret-dependent "memory access (at coarser than cache line granularity)".

This approach is safe in a simplified model of CPUs where the time for memory access depends only on which 64-byte cache line is being accessed. However, I had already warned in [16, 14 April 2005 version, Sections 14 and 15] that this model is broken by other optimizations such as store-to-load forwarding in Intel's Pentium III and multiple cache banks in AMD's Athlon.[8] Schwabe and I presented a demo in [49] extracting a secret from a small test program. OpenSSL did not stop using this approach until Yarom, Genkin, and Heninger in [134] presented a complete attack, CacheBleed, extracting secret RSA keys from timings of OpenSSL on an Intel Xeon E5-2430.

3.2 Timing variations in multipliers

Cryptographic software often makes heavy use of integer multiplications. This works well on CPUs that include the fastest circuits for integer multiplication, since those circuits are naturally constant-time. Those circuits are also easily justified by many other applications of multiplication. CPUs with constant-time multiplications are so common that software aiming to be constant-time often considers solely those CPUs.

However, some CPUs spend less area on multipliers, and it is easy to see how those CPUs can save time in multiplications by applying input-dependent shortcuts. See, e.g., [17, page 2] mentioning the documented timing variations in multiplications on the Motorola PowerPC 7450 (the CPU in the 2001 version of Apple's Power Mac G4) and explaining the security impact of these variations. For newer examples, see [76] (finding undocumented timing variations on the ARM Cortex-M3) and [120] (surveying CPUs).

Sometimes one can avoid timing variations in integer multiplication by switching to faster floating-point multipliers. On the other hand, my paper [21, page 10] warns that there are input-dependent timings for basic floating-point operations on some CPUs, such as "the IBM PowerPC RS64 IV, which

[8] As far as I know, Intel never documented store-to-load forwarding for the Pentium III microarchitecture. On the other hand, Fog [69, Sections 6.5 and 6.9] observed timing variations through experiments, and Intel [86, page 2-43, "Aliasing cases in the Pentium M processor"] documented variations for the subsequent Pentium M microarchitecture. Similarly, AMD [8, Section 5.8] documented cache-bank conflicts for the Athlon 64 microarchitecture but, as far as I know, not for the original Athlon. I pointed out the security impact of cache-bank conflicts in [16, 11 November 2024 version], and pointed out the security impact of store-to-load forwarding in [16, 14 April 2005 version]. A warning about cache banks then appeared without credit in [113, 14 August 2005 version, Section 3].

takes an extra cycle to multiply by 0". Some newer papers such as [95] categorically recommend against floating-point operations on secrets, although this recommendation is based primarily on timing variations in, e.g., divisions and subnormals, neither of which is important for fast cryptography.

There are some code-analysis tools, such as my `saferewrite` tool introduced in [32], that issue warnings for secret multiplications. It is clearly feasible to build software that uses multiplications on CPUs that provide constant-time multipliers, while falling back to non-multiplication code for portability to other CPUs. On the other hand, this is not common practice today, and the necessary fallback code usually still needs to be written. Presumably current software using multiplications has exploitable timing leaks on some CPUs.

The tension that variable-time multipliers create between security and speed is resolved by cryptosystems where multiplications are unimportant for speed: see, e.g., [17] and [40].

3.3 Timing variations in shifts

Fast barrel shifters naturally take constant time, and require much less hardware area than fast multipliers, but still might not be present on a small enough CPU. For example, on the Intel 8088 CPU in the original IBM PC, the instruction to shift left by c bits took $8 + 4c$ cycles; see [84, page 6-50]. Intel CPUs included barrel shifters starting with the 80386 in 1985, but small CPUs without barrel shifters such as the AVR continued to appear. I generally don't bother targeting those CPUs, but we'll find shift issues reappearing when we look at compilers; see Section 4.11.

3.4 Data-dependent caching

Some recent CPUs have added data flow from arbitrary memory contents to timing, for example with data-dependent prefetching or load-value prediction. The GoFetch attack from [59] exploits data-dependent prefetching to extract secrets from a variety of cryptographic computations running on Firestorm cores on Apple M1 CPUs. The FLOP attack from [90] exploits load-value prediction on Apple M3, M4, and A17 Pro for "end-to-end attacks that read arbitrary 64-bit addresses, allowing us to recover sensitive data across webpage origins" in both Safari and Chrome.

Fortunately, one can disable data-dependent prefetching on the M1 by pinning computations to the Icestorm cores, or (as discovered by Martin [105]) by having the operating-system kernel set bit 30 of `SYS_APL_HID11_EL1`. For M3, [59] reports observing that ARM's unprivileged DIT option disables data-dependent prefetching, and [90] reports observing that DIT disables load-value prediction.

A 2023 posting [80] from an Intel engineer indicates that, on CPUs that support Intel's DOIT option, setting the option in the operating-system kernel disables "Data Dependent Prefetchers (DDP)" and "Some Fast Store Forwarding Predictors (FSFP)". The word "some" is concerning. It is also concerning that

there is no mention of store elimination for all-zero cache lines, a data dependency that Downs [65] observed from benchmarks of Intel Skylake and Ice Lake.

It is claimed in [59, Section 8] that disabling data-dependent prefetching "will incur heavy performance penalties", but this claim is neither quantified nor justified in [59]. Measurements of some cryptographic computations in [68, Section 5.2] on Apple M1 and M3 found that the DIT slowdown was so small as to be lost in measurement noise. Measurements of Safari rendering in [90, Section 7] found DIT producing "an overhead of 4.5% on the Speedometer 3.0 benchmark". It would be useful to collect more evidence regarding the performance impact of DIT and DOIT.

3.5 Lists of safe instructions

The DIT and DOIT options, despite their "data-independent timing" and "data-operand-independent timing" names, do not guarantee that timing is independent of *all* instruction operands. For example, timings still depend on branch conditions, load/store addresses, and division inputs. The documentation for ARM's DIT and Intel's DOIT includes lists of instructions guaranteed to be safe when those options are enabled.

Presumably ARM and Intel assembled these lists as lists of instructions where they do not see noticeable prospects for data-dependent speedups—and presumably their assessments of this are much less error-prone than typical public assessments. The aforementioned 2023 posting [80] says that there are no current timing varations in these instructions on any Intel CPUs, and "no plans for any processors where this behavior would change". This does not remove the importance of enabling DIT and DOIT; see Section 3.4.

4 Compilers

Instead of considering software written in assembly, let's now consider software written in C or C++. How do we know that supposedly constant-time C/C++ software doesn't have timing variations?

This question is more complicated than the question about assembly in Section 3. Timings of C/C++ software depend not just on the timings of machine instructions but also on which machine instructions a compiler decides to use given this software. This has a dangerous dependence on compiler options; see Section 4.1. After Section 4.1, this section is organized chronologically, tracing what the literature says about risks and countermeasures.

As noted in Section 1, sufficiently comprehensive binary analysis can catch a compiler's variable-time code before the code is deployed, but one is then faced with the problem of fixing the code. Furthermore, the current situation is that binaries are often deployed *without* analysis. So it is useful, both short-term and longer-term, to understand patterns in source code that lead compilers to generate variable-time code.

4.1 The trust trap

A simple compiler will convert a branch in C into a branch in assembly, will convert a variable index in C into a variable index in assembly, will convert a division in C into a division in assembly, etc.

However, "optimizing" compilers have learned some of the tricks that an assembly programmer can use to gain speed, and in particular will *sometimes* eliminate branches, variable indices, divisions, etc. Programmers looking at the resulting assembly might think that a C snippet is safe, not realizing that the same snippet becomes unsafe when one switches to a different compiler, a different set of compiler options, or a different target architecture.

Consider the following example of C code with an input-dependent branch condition:

```
if (x > y)
  result = x;
else
  result = y;
```

Running this through `clang-14 -O0 -target amd64` produces assembly with an input-dependent branch condition:

```
        movl    -4(%rbp), %eax
        cmpl    -8(%rbp), %eax
        jle     .LBB0_2
        movl    -4(%rbp), %eax
        movl    %eax, -12(%rbp)
        jmp     .LBB0_3
.LBB0_2:
        movl    -8(%rbp), %eax
        movl    %eax, -12(%rbp)
.LBB0_3:
```

Rewriting the above branch using C's ternary if-then-else operator—

```
result = x > y ? x : y;
```

—produces essentially the same assembly. However, switching from `clang -O0` to `gcc-12 -O0` converts the ternary example above into a conditional move, removing the timing variation:[9]

```
movl    -24(%rbp), %edx
movl    -20(%rbp), %eax
cmpl    %eax, %edx
cmovge  %edx, %eax
movl    %eax, -4(%rbp)
```

[9] Conditional moves are like multiplications in that one can see how a CPU designer might sometimes want to introduce timing variations. For many years I was avoiding conditional-move instructions. However, Intel's DOITM documentation lists conditional moves as constant-time.

Higher optimization levels for `clang` and `gcc` also convert the first example into a conditional move.

Does this mean that one should recommend `gcc` over `clang`? Or that one should recommend higher optimization levels? Perhaps, but these recommendations would not be enough to eliminate branches in assembly. A slightly more complicated operation such as

```
if (x > y)
  result = x;
else
  result = x*y;
```

turns into a conditional branch with `gcc -O3`.

4.2 2011: avoiding variable-time source code

NaCl [48] was written to avoid all "data flow from secrets to load addresses" and all "data flow from secrets to branch conditions". NaCl reused assembly that followed the same rules from [24], [21], etc., but it also added C code for portability.

NaCl's coding rules [47] view all C comparisons as branch conditions: "Do not use secret data to control a branch. ... Even on architectures that support fast constant-time conditional-move instructions, always assume that a comparison in C is compiled into a branch, not a conditional move. Compilers can be remarkably stupid." As noted in Section 4.1, an "optimizing" compiler *could* eliminate branches, but this rule avoids *trusting* the compiler to do this.[10]

This rule prohibits the three C examples from Section 4.1—the two branch examples and the ternary rewrite. It also prohibits

```
result = x > y;
```

since this is a comparison. Today one can find, e.g., `clang -O0 -target sparc` turning this comparison into a branch. There are other architectures where the compiler *sometimes* converts this comparison into suitable non-branch machine instructions, but this depends on the context: for example, `gcc-14 -O3` converts

```
result = (x > y) * (b * c - a) + a;
```

into a branch. The rule from [47] says to avoid the comparison in the first place.

As an expository matter, it seems important to include examples emphasizing that `x>y`, `!x`, etc. are prohibited whether or not there are C branches. The "avoid branchings controlled by secret data" rule from [11] and [12] does not obviously prohibit comparisons. Also, for a programmer who knows that in C the result of a comparison is an `int`, the rule from [119] saying "Avoid boolean types (e.g. the C99 type `_Bool`)" does not prohibit comparisons.

[10] Similarly, compilers *sometimes* convert a division by a constant into multiplication, but *trusting* the compiler to do this is dangerous; see [44].

4.3 2015: a variable-time software multiplier

A 2015 paper [89] from Kaufmann, Pelletier, Vaudenay, and Villegas has title "When constant-time source yields variable-time binary: exploiting Curve25519-donna built with MSVC 2015". A closer look shows that the code considered in [89] had `int64` operations that, when compiled by Microsoft's compiler for 32-bit x86, were converted into calls to Microsoft's 32-bit `int64` library, specifically `llmul.asm`, where Microsoft had used data-dependent branches.

Should one recognize `llmul.asm` as source code, and fix it to avoid branches? Avoid 32-bit x86 as a target? Avoid the Microsoft compiler? Perhaps the most robust answer is to pick cryptosystems that naturally avoid multiplications, given that there are other environments where multiplications take variable time; see Section 3.2.

As a side note, some tools to check code for constant-time behavior will not check calls to external subroutines. However, `valgrind`-based binary-analysis tools, starting with Langley's ctgrind [98] and continuing through current tools such as TIMECOP, do check such calls.

4.4 2018: hypothesizing an arms race

A 2018 paper [128] from Simon, Chisnall, and Anderson stated the following: "A compiler upgrade can suddenly and without warning open a timing channel in previously secure code." Later we'll see some post-2018 examples of this happening.

[128] claimed that [89] was an existing example, but provided no evidence that this was the result of a compiler upgrade. [128] also gave a toy example using a secret `bool` to select between two values, but presumably a programmer creating a secret `bool` from secret byte arrays would also have been violating [47]'s prohibition of secret comparisons. [128] admitted that "an extra layer of obfuscation used by cryptographers is to eradicate `bool` completely in critical code"; convincing examples of compiler-induced problems should thus start with source code that does not use `bool`.

4.5 2018, continued: unofficial compiler patches

[128] continued as follows: "This arms race is pointless and has to stop." [128] patched `clang` to support a new function `__builtin_ct_choose`, which was defined as a constant-time ternary if-then-else operator. One could, for example, write the maximum of `x` and `y` as `__builtin_ct_choose(x>y,x,y)`.

[128] concluded that "The developer can use a single function (instead of juggling between the 37 in OpenSSL); this should also improve code readability and productivity". However, recall from Section 4.2 that `x>y` can already produce a branch. Constant-time code needs replacements for `x>y`, for all of the other comparison operators in C, and, as we'll see later, further operations. So the

patch from [128], even if widely adopted, would be missing the point of what those 37 functions in OpenSSL were trying to address.

Seven years later, the patch from [128] does not seem to have been maintained. See [83], [51], and [133] for compiler maintainers discussing the speed impact of this proposal and related proposals. Today developers can *sometimes* use a similar construction `__builtin_unpredictable(x>y)?x:y`, which will *sometimes* generate conditional-move instructions but does not promise to do so, never mind the more fundamental problem of `x>y` sometimes already generating a branch.

Perhaps compiler maintainers will eventually be convinced to make promises, the same way that CPU manufacturers have begun making promises. However, to the extent that compiler upgrades are already breaking constant-time code, [128] is not dealing with the immediate security problem. Even when the security problem is detected by binary analysis, [128] is not helping the developer write replacement code today.

4.6 2019: value barriers

A 2019 patch [15] by Benjamin to BoringSSL added some usage of "value barriers", subsequently copied into OpenSSL. These value barriers, credited in [15] to Carruth, have the form

```
__asm__("" : "+r"(a) : /* no inputs */);
```

telling the compiler to emit empty inline assembly, and telling the compiler that this assembly reads and modifies the variable `a`—which the assembly doesn't actually do, but the `gcc` documentation says "GCC does not parse the assembler instructions themselves and does not know what they mean or even whether they are valid assembler input". See [72].

It is not clear whether this comment from the documentation is meant as a long-term commitment. The comment is followed by "However, it does count the statements". Even without the effort of serious assembly parsing, nothing stops `gcc` from recognizing and discarding empty inline assembly. On the other hand, as far as I know, no version of `gcc` or `clang` does that.

The value barriers were, in particular, applied in [15] to BoringSSL's `constant_time_select_w` function, with the following explanation: "Clang recognizes this pattern as a select. While it usually transforms it to a cmov, it sometimes further transforms it into a branch, which we do not want." It is not clear whether this was an observed problem for cryptographic software at that point, or simply a hypothetical problem; I have not found any security warnings accompanying [15].

For developers using `gcc` or `clang`, this usage of an existing compiler feature resolved the deployment problem of `__builtin_ct_choose`. However, [15] had the same target as `__builtin_ct_choose`, namely replacing C's ternary if-then-else operator. Recall from Sections 4.2 and 4.5 that this target is too narrow.

The constant_time_select_w function uses a mask, an integer that is either -1 or 0, to select one of two values. For the compiler to recognize this as a selection, the compiler has to recognize that this is a two-valued input in the first place—and at that point the compiler can already generate branches, so trying to protect the selection is too late.

4.7 2020: hypothesizing that binary analysis is required

A 2020 paper [62] by Daniel, Bardin, and Rezk stated the following: "we discovered that gcc -O0 and backend passes of clang introduce violations of CT in implementations that were previously deemed secure by a state-of-the-art CT verification tool operating at LLVM level". The main point of [62] is another tool to check whether binaries are constant-time.

The examples named in [62] are functions called ct_sort and ct_sort_mult. The corresponding source code appears to be [61] (see also [63, Appendix C] for partial confirmation), specifically the functions sort2_negative and sort2_multiplex. Each function has a comparison such as in2[0] < in2[1], violating the prohibition from [47].

4.8 2024-04: public warnings about shifts

I issued an alert in April 2024 [35] about an "optimization" that had been added to gcc in 2021 that turns (-x)>>31, pure arithmetic operations applied to a signed 32-bit integer x, into -(x>0). As in Section 4.1, the comparison here sometimes ends up as a branch in assembly. Recall from Section 4.6 that [15] was already trying to protect against compilers introducing branches into arithmetic, but was protecting only if-then-else selections, which (-x)>>31 is not.

I had found this by tracking down the source of some timing leaks in cryptographic software submitted to the KpqC [96] competition. The leaks had been caught by TIMECOP—but, again, the goal here is not merely to detect the problem; one wants to understand and proactively avoid the problem.

For an assembly programmer, replacing two arithmetic operations in (-x)>>31 with a branch sounds like a speed loss. Even on an architecture that can compare and branch in one instruction, even in a context where the negation of x>0 can be skipped, branch-misprediction penalties seem very likely to outweigh any savings. Why, then, did gcc add this "optimization"?

Checking the history of this gcc patch [60] shows that the justification provided for the patch consisted of one code snippet compiled for 64-bit ARM with one set of compiler options before and after the patch:

- The code snippet replaced each entry x[i] in an array with (-x[i])>>31.
- Before the patch, the code snippet was compiled to a loop of vector load, vector negation, vector right shift, and vector store.
- After the patch, the code snippet was compiled to a loop of vector load, vector cmgt, and vector store.

The "optimization" did not produce a branch *in this example*: the transformation of `(-x[i])>>31` into `-(x[i]>0)` was composed with another transformation of `-(x[i]>0)` into `cmgt`. The same "optimization" is bad for performance in various other examples, but [60] did not cover any other examples.

The simplest way to try to justify the patch, starting from the example in [60], would be the following extrapolation: this branch has been eliminated, so it will always be eliminated. But this is not a valid extrapolation: it falls into the trap explained in Section 4.1.

The patch could still be a performance win overall if the branch is eliminated *often enough* for the patch to save more CPU time than it loses. On the other hand, given that this patch *can* add branches that damage performance, presumably it would have been better for performance to instead add a transformation of `(-x[i])>>31` directly into `cmgt`, without ever crossing the line into introducing a comparison.

In any event, [60] did not investigate these broader issues. What one sees in [60], and in random samples of further "optimizations" added to `gcc` and `clang`, is that maintainers of these compilers measure an "optimization" as successful if they can find *any* example where the "optimization" saves time.

Note that `(-x)>>31` and `-(x>0)`, when compiled in the obvious way for typical CPUs, produce different results if `x` is -2^{31}. The compiler's excuse for an "optimization" that changes outputs is that, according to the C standard, `-x` is undefined in this case, and the compiler is free to do whatever it wants in cases of undefined behavior. Fortunately, both `gcc` and `clang` support an option `-fwrapv` that defines integer behavior as twos-complement,[11] which, among other benefits, blocks this particular "optimization". However, even when developers are already using `-fwrapv` or are satisfied to add it, we'll see below that this does not address the broader problem of compilers introducing branches.

4.9 2024-04, continued: splitting shifts

A few days later I released version 20240425 of SUPERCOP, including a shift-defense mechanism as a tweak to my `inttypes` library, the predecessor of `cryptoint`. I commented in the release announcement [36] on the magnitude of the problem:

> There are, presumably, many crypto implementations whose constant-time behavior is broken by this "optimization". I recommend replacing `>>31` in C code with a call to `crypto_int32_negative_mask()` from `crypto_int32.h`, and similarly using the other `crypto_{int,uint}*_*_mask()` functions whenever

[11] Other similarly useful options include `-fno-delete-null-pointer-checks`, `-fno-strict-aliasing`, and `-fno-strict-overflow`. See also [117], which provides patches that try to remove *all* "optimizations" from `clang` that rely on undefined behavior, and which provides benchmarks that seem to amply justify eliminating these "optimizations", given the evidence from, e.g., [132] of damage caused by these "optimizations".

> possible, so any necessary future defense against the dark optimizers can be handled centrally inside the *_mask() implementations rather than requiring widespread code changes.
>
> I expect we'll see more of these "optimizations": it's becoming more and more common for programmers to convert branches into shifts etc. (often for improved vectorization, sometimes to remove timing variations), and this makes it more and more likely that the compiler writers looking around for examples of code to "optimize" will see these shifts and have the bad idea of reintroducing bool. The compiler writer's mindset says that this is "strength reduction" and that any useful optimizations such as vectorization are something the compiler will figure out.

The overall message here is similar to [94, Section 9] saying that "compiler optimizations" can "introduce unexpected timing variations", and more specifically [128] hypothesizing an arms race. But the recommended actions are different. My recommendation is to change the >>31 code so that the compiler no longer sees that there are only two possibilities for the result.

Originally inttypes was simply a wrapper declaring types such as crypto_int32, for portability to platforms without stdint.h. I started adding some centralized constant-time functions to inttypes in 2021, for example defining crypto_int32_negative_mask(x) as x>>31. What I did in the release announced in [36] was redefine crypto_int32_negative_mask to call two separately compiled functions, one of which shifted right by 5 bits, and the other of which shifted right by 26 bits.

I was dissatisfied with this given that compilers have -flto options that break the traditional concept of separate compilation. A few weeks later, I released a new version of inttypes in [37] with a different defense, which is also used in cryptoint and is described in Section 6.2.

4.10 2024-06: public warnings about bit tests

A posting in June 2024 [123] by Purnal reported successful extraction of secret keys from timings of the reference Kyber-512 software compiled with various optimization levels of clang-15 (released September 2022) and newer, specifically because clang had turned the lines

```
mask = -(int16_t)((msg[i] >> j)&1);
r->coeffs[8*i+j] = mask & ((KYBER_Q+1)/2);
```

(inside loops over i and j) into a branch.

A bachelor's thesis [64] by Dankbaar had already been posted on 25 April 2024 (according to server metadata) with examples of TIMECOP failures from clang-15 and newer, in particular with some &1 examples. I did not hear about [64] at the time, but I did hear about [123]. I skimmed the clang code for "optimizations" that convert masks into comparisons, and easily found examples.

I originally blamed this Kyber disaster on `combineShiftAnd1ToBitTest` in `llvm/lib/CodeGen/SelectionDAG/DAGCombiner.cpp`. The first version of this function was added in a 2019 `clang` patch from [114]. I now think—after tracing the compilation of this code snippet with

```
clang -O1 -S snippet.c -mllvm -print-changed -mllvm -debug
```

using a version of `clang` patched for extra debugging information and compiled with `-DCMAKE_BUILD_TYPE=Debug`—that this disaster was actually triggered by a July 2022 `clang` patch from [115], which converts "`(-(X & 1)) & Y`" into "`(X & 1) == 0 ? 0 : Y`". But I haven't bisected commits to be sure about this.

Version 20240625 of SUPERCOP included a new `inttypes` release with functions such as `crypto_int32_bottombit(x)` meant as substitutes for `x&1`. Section 6 explains how the implementations of these functions defend against compiler "optimizations". This version of SUPERCOP also added continual scans for timing variations in compiled code; see Section 5.

4.11 2024-08: public warnings about double-length shifts

In August 2024 [38], I posted the results of an experiment where I patched the "optimizer" in `clang` to detect any intermediate occurrences of `&1`, `>>31`, etc., and then ran SUPERCOP. The patch produces remarks such as the following:

```
x.c:3:5: remark: clang-vs-clang: clang sees signed>>(bits-1);
please take this away before clang does something bad
[-Rpass-analysis=clang-vs-clang]
    3 |   x >>= 31;
      |     ^
```

A patched compiler can catch problematic source-code constructions before the constructions turn into branches visible to binary-analysis tools such as TIMECOP. On the other hand, this particular patch does not understand the distinction between secret values and public values. The results of the experiment turned out to have many false positives from `clang` internally generating `&1` for public branch conditions, and most of the true positives were easy enough to see from source code (as in Sections 4.8 and 4.10), so the patch in its current form is probably not a time-saver for cryptographic developers.

This does not mean that the experiment was useless. I pointed out in [38] that right shifts of `int128_t` values by 64 bits were triggering the `>>` warning:

> Sure, makes sense that the implementation of `int128` is internally using a 63-bit right shift of the top 64-bit word to figure out the sign; but what happens if Clang adds GCC-like support for converting 63-bit right shifts into `bool` and then into conditional branches? Suddenly all sorts of `int128` code will be variable-time, much like what the 2015 paper was claiming but this time really with no `bool` in the source. I think the easiest way to protect against this at the source level is to

avoid the compiler's existing implementation of int128 in favor of some crypto_int128 functions. A side advantage of writing those functions is that crypto_int128, unlike the int128 in GCC and Clang, will work on small 32-bit platforms.

Implementing int128 is beyond the current scope of cryptoint, but I had already included shlmod and shrmod functions in inttypes, and have taken further steps in cryptoint to address the risk of compilers turning int64 shifts into branches on 32-bit platforms. See Section 6.3.

4.12 2024-10: more shift examples and bit-test examples

An October 2024 paper [126] by Schneider, Lain, Puddu, Dutly, and Čapkun reported the results of a one-time test (using another binary-analysis tool) for timing variations in various compiled libraries. All of the specific variable-time code samples linked from [126, Appendix] are following patterns already pointed out in earlier announcements, and some of the examples are even violating the 2011 rules from [47]:

- cmovznz4 from HACL*: This uses a right shift of an integer down to 1 bit, as in Section 4.8.
- point_mul from HACL*: Same.
- bn_mul_comba8 from BoringSSL: This has a conditional branch, violating [47].
- bn_sqr_comba4 from BoringSSL: Same.
- fiat_p256_point_add from BoringSSL (imported from Fiat): This has comparisons (for example, !arg1), violating [47].
- hex_decode from Botan: This uses a right shift of an integer down to 1 bit (in expand_top_bit), as in Section 4.8.
- donna128 from Botan: This has a comparison l<x.l, violating [47].
- key_schedule from Botan: This has an &1, as in Section 4.10.
- modpow_opt from BearSSL: This uses a right shift of an integer down to 1 bit (in EQ), as in Section 4.8.
- br_i31_muladd_small from BearSSL: Same.
- check_scalar from BearSSL: Same.

[126, Table II] says that the tests in [126] detected timing variations in 15268 function-target-compiler combinations for these libraries, plus 2 combinations for libsodium. I did not find details of the libsodium issue in [126].

[126] portrayed its results as showing failures of "state-of-the-art defensive programming techniques employed for side-channel resistance". There was no reference in [126] to the extensive test results that were already online from SUPERCOP, or to the defenses that were already integrated into lib25519, libmceliece, and libntruprime: by the time [126] appeared, these libraries already made systematic use of cryptoint, and already included valgrind-based tests of the compiled binaries. A more reasonable conclusion from [126] is that existing defense mechanisms need to be more widely deployed.

5 SUPERCOP and TIMECOP

The aforementioned SUPERCOP is a benchmarking framework that has been operating for many years, with results continually updated on centralized web pages [46] for new software, new computers, and new tests. Currently SUPERCOP includes 4601 implementations of 1430 different cryptographic primitives in hundreds of different families. For example, `kyber768` and `kyber1024` are two primitives in the same family; there are three[12] compatible implementations of `kyber768` in SUPERCOP.

All 4601 implementations support SUPERCOP's `crypto_*` API. Lange and I had introduced this API in 2008 to unify the interfaces for a broad range of cryptosystems and to carry out a wide range of tests, going beyond previous efforts such as NESSIE, eSTREAM, our BATMAN, and various library APIs. Our main emphasis at the beginning was speed tests (see, e.g., [22], [25], [26], [27], [28], [29], and [33, Section 2.10]), but SUPERCOP also included various functionality tests, and over the years has gradually added further tests, such as various mutation tests. SUPERCOP's tests are often far ahead of what one finds in library test suites and in papers.[13]

Most importantly for this paper, SUPERCOP's tests include TIMECOP scanning for timing variations, as mentioned in Section 1.1. TIMECOP is applied to implementations that are marked as having a *goal* of being constant-time; currently this means 1451 of the 4601 implementations. Each implementation is tested with various compiler options. One computer (`rome0` with SUPERCOP 20250415) currently has

- 9192 implementation-compiler pairs passing TIMECOP,
- 117 implementation-compiler pairs failing TIMECOP, and
- 139 implementation-compiler pairs triggering TIMECOP errors that stop TIMECOP from issuing pass/fail results.

Another computer, `saber214`, has an AMD Bulldozer CPU and currently has 4828 TIMECOP errors, mainly because compilers often generate XOP instructions for that CPU while `valgrind` doesn't support XOP instructions. Most computers have fewer TIMECOP errors.

The online results from TIMECOP include pages designed to help implementors improve their software. For example, [5] focuses on `newhope1024cca` implementations and lists "failed TIMECOP" for many

[12] The Kyber team provided `ref` and `avx2` implementations to SUPERCOP. I modified `ref` as described in [34] to obtain a further implementation named `compact`.

[13] For example, in 2024, [67] reported tests discovering "a bug breaking output consistency of the reference implementation of the KNOT-384 hash function submitted to the first round of the NIST Lightweight Cryptography Standardization standardization process (independently fixed by the authors, albeit not explicitly disclosed), that would allow influencing output digests by modifying bytes neighboring (but not belonging to) the input buffer". I had already pointed this out in [31] in 2019, as one of many bugs found by SUPERCOP's tests.

computers. Per-machine links show valgrind logs for the failures, such as the following from a SUPERCOP run on a virtual machine named speed2supercop:

```
Conditional jump or move depends on uninitialised value(s)
   at 0x...: poly_frommsg (poly.c:170)
```

This virtual machine uses clang-19. The code triggering this failure, on line 170 of newhope1024cca/ref/poly.c, is the ancestor of the Kyber code highlighted in Section 4.10.

This continual scanning began with the appearance of TIMECOP 2 in SUPERCOP 20240625, although various components had appeared earlier:

- In 2010, SUPERCOP added support for automatic dependency tracking (so that each computer would skip testing what it had already tested) and for parallel tests on multiple cores.
- In 2017, building upon a fastbuild prototype from John Schanck, SUPERCOP added a do-part feature for developers to quickly test new software separately from SUPERCOP's normal database of benchmarks.
- In 2019, the original TIMECOP from Neikes [109] patched do-part to use valgrind to scan for timing variations.
- Starting in 2020, SUPERCOP incorporated various extensions of the original TIMECOP, culminating in 2024 with the release of TIMECOP 2.

TIMECOP 2 has the following advantages over the original TIMECOP:

- TIMECOP 2 automatically marks RNG output as secret.
- TIMECOP 2 reports source line numbers for valgrind failures.
- TIMECOP 2 supports crypto_declassify(&var,sizeof var) so that code can designate intermediate variables as public. Consider, for example, an RSA prime-generation loop that generates a new random integer p, tests p for primality, and starts over if p is not prime; it is safe to declassify the output of the "p is not prime" test.
- TIMECOP 2 lets callers designate public inputs to subroutines, allowing the subroutines to use variable-time code. For example, the "90s" version of Kyber applies AES to public inputs, and can safely call variable-time AES code, which is faster than constant-time AES code on many small CPUs.
- As a contribution from [44], TIMECOP 2 supports SUPERCOP's multi-core dependency-tracking database-collection tool, so the TIMECOP 2 results end up continually updated and online from many computers. The original TIMECOP had supported only do-part, which *can* be used at large scale, but TIMECOP 2 is more obviously sustainable.

Results from large-scale runs of the original TIMECOP and of TIMECOP 2 were reported in [109] and [44] respectively; but those are just one-time snapshots, while SUPERCOP's runs of TIMECOP 2 are continually updated. Developers can use do-part to test new code on their own machines, and can submit code to SUPERCOP for testing on many more machines.

6 The cryptoint API and implementation

SUPERCOP 20240807 included the first cryptoint release, and included thousands of lines that had already been converted into calls to cryptoint inside implementations of various cryptographic functions. The current version of cryptoint is version 20250414, distributed in SUPERCOP 20250415. This version is described in this section.

Section 6.1 explains the functions provided by the cryptoint API. Sections 6.2 and 6.3 explain how the implementations of these functions address the risks of timing variations—while at the same time addressing the risk of deployment being hampered by speed complaints. Section 6.4 explains how the implementations have been tested.

6.1 The API

To simplify integration, cryptoint is designed as an almost-header-only library; the reason for "almost" is the optblocker mechanism explained in Section 6.2. For example, a package using crypto_int64 includes copies of crypto_int64.h and int64_optblocker.c from cryptoint. The same comments apply to uint64, int32, uint32, int16, uint16, int8, and uint8. The sizes are independent. Compilation recommendations:

- Use gcc or clang. (Porting to other compilers should be a simple matter of compiling with -D__attribute__(x)=; however, tests have been carried out only with gcc and clang.)
- Compile all code with -fwrapv. (As noted earlier, this disables some compiler "optimizations" that often trigger bugs in integer arithmetic.)
- Compile *optblocker.c separately: i.e., don't manually merge *optblocker into other files, and don't use the -flto option in compiling *optblocker.c.

The functions supported by cryptoint for int64 and uint64 are listed in Table 6.1.1. I'm sure more functions will turn out to be useful, but I plan to continue supporting all of the functions listed in Table 6.1.1 with their current semantics. There are also functions that are not listed in Table 6.1.1 and that may change later; these functions are used internally by cryptoint.

The cryptoint API functions named *_01 are aligned with C's convention of representing true as 1 and false as 0. For example, x<y in C, like x<y?1:0, means 1 if x is smaller than y, else 0; this can be replaced with crypto_int64_smaller_01(x,y) if x and y are int64 variables, i.e., signed 64-bit integers.

The cryptoint API functions named *_mask are aligned with a different convention of representing true as -1 and false as 0. This convention is common in constant-time code (see, e.g., the comments on constant_time_select_w in Section 4.6), is supported by ARM's csetm instructions, and interacts well with logic instructions: for example, the variable-time code x<y?u:v can be rewritten as v^((u^v)&crypto_int64_smaller_mask(x,y)). The alternative

usage	meaning
`z = crypto_{int,uint}64_load(ptr)`	little-endian load
`crypto_{int,uint}64_store(ptr,z)`	little-endian store
`z = crypto_{int,uint}64_load_bigendian(ptr)`	big-endian load
`crypto_{int,uint}64_store_bigendian(ptr,z)`	big-endian store
`z = crypto_int64_positive_mask(x)`	`z = -(x > 0)`
`z = crypto_int64_positive_01(x)`	`z =  (x > 0)`
`z = crypto_int64_negative_mask(x)`	`z = -(x < 0)`
`z = crypto_int64_negative_01(x)`	`z =  (x < 0)`
`z = crypto_int64_topbit_mask(x)`	`z = -(x < 0)`
`z = crypto_int64_topbit_01(x)`	`z =  (x < 0)`
`z = crypto_uint64_topbit_mask(x)`	`z = -(x >> 63)`
`z = crypto_uint64_topbit_01(x)`	`z =  (x >> 63)`
`z = crypto_{int,uint}64_nonzero_mask(x)`	`z = -(x != 0)`
`z = crypto_{int,uint}64_nonzero_01(x)`	`z =  (x != 0)`
`z = crypto_{int,uint}64_zero_mask(x)`	`z = -(x == 0)`
`z = crypto_{int,uint}64_zero_01(x)`	`z =  (x == 0)`
`z = crypto_{int,uint}64_unequal_mask(x,y)`	`z = -(x != y)`
`z = crypto_{int,uint}64_unequal_01(x,y)`	`z =  (x != y)`
`z = crypto_{int,uint}64_equal_mask(x,y)`	`z = -(x == y)`
`z = crypto_{int,uint}64_equal_01(x,y)`	`z =  (x == y)`
`z = crypto_{int,uint}64_smaller_mask(x,y)`	`z = -(x < y)`
`z = crypto_{int,uint}64_smaller_01(x,y)`	`z =  (x < y)`
`z = crypto_{int,uint}64_leq_mask(x,y)`	`z = -(x <= y)`
`z = crypto_{int,uint}64_leq_01(x,y)`	`z =  (x <= y)`
`z = crypto_{int,uint}64_min(x,y)`	`z = (x < y) ? x : y`
`z = crypto_{int,uint}64_max(x,y)`	`z = (x > y) ? x : y`
`crypto_{int,uint}64_minmax(&x,&y)`	in-place `(x,y) = (min,max)`
`z = crypto_{int,uint}64_bottombit_mask(x)`	`z = -(x & 1)`
`z = crypto_{int,uint}64_bottombit_01(x)`	`z =  (x & 1)`
`z = crypto_{int,uint}64_shlmod(x,j)`	`z = x << (j&63)`
`z = crypto_{int,uint}64_shrmod(x,j)`	`z = x >> (j&63)`
`z = crypto_{int,uint}64_bitmod_mask(x,j)`	`z = -((x >> (j&63)) & 1)`
`z = crypto_{int,uint}64_bitmod_01(x,j)`	`z =  ((x >> (j&63)) & 1)`
`z = crypto_{int,uint}64_ones_num(x)`	`z` = bits set in `x`
`z = crypto_{int,uint}64_bottomzeros_num(x)`	`z` = low-order 0 bits in `x`

Table 6.1.1. Functions provided by `crypto_int64.h` and `crypto_uint64.h` from `cryptoint`. The `ones_num` and `bottomzeros_num` functions return, respectively, the number of bits set in `x` ("Hamming weight" or "population count" or "popcount") and the number of low-order 0 bits in `x` ("count trailing zeros"). Unlike `__builtin_ctz`, `bottomzeros_num` is guaranteed to return the number of bits in the type, here 64, if the input is 0.

`v+(u-v)*crypto_int64_smaller_01(x,y)` would rely on multiplication taking constant time, would be slower in many environments, and is not faster in any environment that I am aware of.

I plan to add support for this type of multiplexing operation, but have not finished evaluating interface options. What would be most modular is, e.g.,

```
crypto_int32_mask_then_else(crypto_int64_smaller_mask(x,y),u,v)
```

but it is tempting to support `crypto_int64_smaller_then_else(x,y,u,v)` for the common case of matching types. Either way, I think `then_else` is a better name than `select` or `mux` since `then_else` indicates the order of those two inputs.

Beware that the `mask` convention, like any other use of negative integers, is incompatible with unsigned integer extension. For example, conversion from `uint8` to `uint64` will convert `-1` to `255` rather than to `-1`. This is an argument for using signed integers rather than unsigned integers. On the other hand, the C standard allows compilers to damage the correctness of `int` code in various ways that aren't allowed for `uint`. Compiling with `-fwrapv`, as recommended above (and recommended by some other projects), has been observed to rectify this damage in various cases.

6.2 Defense 1: using a global `volatile` zero variable

The `optblocker` mechanism works as follows. First, all `int64_optblocker.c` does is define a global `volatile crypto_int64_optblocker` variable initialized to `0`:

```
#include "crypto_int64.h"
volatile crypto_int64 crypto_int64_optblocker = 0;
```

The `crypto_int64.h` file declares

```
extern volatile crypto_int64 crypto_int64_optblocker;
```

and defines all of the `crypto_int64` functions as `static inline`, so those functions are compiled in calling files separately from `int64_optblocker.c`.

Second, any operations inside the `crypto_int64` functions that produce just two possible results—i.e., few enough results to fit into a `bool`, as in Sections 4.8 and 4.10—are tweaked to insert `optblocker` in a way that would make the outputs hard to predict if `optblocker` were changed. For example:

- `crypto_int64_negative_mask` shifts right by 58 bits, then adds `crypto_int64_optblocker`, then shifts right by 5 more bits.
- `crypto_int64_bottombit_01` masks its input not with `1` but rather with `1+crypto_int64_optblocker`.
- An internal function `crypto_int64_bitinrangepublicpos_mask`, used to build `crypto_int64_shlmod` and `crypto_int64_shrmod`, xors its public shift input with `crypto_int64_optblocker` before carrying out the shift.

The choices between addition and xor are designed to eliminate any easy algebraic relationships with the surrounding operations.

Third, cryptoint has no code that changes the optblocker variable; and optblocker isn't part of the API, so correct applications won't change it. The correctness of cryptoint assumes that the value does not in fact change. But, crucially, the compiler has no way to know that the value does not change.

The gcc documentation says "Writing volatile with the type in a variable or field declaration says that the value may be examined or changed for reasons outside the control of the program at any moment"—the compiler is supposed to respect changes that the operating system makes to the variable, for example. A draft of the C23 standard [106], reportedly very close to the final (for-pay) standard, says the following: "Volatile accesses to objects are evaluated strictly according to the rules of the abstract machine."

I wouldn't rely solely on volatile—for example, I worry about code that copies data to and from a *local* volatile variable (see Section 7), in much the same way that I worry about empty inline assembly (see Section 4.6)—but the fact that optblocker is a *global* variable should disable many "optimizations" that attempt to reason about values of variables.

What happens if the application developer ignores the recommendation to separately compile int64_optblocker.c? For example, what if the application is compiled with -flto? The compiler can then see at "optimization" time that this variable is *initialized* to 0, but the compiler still has no idea whether the variable *continues* to be 0. There could be code anywhere in the program that changes the variable. Even if the compiler sees the entire program at once, it has no idea whether dynamically loaded code will change the variable.

To be clear, nothing that can be done in C code is a substitute for binary analysis. Checking the gcc and clang repositories shows that these compilers are adding new bugs all the time; maybe a compiler will somehow manage to convince itself that optblocker is always 0. Or perhaps compiler maintainers will start searching for ways to "optimize" 2-bit results into 2 bool values, forcing optblocker to be used in more contexts. But I expect that the way cryptoint currently uses optblocker will make failures much less frequent.

6.3 Defense 2: assembly

I've been adding assembly for more and more function-target pairs in cryptoint. Currently, out of the 272 functions (144 int, 128 uint) defined in the .h files (not just the API functions), there are 200 that directly invoke assembly on 64-bit Intel/AMD CPUs. There are also 200 out of 272 that directly invoke assembly for 64-bit ARM CPUs; 80 out of 272 for 32-bit ARM CPUs; and 104 out of 272 for 32-bit SPARC CPUs, which are still used in radiation-hardened space applications.

6.3.1 Reasons for assembly. I have two main motivations for using assembly. First, I see reasons to believe that assembly further reduces the risk of

triggering timing variations. Second, I think developers considering a switch to `cryptoint` will often try microbenchmarks (whether or not speed really matters for the application); assembly means that this will often produce speedups, encouraging deployment, whereas without assembly there are more likely to be slowdowns, discouraging deployment.

Consider, for example, `crypto_int64_shrmod(x,j)`, which means the same thing as `x>>(j&63)`. In case the CPU or compiler introduces timing variations depending on `j`, the portable implementation of `crypto_int64_shrmod` converts the variable-distance shift into a series of constant-distance shifts (as in typical barrel-shifter hardware), using the bottom bit of `j` to control a conditional shift by 1 bit, the next bit of `j` to control a conditional shift by 2 bits, the next bit after that to control a conditional shift by 4 bits, etc., with `optblocker` hopefully stopping the compiler from introducing any branches. But what happens if a shift of `int64` by 32 bits on a 32-bit CPU triggers branches by the mechanism described in Section 4.11?

The assembly implementations of `crypto_int64_shrmod(x,j)` are just 1 instruction on 64-bit CPUs, 7 instructions on 32-bit ARM, and 13 instructions on 32-bit SPARC: certainly faster than the portable implementation, and no need to worry about how the compiler will handle 64-bit shifts on these 32-bit CPUs, or about what might happen if an application is compiled without `-fwrapv`.

As another example, the portable implementation of `crypto_int64_max` carries out 7 arithmetic operations on top of `crypto_int64_negative_mask`. The assembly implementations of `crypto_int64_max` for 64-bit CPUs use just 2 instructions. I don't mean to suggest that switching to `cryptoint` will always be a speedup—for example, maybe a compiler figures out how to use the parallel-max instructions mentioned in Section 2.1—but assembly makes speedups more likely.

I'm using inline assembly rather than separate `.s` files. This avoids function-call overhead, and in any case is forced by the requirement of being an almost-header-only library. For applications that don't mind more files per size and don't notice the overhead, it would be easy to compile `cryptoint` into separate `.s` files.

6.3.2 Readable inline assembly. The obvious disadvantage of assembly, beyond the extra development time for each targeted CPU, is the extra risk of bugs. There are more lines of code; reviewing a typical line is more difficult, because assembly is generally harder to read than C; and the inline-assembly interface is more complicated than a simple function call, in particular in how inputs and outputs are annotated.

To improve auditability and reduce the risk of bugs, I wrote a new `readasm` tool that generates inline assembly, including annotations, from an easier-to-read format. What follows are some examples of the outputs and inputs of `readasm`.

Figure 6.3.3 displays an excerpt from `cryptoint/crypto_int64.h`, namely the C code with inline assembly for `crypto_int64_negative_mask`; recall that this function is a 63-bit right shift of an `int64`. Figure 6.3.4 displays the `readasm`

```
__attribute__((unused))
static inline
crypto_int64 crypto_int64_negative_mask(crypto_int64 crypto_int64_x) {
#if defined(__GNUC__) && defined(__x86_64__)
  __asm__ ("sarq $63,%0"
    : "+r"(crypto_int64_x) : : "cc");
  return crypto_int64_x;
#elif defined(__GNUC__) && defined(__aarch64__)
  crypto_int64 crypto_int64_y;
  __asm__ ("asr %0,%1,63"
    : "=r"(crypto_int64_y) : "r"(crypto_int64_x) : );
  return crypto_int64_y;
#elif defined(__GNUC__) && defined(__arm__) && defined(__ARM_ARCH)
    && (__ARM_ARCH >= 6) && !defined(__thumb__)
  crypto_int64 crypto_int64_y;
  __asm__ ("asr %Q0,%R1,#31\n mov %R0,%Q0"
    : "=r"(crypto_int64_y) : "r"(crypto_int64_x) : );
  return crypto_int64_y;
#elif defined(__GNUC__) && defined(__sparc_v8__)
  crypto_int64 crypto_int64_y;
  __asm__ ("sra %H1,31,%L0\n mov %L0,%H0"
    : "=r"(crypto_int64_y) : "r"(crypto_int64_x) : );
  return crypto_int64_y;
#else
  crypto_int64_x >>= 64-6;
  crypto_int64_x += crypto_int64_optblocker;
  crypto_int64_x >>= 5;
  return crypto_int64_x;
#endif
}
```

Fig. 6.3.3. `crypto_int64_negative_mask` excerpt from `cryptoint/crypto_int64.h`, except that lines indented by 4 spaces were, in the original, joined with the previous lines. See Figure 6.3.4 for source code in `readasm`.

source code in `cryptoint/functions` for `SIGNED_negative_mask`, which is used to automatically generate the C code for `crypto_int*_negative_mask`.

The instruction

```
__asm__ ("sarq $63,%0" : "+r"(crypto_int64_x) : : "cc")
```

in Figure 6.3.3 includes annotations telling the compiler that the assembly `sarq $63,%0` reads and writes `x` and modifies the CPU flags (condition codes). This instruction is automatically generated from the easier-to-read instruction

```
readasm("amd64; int64 X; X signed>>= 63")
```

in Figure 6.3.4.

Internally, the `readasm` tool has a domain-specific language to describe each CPU using a series of concise lines with colon-separated fields, similar to CPU

```
SIGNED SIGNED_negative_mask(SIGNED X) {
#if amd64
 8:  readasm("amd64; int8 X; X signed>>= 7");
16:  readasm("amd64; int16 X; X signed>>= 15");
32:  readasm("amd64; int32 X; X signed>>= 31");
64:  readasm("amd64; int64 X; X signed>>= 63");
  return X;
#elif arm64
  SIGNED Y;
 8:  readasm("arm64; int8 X Y; Y = -(1 & (X unsigned>> 7))");
16:  readasm("arm64; int16 X Y; Y = -(1 & (X unsigned>> 15))");
32:  readasm("arm64; int32 X Y; Y = X signed>> 31");
64:  readasm("arm64; int64 X Y; Y = X signed>> 63");
  return Y;
#elif arm32
  SIGNED Y;
 8:  readasm("arm32; int8 X Y; Y = (int8) X; Y = Y signed>> 31");
16:  readasm("arm32; int16 X Y; Y = (int16) X; Y = Y signed>> 31");
32:  readasm("arm32; int32 X Y; Y = X signed>> 31");
64:  readasm("arm32; int64 X Y; Y.lo = X.hi signed>> 31; Y.hi = Y.lo");
  return Y;
#elif sparc32
  SIGNED Y;
 8:  readasm("sparc32; int8 X Y; Y = X << 24; Y = Y signed>> 31");
16:  readasm("sparc32; int16 X Y; Y = X << 16; Y = Y signed>> 31");
32:  readasm("sparc32; int32 X Y; Y = X signed>> 31");
64:  readasm("sparc32; int64 X Y; Y.lo = X.hi signed>> 31; Y.hi = Y.lo");
  return Y;
#else
  X >>= N-6;
  X += SIGNED_optblocker;
  X >>= 5;
  return X;
#endif
}
```

Fig. 6.3.4. `SIGNED_negative_mask` excerpt from `cryptoint/functions`. See Figure 6.3.3 for automatically generated C code.

descriptions in my existing `qhasm` tool for generating `.s` files. For example, the line

```
r signed>>= n:+r=int64:#n:asm/sarq $#n,+r:>?cc:
```

for `amd64` creates syntax `r signed>>= n` for an instruction that modifies `r`, an `int64` register, in place; reads a constant `n`; is written as `sarq $n,r` in assembly; and modifies the CPU flags.

This might look like simply a rephrasing of the `sarq` line quoted above from Figure 6.3.3. The advantage is that the CPU-description lines are reused. For

example, currently there are five uses of sarq in cryptoint; two are generated from the CPU-description line above, and three are generated from

```
r signed>>= s:+r=int64:<s=int64#4:asm/sarq %%cl,+r:>?cc:
```

for variable-distance shifts. This factorization means that, instead of five opportunities to forget cc for sarq (one in each inline-assembly snippet using sarq), there are just two (one in each of the CPU-description lines using sarq).

This does not mean that the risk of bugs has disappeared. On the contrary, there could be bugs in the CPU descriptions, or in other aspects of the readasm tool. Perhaps it would be better for readasm to *always* generate cc, for example; the tricky question here is whether the risk of an omitted cc outweighs the microbenchmark-based deployment risk described above. The risk of bugs in CPU descriptions can be addressed via automatic cross-checks against other sources of CPU information (or via auto-generation of CPU descriptions), but this has not been done yet.

6.4 Tests

The fact that there are 272 different functions in cryptoint poses a clear risk of bugs. The use of assembly amplifies this risk. The steps that I have taken to address this risk, beyond trying to get the code right in the first place, include conventional tests (see Section 6.4.1), equivalence checking via symbolic execution (see Section 6.4.2), and caller tests (see Section 6.4.3).

Recall from Section 1.1 that I recommend using cryptoint. This isn't saying that the risks have been eliminated; it means that I see higher risks in code that *isn't* using cryptoint. Common practice (see, e.g., [125] or the examples in Section 7) involves less comprehensive testing of such code snippets, and of analogous assembly generated by compilers.

6.4.1 Conventional unit tests. Each SUPERCOP run includes a battery of conventional unit tests for cryptoint, as specified by cryptoint/test.c, which is compiled and run with command-line arguments 1 0. The inputs x and y for these tests are generated in three ways:

- All pairs of small integers. Small means between -100 and 100 for int, or between 0 and 200 for uint.
- All pairs of integers of the form $k \pm 2^i$, where $-3 \leq k \leq 3$ and $0 \leq i < N$. Here N is the number of bits in the type.
- A pseudorandom sequence of 10000 pairs of x and y. This sequence is deterministic for reproducibility.

For shift functions such as shlmod, all shift distances between $-10N$ and $10N$ are tested.

Overall the tests try 239962288 calls to the cryptoint functions, not counting calls to load, store, and minmax. These calls have 219612398 unique inputs. Most of the collisions are for int8 and uint8.

6.4.2 Equivalence checking via symbolic execution. I've used symbolic execution and SMT solving to check that `cryptoint`, compiled using various compilers for `amd64`, `arm64`, `arm32`, `mips64`, `sparc32`, and `x86`, produces, for *all* possible inputs, the same results as reference implementations of the same functions. This addresses the risk of bugs appearing for inputs that aren't tested by `test.c`.

This equivalence checking uses the current version (available from [43]) of `saferewrite`. Most of the underlying work is handled by the `angr` tool introduced in [127], available from [6]. That tool doesn't support `sparc32`; I worked around this by symbolically executing a SPARC emulator. See [42] for details.

6.4.3 Caller tests. Each SUPERCOP run includes unit tests for a wide range of cryptographic implementations. Many of those cryptographic implementations are now using `cryptoint`, so these tests include tests for a variety of `cryptoint` calls in context.

Testing a variety of callers could catch, e.g., an assembly-annotation error that shows up only for some types of callers and that isn't triggered by the direct unit tests for `cryptoint`. I scanned SUPERCOP results on various machines before and after switching code to using `cryptoint` (and, earlier, `inttypes`), and didn't detect any changes of outputs.

I also periodically investigate SUPERCOP test failures, and if a new compiler triggers `cryptoint` bugs then I would expect to see this as a variation across machines—although there is continual background noise here from, e.g., implementations that haven't been ported to big-endian platforms or that don't even compile with a new compiler. It might be useful to add a tool that, for each test failure, automatically checks whether substituting different implementations of the `cryptoint` functions resolves the failure.

7 Other libraries

Code designed to be constant-time often factors out subroutines similar to, e.g., `crypto_int64_nonzero_mask` from `cryptoint`. As concrete examples, Sections 7.1, 7.2, 7.3, and 7.4 look at subroutines provided in OpenSSL, BoringSSL, BearSSL, and Botan respectively. There have been many changes in these subroutines over the years; this section focuses on the current versions at the time of this writing, the April 2025 development versions.

Natural questions in each case include which subroutines are supported; how broadly the subroutines are used; how well tested the subroutines are; how well optimized the subroutines are; and what protections the subroutines have against compilers introducing timing variations.

There are large overlaps in the APIs. It would be easy to replace various existing subroutine implementations with calls to `cryptoint`, generally improving the levels of testing, optimization, and protection against timing

attacks. Furthermore, supporting the uniform cryptoint interface should help encourage broader usage of constant-time subroutines.

7.1 OpenSSL

OpenSSL's include/internal/constant_time.h provides constant_time_* functions: msb, lt, and is_zero for unsigned int, uint32_t, uint64_t, size_t, and BN_ULONG inputs, along with eq and ge functions for a narrower range of types, in each case returning a mask; select and cond_swap functions using masks; and a higher-level lookup function for array lookups. See [111].

There are 277 lines in OpenSSL mentioning constant_time_, including 91 in constant_time.h and 50 in the test directory. The tests try, e.g.,

```
static uint64_t test_values_64[] = {
    0, 1, 1024, 12345, 32000, 32000000, 32000000001,
    UINT64_MAX / 2, UINT64_MAX / 2 + 1,
    UINT64_MAX - 1, UINT64_MAX
};
```

as inputs to the 64-bit functions.

Internally, select passes its input masks through a value_barrier function designed to avoid optimizations, based on the BoringSSL patch covered in Section 4.6. The value_barrier details are slightly different from [15]: the empty inline assembly uses two C variables, and there is a fallback to a volatile store and load for compilers not supporting inline assembly.

As in [128] and [15], the OpenSSL barriers focus on protecting selection rather than protecting comparisons. The mask computations and cond_swap in [111] are not defended against compilers recognizing that there are only two possible mask values.

As an example of potentially dangerous code in OpenSSL not using any of these functions, consider the line

```
copy_conditional(temp[0].Y, temp[1].Y, (wvalue & 1));
```

in [112], OpenSSL's crypto/ec/ecp_nistz256.c. The copy_conditional function is a static function defined in the same file, with no optimization barriers.

7.2 BoringSSL

BoringSSL's crypto/internal.h provides the same basic constant_time_* functions (msb, lt, ge, is_zero, and eq) on crypto_word_t inputs; select; and higher-level conditional_memcpy and conditional_memxor. See [52].

There are 269 lines in BoringSSL mentioning constant_time_, including 49 in internal.h and 20 in crypto/constant_time_test.cc. The tests are similar to OpenSSL, plus random inputs to conditional_memcpy etc. There are also comments for lt and is_zero showing manual verification via an SMT solver.

Internally, BoringSSL uses barriers in `select`, essentially as in [15]. As for speed, BoringSSL has some vectorized code in `conditional_memxor`.

There is a BoringSSL option to run some tests under `valgrind`. These tests are triggered by `CONSTTIME_SECRET`, which is used in BoringSSL for Curve25519 and a few other cryptographic computations.

As an example of potentially dangerous code in BoringSSL avoiding these functions, BoringSSL's `crypto/fipsmodule/ec/p256-nistz.cc.inc` [53] includes a `copy_conditional` line similar to the OpenSSL example above.

7.3 BearSSL

BearSSL's `src/inner.h` [121] provides `NOT`, `MUX`, `EQ`, `NEQ`, `GT`, `LT`, `GE`, `LE`, `EQ0`, `GT0`, `GE0`, `LT0`, `LE0`, `MIN`, `MAX`, and `CMP` on 32-bit integers, using the 0-1 convention for false-true values and a three-value convention for `CMP`. It also provides `BIT_LENGTH` as a variant of counting leading zeros; `br_ccopy` for a conditional `memcpy`; `br_divrem` for constant-time divisions; and some multiplication functions that work around sources of timing variations in multiplications on some CPUs.

Because these names are so short, easy ways to count callers produce many false positives (e.g., `MAX` is overridden by a variable-time `MAX` macro in `src/rsa/rsa_i15_keygen.c`), but I think there are more than 200 calls to these functions in BearSSL.

Tests in BearSSL seem to focus on higher-level functions, meaning that the `inner.h` functions are tested only via caller tests. I did not find any support in BearSSL for barriers or for binary analysis.

As an example of potentially dangerous code in BearSSL not using these functions, `src/hash/ghash_ctmul.c` [118] has 31-bit right shifts of `uint32` values.

7.4 Botan

Botan's `src/lib/utils/bit_ops.h` [99] includes various `ct_` functions, such as `ct_is_zero<T>` converting values of unsigned[14] type `T` into masks, and higher-level functions such as `ct_reverse_bits<T>` and `ct_popcount<T>`. Botan has other files providing further `ct_` functions, such as `ct_divide`. There are also various further subroutines that are not `ct_*` but seem intended to be constant-time, such as `ctz<T>` for counting trailing zeros (there is a separate `var_ctz32` function documented as potentially variable time) and `expand_top_bit<T>`.

Botan's `src/lib/utils/ct_utils.h` [101] provides various `CT::` functions supporting `is_zero`, `is_equal`, `is_lt`, `is_gt`, `expand_top_bit`, `expand_bit` for a bit at any position, `conditional_swap`, `is_any_of` with a list of inputs, `select_n` handling input arrays, `all_zeros`, and more.

[14] This function enforces only `is_integral`, but seems to be called only for `uint` types. For signed types, the implementation would not match the documentation.

There are 759 lines in Botan mentioning CT::, including 45 and 28 in src/tests/test_ct_utils.cpp and src/ct_selftest/ct_selftest.cpp respectively. There are also some further uses of the ct_ functions.

The test_ct_utils tests include 20 inputs for is_zero and is_lt, a wider range of inputs for an array operation CT::copy_output that indirectly tests other functions, and miscellaneous checks on other CT features.

Botan has an option for binary analysis with valgrind, triggered by CT::poison calls; there are 101 of these calls. The ct_selftest tests check whether CT::poison works correctly.

Internally, barriers seem to be used on more data paths in these Botan subroutines than in corresponding subroutines in OpenSSL or BoringSSL,[15] but there still seem to be many paths around the barriers. For example, ct_is_zero does not have barriers.

As an example of potentially dangerous code in Botan not using these functions, math/mp/mp_asmi.h [100] includes a word_add function that uses carry & 1 rather than using expand_bit.

References

[1] — (no editor), *IEEE symposium on security and privacy, SP 2016, San Jose, CA, USA, May 22–26, 2016*, IEEE Computer Society, 2016. ISBN 978-1-5090-0824-7. URL: https://ieeexplore.ieee.org/xpl/conhome/7528194/proceeding. See [127].

[2] — (no editor), *IEEE cybersecurity development, SecDev 2017, Cambridge, MA, USA, September 24–26, 2017*, IEEE Computer Society, 2017. ISBN 978-1-5386-3467-7. URL: https://ieeexplore.ieee.org/xpl/conhome/8071083/proceeding. See [58].

[3] — (no editor), *2018 IEEE European symposium on security and privacy, EuroS&P 2018, London, United Kingdom, April 24–26, 2018*, IEEE, 2018. ISBN 978-1-5386-4228-3. URL: https://ieeexplore.ieee.org/xpl/conhome/8405665/proceeding. See [128].

[4] — (no editor), *2020 IEEE symposium on security and privacy, SP 2020, San Francisco, CA, USA, May 18–21, 2020*, IEEE, 2020. URL: https://ieeexplore.ieee.org/xpl/conhome/9144328/proceeding. See [62].

[5] — (no editor), *Implementation comparison: crypto_kem/newhope1024cca* (2025), accessed 2025-04-24. URL: https://bench.cr.yp.to/impl-kem/newhope1024cca.html. Citations in this document: §5.

[6] — (no editor), *angr* (2025), accessed 2025-04-24. URL: https://angr.io. Citations in this document: §6.4.2.

[7] Onur Aciiçmez, Billy Bob Brumley, Philipp Grabher, *New results on instruction cache attacks*, in CHES 2010 [104] (2010), 110–124. Citations in this document: §3.

[8] Advanced Micro Devices, *Software optimization guide for AMD Athlon 64 and AMD Opteron processors*, 2004. URL: https://cr.yp.to/bib/2004/-amd-25112.pdf. Citations in this document: §3.1.

[15] To avoid "significant performance regressions", Botan's barriers are disabled "with MSVC, which is not known to perform optimizations that break constant time code", although there is an option to copy data to and from a volatile variable.

[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, Tiago Oliveira, *Certified compilation for cryptography: extended x86 instructions and constant-time verification*, in Indocrypt 2020 [**50**] (2020), 107–127. DOI: 10.1007/978-3-030-65277-7_6. Citations in this document: §1.

[10] Arm, *DIT, Data Independent Timing* (2020). URL: https://developer.arm.com/documentation/ddi0601/2020-12/AArch64-Registers/DIT--Data-Independent-Timing. Citations in this document: §1.3.

[11] Jean-Philippe Aumasson, *Coding rules* (2013). URL: https://web.archive.org/web/20130709013142/https://cryptocoding.net/index.php/Coding_rules. Citations in this document: §4.2.

[12] Jean-Philippe Aumasson, *crypto coding (bis)* (2014). URL: https://www.aumasson.jp/data/talks/cryptocoding_irmar14.pdf. Citations in this document: §4.2.

[13] Davide Balzarotti, Wenyuan Xu (editors), *33rd USENIX security symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14–16, 2024*, USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24. See [59], [71].

[14] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David Wu, Yuval Yarom, *Testing side-channel security of cryptographic implementations against future microarchitectures*, in CCS 2024 [**102**] (2024), 1076–1090. URL: https://arxiv.org/abs/2402.00641. DOI: 10.1145/3658644.3670319. Citations in this document: §3.

[15] David Benjamin, *Add a value barrier to constant-time selects* (2019). URL: https://boringssl.googlesource.com/boringssl/+/92b7c89e6e8ba82924b57153bea68241cc45f658%5E%21/. Citations in this document: §4.6, §4.6, §4.6, §4.6, §4.6, §4.8, §7.1, §7.1, §7.2.

[16] Daniel J. Bernstein, *Cache-timing attacks on AES* (2005). URL: https://cr.yp.to/papers.html#cachetiming. Citations in this document: §1, §1.3, §1.3, §2, §3.1, §3.1, §3.1.

[17] Daniel J. Bernstein, *Salsa20 design* (2005). URL: https://cr.yp.to/snuffle/design.pdf. Citations in this document: §2, §3.2, §3.2.

[18] Daniel J. Bernstein, *Salsa20 speed* (2005). URL: https://cr.yp.to/snuffle/speed.pdf. Citations in this document: §2.

[19] Daniel J. Bernstein, *High-speed elliptic-curve cryptography* (2005). URL: https://cr.yp.to/talks.html#2005.05.19. Citations in this document: §2.

[20] Daniel J. Bernstein, *Cache-timing attacks on AES* (2005). URL: https://cr.yp.to/talks.html#2005.06.01. Citations in this document: §2.

[21] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [**135**] (2006), 207–228. URL: https://cr.yp.to/papers.html#curve25519. Citations in this document: §2, §2, §2, §3, §3, §3.2, §4.2.

[22] Daniel J. Bernstein, *eBATS: ECRYPT Benchmarking of Asymmetric Systems* (2006). URL: https://cr.yp.to/talks.html#2006.05.30. Citations in this document: §5.

[23] Daniel J. Bernstein, *The impact of side-channel attacks on the design of cryptosystems* (2007). URL: https://cr.yp.to/talks.html#2007.05.28. Citations in this document: §3.

[24] Daniel J. Bernstein, *The Salsa20 family of stream ciphers*, in [**124**] (2008), 84–97. URL: https://cr.yp.to/papers.html#salsafamily. Citations in this document: §2, §3, §4.2.

[25] Daniel J. Bernstein, *eBASH: ECRYPT Benchmarking of All Submitted Hashes* (2008). URL: https://cr.yp.to/talks.html#2008.12.09. Citations in this document: §5.

[26] Daniel J. Bernstein, *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2009). URL: https://cr.yp.to/talks.html#2009.01.12. Citations in this document: §5.

[27] Daniel J. Bernstein, *eBASH: ECRYPT Benchmarking of All Submitted Hashes* (2009). URL: https://cr.yp.to/talks.html#2009.02.28-1. Citations in this document: §5.

[28] Daniel J. Bernstein, *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2009). URL: https://cr.yp.to/talks.html#2009.09.08-1. Citations in this document: §5.

[29] Daniel J. Bernstein, *Software benchmarking* (2009). URL: https://cr.yp.to/talks.html#2009.11.17. Citations in this document: §5.

[30] Daniel J. Bernstein, *djbsort* (2018). URL: https://sorting.cr.yp.to. Citations in this document: §2.1, §2.1.

[31] Daniel J. Bernstein, *LWC in SUPERCOP* (2019). URL: https://groups.google.com/a/list.nist.gov/g/lwc-forum/c/SblnAnaKNy0/m/L4XzMdPpEAAJ. Citations in this document: §5.

[32] Daniel J. Bernstein, *Fast verified post-quantum software* (2021). URL: https://cr.yp.to/talks/2021.09.03/slides-djb-20210903-saferewrite-4x3.pdf. Citations in this document: §3.2.

[33] Daniel J. Bernstein, *Cryptographic competitions*, Journal of Cryptology **37** (2024), article 7. URL: https://cr.yp.to/papers.html#competitions. Citations in this document: §5.

[34] Daniel J. Bernstein, *Analyzing the complexity of reference post-quantum software: the case of lattice-based KEMs* (2024). URL: https://cr.yp.to/papers.html#pqcomplexity. Citations in this document: §5.

[35] Daniel J. Bernstein, *Tracking down some TIMECOP alerts led to a 2021 gcc patch from ARM (https://gcc.gnu.org/git/?p=gcc.git;a=commit;f=gcc/match.pd;h=d70720c2382e687e192a9d666e80acb41bfda856) turning `(-x)>>31` into a bool, often breaking constant-time code. Can often work around with `(-x)>>30`, and asm is safer anyway, but for portable fallbacks we need security-aware compilers* (2024). URL: https://microblog.cr.yp.to/1713627640/. Citations in this document: §4.8.

[36] Daniel J. Bernstein, *supercop-20240425 available* (2024), email to ebats@list.cr.yp.to; available via ebats-help@list.cr.yp.to as message 1138. Citations in this document: §4.9, §4.9.

[37] Daniel J. Bernstein, *inttypes 20240513* (2024). URL: https://lib.mceliece.org/libmceliece-20240513/inttypes.html. Citations in this document: §4.9.

[38] Daniel J. Bernstein, *Clang vs. Clang* (2024). URL: https://blog.cr.yp.to/20240803-clang.html. Citations in this document: §4.11, §4.11.

[39] Daniel J. Bernstein (editor), *lib25519-20241004* (2024), accessed 2025-04-24. URL: https://lib25519.cr.yp.to. Citations in this document: §1.

[40] Daniel J. Bernstein (editor), *libmceliece-20241009* (2024), accessed 2025-04-24. URL: https://lib.mceliece.org. Citations in this document: §1, §3.2.

[41] Daniel J. Bernstein (editor), *libntruprime-20241021* (2024), accessed 2025-04-24. URL: https://libntruprime.cr.yp.to. Citations in this document: §1.

[42] Daniel J. Bernstein, *Symbolically executing emulators* (2025), in preparation. Citations in this document: §6.4.2.

[43] Daniel J. Bernstein, *Post-Quantum Software Research Center: Downloads* (2025), accessed 2025-04-24. URL: https://pqsrc.cr.yp.to/downloads.html. Citations in this document: §6.4.2.

[44] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, Goutam Tamvada, *KyberSlash: Exploiting secret-dependent division timings in Kyber implementations* (2025), IACR Transactions on Cryptographic Hardware and Embedded Systems, to appear. URL: https://cr.yp.to/papers.html#kyberslash. Citations in this document: §1, §4.2, §5, §5.

[45] Daniel J. Bernstein, Billy Bob Brumley, *Timing attacks* (2022). URL: https://timing.attacks.cr.yp.to. Citations in this document: §1.

[46] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2025), accessed 2025-04-24. URL: https://bench.cr.yp.to. Citations in this document: §1, §5.

[47] Daniel J. Bernstein, Tanja Lange, Peter Schwabe, *Internals* (2011). URL: https://nacl.cr.yp.to/internals.html. Citations in this document: §4.2, §4.2, §4.4, §4.7, §4.12, §4.12, §4.12, §4.12.

[48] Daniel J. Bernstein, Tanja Lange, Peter Schwabe, *The security impact of a new cryptographic library*, in Latincrypt 2012 [**82**] (2012), 159–176. URL: https://cr.yp.to/papers.html#coolnacl. DOI: 10.1007/978-3-642-33481-8_9. Citations in this document: §4.2.

[49] Daniel J. Bernstein, Peter Schwabe, *A word of warning* (2013). URL: https://cryptojedi.org/peter/data/chesrump-20130822.pdf. Citations in this document: §3.1.

[50] Karthikeyan Bhargavan, Elisabeth Oswald, Manoj Prabhakaran (editors), *Progress in cryptology—INDOCRYPT 2020—21st international conference on cryptology in India, Bangalore, India, December 13–16, 2020, proceedings*, 12578, Springer, 2020. ISBN 978-3-030-65276-0. DOI: 10.1007/978-3-030-65277-7. See [9].

[51] Dávid Bolvanský, *[SDAG] Preserve unpredictable metadata, teach X86CmovConversion to respect this metadata* (2022). URL: https://reviews.llvm.org/D118118. Citations in this document: §4.5.

[52] BoringSSL, *internal.h* (2025), accessed 2025-04-24. URL: https://github.com/google/boringssl/blob/main/crypto/internal.h. Citations in this document: §7.2.

[53] BoringSSL, *p256-nistz.cc.inc* (2025), accessed 2025-04-24. URL: https://github.com/google/boringssl/blob/main/crypto/fipsmodule/ec/p256-nistz.cc.inc. Citations in this document: §7.2.

[54] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, Cristiano Giuffrida, *Constantine: automatic side-channel resistance using efficient control and data flow linearization*, in CCS 2021 [**91**] (2021), 715–733. DOI: 10.1145/3460120.3484583. Citations in this document: §1.

[55] Ernie Brickell, *Technologies to improve platform security* (2011). URL: https://www.iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf. Citations in this document: §3.1.

[56] Joseph A. Calandrino, Carmela Troncoso (editors), *32nd USENIX security symposium, USENIX Security 2023, Anaheim, CA, USA, August 9–11, 2023*, USENIX Association, 2023. URL: https://www.usenix.org/conference/usenixsecurity23. See [132].

[57] Larry Campbell, *TENEX hackery* (1991). URL: https://groups.google.com/g/alt.folklore.computers/c/v9KnB8BIXGY/m/aZ-qDLtD0gAJ. Citations in this document: §1, §2.

[58] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, Deian Stefan, *FaCT: A flexible, constant-time programming language*, in SecDev 2017 [**2**] (2017), 69–76. DOI: 10.1109/SecDev.2017.24. Citations in this document: §1.

[59] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, Daniel Genkin, *GoFetch: breaking constant-time cryptographic implementations using data memory-dependent prefetchers*, in USENIX Security 2024 [**13**] (2024). URL: https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru. Citations in this document: §1.3, §3.4, §3.4, §3.4, §3.4.

[60] Tamar Christina, *middle-end: convert negate + right shift into compare greater* (2021). URL: https://gcc.gnu.org/git/?p=gcc.git;a=commit;f=gcc/match.pd;h=d70720c2382e687e192a9d666e80acb41bfda856. Citations in this document: §4.8, §4.8, §4.8, §4.8, §4.8.

[61] Lesly-Ann Daniel, *lib.c* (2022). URL: https://github.com/binsec/rel_bench/blob/main/src/ct-sort/lib.c. Citations in this document: §4.7.

[62] Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk, *Binsec/Rel: efficient relational symbolic execution for constant-time at binary-level*, in S&P 2020 [**4**] (2020), 1021–1038; see also newer version [**63**]. URL: https://arxiv.org/abs/1912.08788. DOI: 10.1109/SP40000.2020.00074. Citations in this document: §4.7, §4.7, §4.7.

[63] Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk, *Binsec/Rel: symbolic binary analyzer for security with applications to constant-time and secret-erasure*, ACM Transactions on Privacy and Security **26** (2023), 11:1–11:42; see also older version [**62**]. URL: https://arxiv.org/abs/2209.01129. DOI: 10.1145/3563037. Citations in this document: §4.7.

[64] Vincent Dankbaar, *The disappearance of constant-time execution. A C compiler's magic trick* (2024). URL: https://www.cs.ru.nl/bachelors-theses/2024/Vincent_Dankbaar___s1031350___The_disappearance_of_constant-time_execution.pdf. Citations in this document: §4.10, §4.10.

[65] Travis Downs, *Hardware store elimination* (2020). URL: https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html. Citations in this document: §3.4.

[66] Elena Dubrova, Kalle Ngo, Joel Gärtner, Ruize Wang, *Breaking a fifth-order masked implementation of CRYSTALS-KYBER by copy-paste*, in APKC 2023 [**73**] (2023), 10–20. DOI: 10.1145/3591866.3593072. Citations in this document: §2.

[67] Giacomo Fenzi, Jan Gilcher, Fernando Virdia, *Finding bugs and features using cryptographically-informed functional testing* (2024). URL: https://eprint.iacr.org/2024/1122. Citations in this document: §5.

[68] Décio Luiz Gazzoni Filho, Tomás S. R. Silva, Julio López, *Efficient isochronous fixed-weight sampling with applications to NTRU*, IACR Communications in Cryptology **1.2** (2024), article 14. URL: https://cic.iacr.org/p/1/2/14/pdf. DOI: 10.62056/a6n59qgxq. Citations in this document: §3.4.

[69] Agner Fog, *The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers* (2024). URL: https://agner.org/optimize/microarchitecture.pdf. Citations in this document: §3, §3.1.

[70] Sara Foresti, Giuseppe Persiano (editors), *Cryptology and network security—15th international conference, CANS 2016, Milan, Italy, November 14–16, 2016, proceedings*, 10052, 2016. ISBN 978-3-319-48964-3. DOI: 10.1007/978-3-319-48965-0. See [89].

[71] Marcel Fourné, Daniel De Almeida Braga, Jan Jancar, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, Yasemin Acar, *"These results must be false": A usability evaluation of constant-time analysis tools*, in USENIX Security 2024 [**13**] (2024). URL: https://www.usenix.org/conference/usenixsecurity24/presentation/fourne. Citations in this document: §1.1.

[72] Free Software Foundation, *Extended asm - assembler instructions with C expression operands* (2025), accessed 2025-04-24. URL: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html. Citations in this document: §4.6.

[73] Masayuki Fukumitsu, Shingo Hasegawa (editors), *Proceedings of the 10th ACM Asia public-key cryptography workshop, APKC 2023, Melbourne, VIC, Australia, July 10–14, 2023*, ACM, 2023. DOI: 10.1145/3591866. See [66].

[74] Benedikt Gierlichs, Axel Y. Poschmann (editors), *Cryptographic hardware and embedded systems—CHES 2016—18th international conference, Santa Barbara, CA, USA, August 17–19, 2016, proceedings*, Lecture Notes in Computer Science, 9813, Springer, 2016. ISBN 978-3-662-53139-6. See [134].

[75] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, Martin Dixon, *Fast and constant-time implementation of modular exponentiation* (2009). URL: https://cse.buffalo.edu/srds2009/escs2009_submission_Gopal.pdf. Citations in this document: §3.1.

[76] Wouter de Groot, *A performance study of X25519 on Cortex-M3 and M4* (2015). URL: https://research.tue.nl/en/studentTheses/a-performance-study-of-x25519-on-cortex-m3-and-m4. Citations in this document: §3.2.

[77] Shay Gueron, *Intel Advanced Encryption Standard (AES) New Instructions Set* (2010). URL: https://web.archive.org/web/20141011221834/https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf. Citations in this document: §1.3.

[78] Shay Gueron, *Efficient software implementations of modular exponentiation*, Journal of Cryptographic Engineering **2** (2012), 31–43. URL: https://eprint.iacr.org/2011/239. Citations in this document: §3.1.

[79] Qian Guo, Thomas Johansson, Alexander Nilsson, *A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM*, in Crypto 2020 [**107**] (2020), 359–386. DOI: 10.1007/978-3-030-56880-1_13. Citations in this document: §1.

[80] Dave Hansen, *Subject: Re: [PATCH] x86: enable Data Operand Independent Timing Mode* (2023). URL: https://lore.kernel.org/lkml/851920c5-31c9-ddd9-3e2d-57d379aa0671@intel.com/. Citations in this document: §3.4, §3.5.

[81] John Harrison, *The x25519 function for curve25519* (2023). URL: https://github.com/awslabs/s2n-bignum/blob/main/x86/proofs/curve25519_x25519.ml. Citations in this document: §2, §3.

[82] Alejandro Hevia, Gregory Neven (editors), *Progress in cryptology—LATINCRYPT 2012—2nd international conference on cryptology and information security in Latin America, Santiago, Chile, October 7–10, 2012. proceedings*, 7533, Springer, 2012. ISBN 978-3-642-33480-1. DOI: 10.1007/978-3-642-33481-8. See [48].

[83] Jeff Hurchalla, *Bug 98801 - Request for a conditional move built-in function* (2021). URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=98801. Citations in this document: §4.5.

[84] Intel, *8086/8088 user's manual: programmer's and hardware reference*, 1989. URL: https://archive.org/details/80868088usersman0000unse. Citations in this document: §3.3.

[85] Intel, *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance* (2022). URL: https://archive.cr.yp.to/2022-09-24/09:44:12/iX88hHTkrNsX18tbTWfGuhwVv_E0DBpgyfvuupQzCBw/https/www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html. Citations in this document: §1.3.

[86] Intel Corporation, *IA-32 Intel Architecture optimization: reference manual*, 2004. URL: https://cr.yp.to/bib/2004/-intel-optimization.pdf. Citations in this document: §3.1.

[87] Jan Jancar, *The state of tooling for verifying constant-timeness of cryptographic implementations* (2021). URL: https://neuromancer.sk/article/26. Citations in this document: §1.1.

[88] Arseny Kapoulkine, *On Proebsting's Law* (2022). URL: https://zeux.io/2022/01/08/on-proebstings-law/. Citations in this document: §1.2.

[89] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, Karine Villegas, *When constant-time source yields variable-time binary: exploiting Curve25519-donna built with MSVC 2015*, in CANS 2016 [**70**] (2015), 573–582. URL: https://infoscience.epfl.ch/server/api/core/bitstreams/9e05464d-6192-4b77-9441-286c8679a976/content. DOI: 10.1007/978-3-319-48965-0_36. Citations in this document: §4.3, §4.3, §4.4.

[90] Jason Kim, Jalen Chuang, Daniel Genkin, Yuval Yarom, *FLOP: Breaking the Apple M3 CPU via False Load Output Predictions* (2025). URL: https://yuval.yarom.org/pdfs/KimCGY25.pdf. Citations in this document: §1.3, §3.4, §3.4, §3.4.

[91] Yongdae Kim, Jong Kim, Giovanni Vigna, Elaine Shi (editors), *CCS '21: 2021 ACM SIGSAC conference on computer and communications security, virtual event, Republic of Korea, November 15–19, 2021*, ACM, 2021. ISBN 978-1-4503-8454-4. DOI: 10.1145/3460120. See [54].

[92] Engin Kirda, Thomas Ristenpart (editors), *26th USENIX security symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16–18, 2017*, USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17. See [95].

[93] Neal Koblitz (editor), *Advances in cryptology—CRYPTO '96, 16th annual international cryptology conference, Santa Barbara, California, USA, August 18–22, 1996, proceedings*, 1109, Springer, 1996. ISBN 3-540-61512-1. DOI: 10.1007/3-540-68697-5. See [94].

[94] Paul C. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, in Crypto 1996 [**93**] (1996), 104–113. DOI: 10.1007/3-540-68697-5_9. Citations in this document: §1, §2, §2, §2, §4.9.

[95] David Kohlbrenner, Hovav Shacham, *On the effectiveness of mitigations against floating-point timing channels*, in USENIX Security 2017 [**92**] (2017), 69–81. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/kohlbrenner. Citations in this document: §3.2.

[96] Korean Post-Quantum Cryptography, *Selected algorithms from the KpqC competition round 2* (2025). URL: https://kpqc.or.kr/competition_02.html. Citations in this document: §4.8.

[97] Georg Land, Adrian Marotzke, Jan Richter-Brockmann, Tim Güneysu, *Gadget-based masking of streamlined NTRU Prime decapsulation in hardware*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2024** (2024), 1–26. DOI: 10.46586/tches.v2024.i1.1-26. Citations in this document: §2.

[98] Adam Langley, *Checking that functions are constant time with Valgrind* (2010). URL: https://github.com/agl/ctgrind. Citations in this document: §4.3.

[99] Jack Lloyd, Bhaskar Biswas, Nicolas Sendrier, Falko Strenzke, *bit_ops.h* (2025), accessed 2025-04-24. URL: https://github.com/randombit/botan/blob/master/src/lib/utils/bit_ops.h. Citations in this document: §7.4.

[100] Jack Lloyd, Luca Piccarreta. URL: https://github.com/randombit/botan/blob/master/src/lib/math/mp/mp_asmi.h. Citations in this document: §7.4.

[101] Jack Lloyd, Falko Strenzke, *ct_utils.h* (2024), accessed 2025-04-24. URL: https://github.com/randombit/botan/blob/master/src/lib/utils/ct_utils.h. Citations in this document: §7.4.

[102] Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, David Lie (editors), *Proceedings of the 2024 on ACM SIGSAC conference on computer and communications security, CCS 2024, Salt Lake City, UT, USA, October 14–18, 2024*, ACM, 2024. ISBN 979-8-4007-0636-3. DOI: 10.1145/3658644. See [14].

[103] Stefan Mangard, Elisabeth Oswald, Thomas Popp, *Power analysis attacks—revealing the secrets of smart cards* (2007). ISBN 978-0-387-30857-9. Citations in this document: §2.

[104] Stefan Mangard, François-Xavier Standaert (editors), *Cryptographic hardware and embedded systems, CHES 2010, 12th international workshop, Santa Barbara, CA, USA, August 17–20, 2010, proceedings*, Lecture Notes in Computer Science, 6225, Springer, 2010. ISBN 978-3-642-15030-2. See [7].

[105] Hector Martin, *Found the DMP disable chicken bit* (2024). URL: https://web.archive.org/web/20240410142137/https://social.treehouse.systems/@marcan/112238385679496096. Citations in this document: §3.4.

[106] JeanHeyd Meneide, Freek Wiedijk (editors), *Information technology — Programming languages — C*, 2024. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf. Citations in this document: §6.2.

[107] Daniele Micciancio, Thomas Ristenpart (editors), *Advances in cryptology—CRYPTO 2020—40th annual international cryptology conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, proceedings, part II*, 12171, Springer, 2020. ISBN 978-3-030-56879-5. DOI: 10.1007/978-3-030-56880-1. See [79].

[108] David Molnar, Matt Piotrowski, David Schultz, David A. Wagner, *The program counter security model: automatic detection and removal of control-flow side channel attacks*, in ICISC 2005 [**130**] (2005), 156–168. DOI: 10.1007/11734727_14. Citations in this document: §1.

[109] Moritz Neikes, *TIMECOP: automated dynamic analysis for timing side-channels* (2019). URL: https://www.post-apocalyptic-crypto.org/timecop/. Citations in this document: §5, §5.

[110] OpenSSH, *sntrup761.c* (2025), accessed 2025-04-24. URL: https://github.com/openssh/openssh-portable/blob/master/sntrup761.c. Citations in this document: §1.

[111] OpenSSL, *constant_time.h* (2025), accessed 2025-04-24. URL: https://github.com/openssl/openssl/blob/master/include/internal/constant_time.h. Citations in this document: §7.1, §7.1.

[112] OpenSSL, *ecp_nistz256.c* (2025), accessed 2025-04-24. URL: https://github.com/openssl/openssl/blob/master/crypto/ec/ecp_nistz256.c. Citations in this document: §7.1.

[113] Dag Arne Osvik, Adi Shamir, Eran Tromer, *Cache attacks and countermeasures: the case of AES* (2005); see also newer version [**129**]. URL: https://eprint.iacr.org/2005/271. Citations in this document: §3.1.

[114] Sanjay Patel, *[DAGCombiner] try to form test+set out of shift+mask patterns* (2019). URL: https://github.com/llvm/llvm-project/commit/4e54cf3e0e71b38b2fde1a815e8460b14026762a. Citations in this document: §4.10.

[115] Sanjay Patel, *[InstCombine] fold negated low-bit-mask to cmp+select* (2022). URL: https://github.com/llvm/llvm-project/commit/f9f40aa10d9862b8db1b99bb9e7f5898a90b35f7. Citations in this document: §4.10.

[116] Colin Percival, *Cache missing for fun and profit* (2005). URL: https://www.daemonology.net/papers/htt.pdf. Citations in this document: §1, §1.3, §3.1.

[117] Lucian Popescu, Nuno P. Lopes, *Exploiting undefined behavior in C/C++ programs for optimization: a study on the performance impact* (2025). URL: https://web.ist.utl.pt/nuno.lopes/pubs/ub-pldi25.pdf. Citations in this document: §4.8.

[118] Thomas Pornin, *ghash_ctmul.c* (2016). URL: https://www.bearssl.org/gitweb/?p=BearSSL;a=blob_plain;f=src/hash/ghash_ctmul.c;hb=HEAD. Citations in this document: §7.3.

[119] Thomas Pornin, *Why constant-time crypto?* (2018). URL: https://www.bearssl.org/constanttime.html. Citations in this document: §4.2.

[120] Thomas Pornin, *The problem* (2018). URL: https://www.bearssl.org/ctmul.html. Citations in this document: §3.2.

[121] Thomas Pornin, *inner.h* (2019). URL: https://www.bearssl.org/gitweb/?p=BearSSL;a=blob_plain;f=src/inner.h;hb=HEAD. Citations in this document: §7.3.

[122] Thomas Pornin, *Constant-time code: the pessimist case* (2025). URL: https://eprint.iacr.org/2025/435. Citations in this document: §1.2, §1.3, §1.3, §1.3, §3.

[123] Antoon Purnal, *PQShield plugs timing leaks in Kyber / ML-KEM to improve PQC implementation maturity* (2024). URL: https://pqshield.com/pqshield-plugs-timing-leaks-in-kyber-ml-kem-to-improve-pqc-implementation-maturity/. Citations in this document: §4.10, §4.10.

[124] Matthew Robshaw, Olivier Billet (editors), *New stream cipher designs*, Lecture Notes in Computer Science, 4986, Springer, 2008. ISBN 978-3-540-68350-6. See [24].

[125] Markku Saarinen, *ROUND 3 OFFICIAL COMMENT: FrodoKEM – CCA Bug* (2020), email dated 10 Dec 2020 07:11:18 -0800. URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/kSUKzDNc5ME/m/EMFYz9RNCAAJ. Citations in this document: §6.4.

[126] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, Srdjan Čapkun, *Breaking bad: how compilers break constant-time implementations* (2024). URL: https://arxiv.org/abs/2410.13489. DOI: 10.48550/arXiv.

2410.13489. Citations in this document: §4.12, §4.12, §4.12, §4.12, §4.12, §4.12, §4.12, §4.12, §4.12.

[127] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, Giovanni Vigna, *SOK: (state of) the art of war: offensive techniques in binary analysis*, in [1] (2016), 138–157. DOI: 10.1109/SP.2016.17. Citations in this document: §6.4.2.

[128] Laurent Simon, David Chisnall, Ross J. Anderson, *What you get is what you C: controlling side effects in mainstream C compilers*, in EuroS&P 2018 [3] (2018), 1–15. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8406587. DOI: 10.1109/EuroSP.2018.00009. Citations in this document: §4.4, §4.4, §4.4, §4.4, §4.5, §4.5, §4.5, §4.5, §4.5, §4.5, §4.5, §4.9, §7.1.

[129] Eran Tromer, Dag Arne Osvik, Adi Shamir, *Efficient cache attacks on AES, and countermeasures*, Journal of Cryptology **23** (2010), 37–71; see also older version [113]. DOI: 10.1007/s00145-009-9049-y. Citations in this document: §1, §1.3.

[130] Dongho Won, Seungjoo Kim (editors), *Information security and cryptology—ICISC 2005, 8th international conference, Seoul, Korea, December 1–2, 2005, revised selected papers*, 3935, Springer, 2006. ISBN 3-540-33354-1. DOI: 10.1007/11734727. See [108].

[131] Matthew D. Wood, *Implement fixed-window exponentiation to mitigate hyper-threading timing attacks* (2005). URL: https://github.com/openssl/openssl/commit/46a643763de6d8e39ecf6f76fa79b4d04885aa59. Citations in this document: §3.1.

[132] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, Bing Mao, *Silent bugs matter: a study of compiler-introduced security bugs*, in USENIX Security 2023 [56] (2023), 3655–3672. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao. Citations in this document: §4.8.

[133] Richard Yao, *Bug 110007 - Implement support for Clang's __builtin_unpredictable()* (2023). URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=110007. Citations in this document: §4.5.

[134] Yuval Yarom, Daniel Genkin, Nadia Heninger, *CacheBleed: a timing attack on OpenSSL constant time RSA*, in CHES 2016 [74] (2016), 346–367. URL: https://eprint.iacr.org/2016/224. Citations in this document: §3.1.

[135] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography—9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. ISBN 978-3-540-33851-2. See [21].