Why Constant-Time Crypto?

In 1996, Paul Kocher published a novel attack on RSA, specifically on RSA implementations, that extracted information on the private key by simply measuring the time taken by the private key operation on various inputs. It took a few years for people to accept the idea that such attacks were practical and could be enacted remotely on, for instance, an SSL server; see this article (http://crypto.stanford.edu/~dabo/abstracts/ssl-timing.html) from Boneh and Brumley in 2003, who conclude that:

> Our results demonstrate that timing attacks against network servers are practical and therefore all security systems should defend against them.

Since then, many timing attacks have been demonstrated in lab conditions, against both symmetric and asymmetric cryptographic systems.

This requires a few comments. First, while timing attacks work well in research conditions, they are extremely rarely spotted in the wild (I am not aware of a single case). Timing attacks usually require many attempts to gather enough samples for statistics to reveal the sought timing difference; as such, they tend to be somewhat slow and not very discreet. This does not mean that timing attacks are not real or do not apply, only that the state of the security of many systems is such that typical attackers have easier, faster ways in.

Another important point is that when timing attacks apply, they are all-encompassing: if the context is such that secret information held in a system may leak through external timing measures, then everything the system does may be subject to such leaking. This is not limited to cryptographic algorithms. Research on timing attacks tends to focus on secret keys because keys are high-value targets (a key concentrates a lot of secrecy), and cryptographers talk mostly about cryptography; however, even if all cryptographic algorithms in your system are protected against timing attacks, you are not necessarily out of trouble in that respect. In BearSSL I am doing my part, by providing constant-time implementations for all operations that are relevant to SSL; but slapping a constant-time SSL implementation over existing software is not sufficient to achieve general timing immunity. This is only a good start.

Timing attacks are a subset of a more general class of attacks known as side-channel attacks. A computer system runs operations in a conceptual abstract machine, that takes some inputs and provides some outputs; side-channel attacks are all about exploiting the difference between that abstract model and the real thing. In the context of smart card security, for instance, power analysis attacks (in particular Differential Power Analysis, that compares power usage between successive runs) have proven to be a great threat. Timing attacks still have a special place in that they can be applied remotely, through a network, while all other side-channel leakages require the attacker to be physically close to its target.

Constant-time implementations are pieces of code that do not leak secret information through timing analysis. This is one of the two main ways to defeat timing attacks: since such attacks exploit differences in execution time that depend on secret elements, make it so that execution time does not depend on secret elements. Or, more precisely, that variations in execution time are not correlated with secret elements: execution time may still vary, but not in a way that can be traced back to any kind of value that you wish to keep secret, in particular (but not only) cryptographic keys.

The other main method is masking. For instance, in the case of RSA, the core operation is computing $m^d \bmod n$, for a messsage m (encrypted message to decrypt, or hashed and padded data to sign), modulus n (a public value) and private exponent d. Masking entails generating a random integer r modulo n, and really computing $r^{-1}(mr^e)^d \bmod n$. Thus, the private exponent is applied on a value that the attacker does not know (the product of m with r^e), depriving him of any chance of correlating the execution time with the actual message or private key data.

Masking has some drawbacks, in particular:

- It works only in algorithms that have an algebraic structure that is amenable to such manipulations.
- Whether the masking is really efficient at thwarting attacks depends on a mathematical analysis which rarely leads to a conclusive proof (i.e. it more often is: "This breaks the attack as described, and nobody found a way to fix it. Yet.").
- Masking with a random value requires a source of randomness, which can be a hard requirement, especially in embedded systems. Even if a strong RNG is available at some level (a running SSL client or server mostly needs one anyway), bringing it to the implementation of a nominally deterministic algorithm (e.g. RSA decryption) can be troublesome in terms of internal API.

For these reasons, BearSSL aims for constant-time implementations for all algorithms, or at least all implementations used by default. For instance, for AES, version 0.3 contains five implementations, three of them being constant-time (and used by default), the two others meant only for special purposes.

Generic Tools

Execution Model

Most elementary operations (opcodes in the machine language) are constant-time. The potentially non-constant-time operations to look for are the following:

- **Memory accesses**. Whenever a table element is read from or written to, the address of the element may leak. In general, in the presence of caches, out-of-cache accesses take longer time; moreover, the new access will evict from cache some older data, and the eviction victim depends on the target address, so this can be detected afterwards. Cache-based leaks have been successfully leveraged to retrieve secret keys (for both RSA and

AES implementations) by remote attackers, or by attackers running their own code on the same hardware (from another unprivileged process, or even from another virtual machine co-hosted on the same CPU). All levels of cache could be exploited that way.

Even in cache-less architectures, memory accesses may leak some data; for instance, many Flash controllers will take an extra delay to serve read requests that occur at the very start of a block.

- **Conditional jumps**. Since executing code involves reading the opcodes from RAM or ROM, a conditional jump necessarily involves reading bytes from distinct addresses, so the points about memory accesses apply. But there is also an additional effect in that conditional jumps are subject to jump prediction: modern CPU try to work out in advance how the jump will be taken, and fetch the corresponding instructions. A bad prediction entails a detectable delay. Jump prediction uses both static rules, and a dedicated cache system, which can lead to the same kind of attack as table lookups.

  Note that conditional jumps may leak information on the condition. This is a problem only if the condition is secret. For instance, when implementing AES-128, there are ten rounds, so an implementation may use a loop with a conditional jump that will exit after the tenth round. That AES-128 includes ten rounds is not secret, so that specific conditional jump is not problematic.

- **Integer divisions**. On architectures that provide an integer division opcode, the hardware implementation often uses microcode, and, in some platforms, will trigger a faster code path when the divisor and/or the dividend is small. On architectures that do not provide a division opcode, use of integer division (the `/` or `%` C operators) may trigger the use of a compiler-provided subroutine that may exhibit the same behaviour.

  Some divisions can be optimised into shifts and masking, but C compilers can make surprising choices in that respect; also, the C standard mandates that `(-1) / 2 == 0`, while most CPU will right-shift `-1` into `-1` (arithmetic shift with sign extension), so divisions by powers of two may still involve some special code with conditional jumps if signed types are involved.

- **Shifts and rotations** can have an execution time that depends on the shift/rotation count. This is not the case on CPU that feature a "barrel shifter". Famously, the Pentium IV (NetBurst architecture) does not have barrel shifters, so shift and rotation counts may leak through timing. The shifted or rotated data, though, does not leak. This point impacts mostly algorithms that use shift or rotations by amounts that depend on potentially secret data (e.g. RC5 encryption).

- **Multiplications** might be a problem in older systems. This one is very irksome: in almost all recent CPU designs, multiplications are constant-time, but some older CPU may have shortcuts that make multiplications faster when operands are small. In the Intel x86 line, the 80486 had a variable-time multiplication, while the Pentium has constant-time multiplication. One of the most recent yet significant CPU designs with variable-time multiplication is the ARM9.

Not having constant-time multiplications is a severe impediment to implementations of many cryptographic algorithms, in particular (but not only) asymmetric cryptography. Since modern CPU offer constant-time multiplications, I decided to assume, in BearSSL, that integer multiplications are constant-time. This is an **important caveat** which should be minded by, in particular, people deploying systems that use ARM7 or ARM9 processors.

A dedicated page (ctmul.html) has been added to list architectures with non-constant-time multiplications, and possible workarounds.

Compiler Woes

The C programming language is defined to run in an abstract machine under the "as if" rule, so the compiler is free to translate your code in any sequence of instructions that yield the expected result, with execution time not being part of the observable elements that must be preserved.

In that respect, there is a kind of truce between compilers and cryptographic developers, with occasional incidents. In older times, a C compiler was a very straightforward piece of software, a kind of "portable assembler"; the developer would easily guess how his code would be converted to machine opcodes. This is no longer true; compilers now run complex, expensive optimisation algorithms that often bite the unwary, especially in association with the dreaded undefined behaviour. "UB" is C terminology for "anything goes", and they mean it.

For instance, adding two `int` values with a result that would not fit in the range allowed for `int` triggers UB. A "traditional" C developer would assume that the addition is mapped to the underlying opcode, so the addition would use modular reduction (value would be truncated to its low 32 bits, in two's complement representation, on a typical 32-bit machine). However, since out-of-range signed values imply UB, compiler optimisations may break such assumptions. To take a concrete example, suppose that you write an implementation for big integers, and you represent such integer values as arrays of `int32_t` values. Using a signed type would be a bit weird, but it would make some sort of sense if the developer is in fact converting an existing implementation in Java to C code, because Java does not have unsigned types[1], and guarantees modular arithmetics.

So you could have, as part of that implementation, a function that adds two big integers together. That function could look like this:

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

/*
 * Compute z = x + y. Source and destination operands have the same
 * length. The final carry is returned.
 */
int32_t
add(int32_t *z, const int32_t *x, const int32_t *y, size_t len)
{
        int32_t cc;
        size_t u;

        cc = 0;
        for (u = 0; u < len; u ++) {
                int32_t xw, yw, zw;

                xw = x[u];
                yw = y[u];
                zw = xw + yw + cc;
                z[u] = zw;
                cc &= (xw == zw);
                cc |= (zw - (int32_t)0x80000000) < (xw - (int32_t)0x80000000);
        }
        return cc;
}
```

The interesting part here is the subtractions with the `0x80000000` constants. This is typical of translated-from-Java code, in which it would look like `xw - Integer.MIN_VALUE`. The idea is that after adding two 32-bit words, there may be a carry, which is detected by the result having "wrapped around". The developer thus wants an unsigned comparison, but, in Java, there is no such thing, so instead one must add that constant value to get both comparison operands within the signed integer range, so that the signed comparison does the intended job. Our envisioned developer, translating his Java prototype into C code, will simply use the same trick, as shown above.

Now let's compile that with a fairly normal C compiler (GCC, as installed with Ubuntu 16.04, i.e. not the most confidential of things ever). With basic optimisations (`gcc -O`), we get this assembly output:

```
        .globl  add
        .type   add, @function
add:
.LFB0:
        .cfi_startproc
        testq   %rcx, %rcx
        je      .L4
        movl    $0, %r9d
        movl    $0, %eax
.L3:
        movl    (%rsi,%r9,4), %r10d
        movl    %r10d, %r8d
        addl    (%rdx,%r9,4), %r8d
        addl    %eax, %r8d
        movl    %r8d, (%rdi,%r9,4)
        cmpl    %r8d, %r10d
        sete    %r11b
        movzbl  %r11b, %r11d
        andl    %r11d, %eax
        addl    $-2147483648, %r8d
        addl    $-2147483648, %r10d
        cmpl    %r10d, %r8d
        setl    %r8b
        movzbl  %r8b, %r8d
        orl     %r8d, %eax
        addq    $1, %r9
        cmpq    %r9, %rcx
        jne     .L3
        rep ret
.L4:
        movl    $0, %eax
        ret
        .cfi_endproc
.LFE0:
        .size   add, .-add
```

(This is on a 64-bit x86 machine.)

We see the `addl $-2147483648, %r*d` opcodes, that correspond to our subtractions with `0x80000000`. So far so good.

The assembly output above does not seem that much optimised, though, so let's use more aggressive options, namely `gcc -O9`. And now we get that:

```
        .globl  add
        .type   add, @function
add:
.LFB0:
        .cfi_startproc
        testq   %rcx, %rcx
        je      .L4
        xorl    %r8d, %r8d
        xorl    %eax, %eax
        .p2align 4,,10
        .p2align 3
.L3:
        movl    (%rsi,%r8,4), %r10d
        movl    (%rdx,%r8,4), %r9d
        xorl    %r11d, %r11d
        addl    %r10d, %r9d
        addl    %eax, %r9d
        cmpl    %r9d, %r10d
        movl    %r9d, (%rdi,%r8,4)
        sete    %r11b
        andl    %r11d, %eax
        cmpl    %r9d, %r10d
        setg    %r9b
        addq    $1, %r8
        movzbl  %r9b, %r9d
        orl     %r9d, %eax
        cmpq    %r8, %rcx
        jne     .L3
        rep ret
.L4:
        xorl    %eax, %eax
        ret
        .cfi_endproc
.LFE0:
        .size   add, .-add
```

The additions are gone! What has happened here? Namely, the aggressive optimiser noticed that the same constant was subtracted from both operands of the comparison operation, and so they should not, mathematically speaking, impact the comparison result. Indeed, operations that go out of range for signed integer types trigger UB, so the compiler may assume that operations do not go out of range, and optimise things accordingly. However, the "going out of range" is important in the code above, and the removal of the `addl` opcodes breaks the code (it will compute wrong carries).

At that point, many developers will cringe and wail, and denounce that treacherous compiler that uses UB as an excuse to backstab them and make the software blow up. Compiler implementors will retort that this is a natural consequence of running the optimisation algorithms, totally within the range of actions allowed by the standard, and they do such optimisations because the same whining developers are begging for removal of unnecessary opcodes, since they cannot apparently be bothered with not writing them in the first place. Some people will try to keep everybody content by saying that GCC has an option called `-fwrapv` that guarantees Java-like modular semantics, and others will then point out that GCC is not the only C compiler in the world.

This example shows the difficulty of ensuring that a given piece of code is translated correctly. This particularly impacts efforts at making constant-time code, because the result being not constant-time will be tough to test. In the example above, the code triggers UB and at high optimisation levels, it simply breaks (it computes the wrong values). But a constant-time code may also be compiled into machine code that computes the right value but with a variable time, and this won't be detected during development.

Constant-time development thus requires heavy mental gymnastics to try to predict what various present and future compilers will do to the code, fueled by a maniacal knowledge of the darkest details for the C standard. A few generic guidelines may help:

- **Avoid boolean types** (e.g. the C99 type `_Bool`). Instead, use large "control words" (say, 32 bits) that the caller will set to 1 or 0, to enable or disable a specific computation. The C compiler cannot assume that the value is always 0 or 1 unless it sees the call site, and thus won't try to replace bitwise boolean operations with conditional jumps.
- **Use unsigned types**. In C, unsigned types are guaranteed to use modular arithmetic, so you get a clean truncation, and no UB.
- **Look at the assembly**. While inspecting the assembly output will only teach you what a specific version of the compiler on a specific architecture with specific switches did, it may give you a fair level of intuition about what compilers can actually do to your code[2].

C Code Snippets

A very good source on how to implement constant-time operations in C is the Cryptography Coding Standard (https://cryptocoding.net/index.php/Cryptography_Coding_Standard), a collaborative effort for describing issues related to secure implementation of cryptographic algorithms, and, more generally, of code that processes secret data elements.

The same techniques can be observed in BearSSL's source code, especially in src/inner.h (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/inner.h;h=472dc2e6f1099539a9371595f4e74db5b45e6d7b;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l365).

In BearSSL, control words (parameters used to enable or disable a specific computation) have type `uint32_t` and value 0 or 1. This matches the examples in the Cryptography Coding Standard; however, this is not the only possible convention. A case can be made that the "enable" word should be -1, not 1, because -1 in an unsigned integer type yields an all-one word, which is convenient in some constructions. I tried both and the "0/1" convention seemed slightly better in a C context than the "0/-1" convention.

An important tool for constant-time code is a conditional copy, which copies or not a source array into a destination array. The copy is enabled by a control word, but the same memory accesses are performed whether the control word is 0 or 1. In BearSSL, this is done with the `br_ccopy()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/codec/ccopy.c;h=2beace723b03a526b280396d1ccf13b1cb7a540a;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l27) function.

A use case of conditional copy is constant-time table lookup. This is used, for instance, in the implementation of elliptic curves. The normal point multiplication algorithm is a double-and-add system; a classic optimisation (called "window") consists in doing several doubles in succession, then one addition with a proper multiple of the point. For instance, a simple double-and-add algorithm for multiplying point P by integer n processes the multiplier bit by bit:

1. Set Q = 0 (the "point at infinity")
2. Compute Q ← 2·Q
3. If next bit of n is set, then add P to Q
4. Loop to step 2 until end of multiplier is reached

In a constant-time implementation, the addition at step 3 must be done each time, since you do not want to leak whether the corresponding bit of n is 0 or 1. This is expensive; with Jacobian coordinates, a generic point addition costs about twice the cost of a doubling. BearSSL thus uses a 2-bit window, like this:

1. Compute 2·P and 3·P in temporary variables
2. Set Q = 0 (the "point at infinity")
3. Compute Q ← 2·Q
4. Compute Q ← 2·Q again
5. Depending on the next two bits of n, add 0, P, 2·P or 3·P to Q
6. Loop to step 3 until end of multiplier is reached

This basically avoids half of the point additions (the number of doubling is unchanged). However, this entails making a lookup of the value to add in a table that contains 0, P, 2·P and 3·P. Since the lookup index is a pair of bits of the multiplier and is secret, the lookup MUST be done in a constant-time way. In practice, this entails using `br_ccopy()` on P, 2·P and 3·P, with control words that will read only one of them.

Bitslicing

There is a classical implementation technique, that has been discovered several times, in several distinct fields. When it was first applied to cryptography, it was called bitslicing. Another, older, non-crypto name is "data orthogonalisation".

Bitslicing relies on the following idea: if you have a 32-bit data element in an algorithm, don't store it in a variable of size 32 bits (or more). Instead, store it in 32 distinct variables: the data bits will be spread into the bit 0 of each of the distinct variables. Then, express all operations with bitwise operators (like XOR or AND) as you would in a dedicated circuit with transistors.

This process, in the field of cryptography, was first applied to the DES encryption algorithm; it was the work of Eli Biham, but the name "bitslice" was coined by Matthew Kwan, who proceeded to make the best known bitslice implementation of the DES S-boxes (at that time), described here (http://www.darkside.com.au/bitslice/). The DES algorithm internally uses eight functions called "S-boxes", each taking 6 bits as input and producing 4 bits of output. A basic implementation would simply use a table with 64 entries; in bitslice code, the first S-box becomes this:

```
static void
s1 (
	unsigned long	a1,
	unsigned long	a2,
	unsigned long	a3,
	unsigned long	a4,
	unsigned long	a5,
	unsigned long	a6,
	unsigned long	*out1,
	unsigned long	*out2,
	unsigned long	*out3,
	unsigned long	*out4
) {
	unsigned long	x1, x2, x3, x4, x5, x6, x7, x8;
	unsigned long	x9, x10, x11, x12, x13, x14, x15, x16;
	unsigned long	x17, x18, x19, x20, x21, x22, x23, x24;
	unsigned long	x25, x26, x27, x28, x29, x30, x31, x32;
	unsigned long	x33, x34, x35, x36, x37, x38, x39, x40;
	unsigned long	x41, x42, x43, x44, x45, x46, x47, x48;
	unsigned long	x49, x50, x51, x52, x53, x54, x55, x56;
	unsigned long	x57, x58, x59, x60, x61, x62, x63;

	x1 = ~a4;
	x2 = ~a1;
	x3 = a4 ^ a3;
	x4 = x3 ^ x2;
	x5 = a3 | x2;
	x6 = x5 & x1;
	x7 = a6 | x6;
	x8 = x4 ^ x7;
	x9 = x1 | x2;
	x10 = a6 & x9;
	x11 = x7 ^ x10;
	x12 = a2 | x11;
	x13 = x8 ^ x12;
	x14 = x9 ^ x13;
	x15 = a6 | x14;
	x16 = x1 ^ x15;
	x17 = ~x14;
	x18 = x17 & x3;
	x19 = a2 | x18;
	x20 = x16 ^ x19;
	x21 = a5 | x20;
	x22 = x13 ^ x21;
	*out4 ^= x22;
	x23 = a3 | x4;
	x24 = ~x23;
	x25 = a6 | x24;
	x26 = x6 ^ x25;
	x27 = x1 & x8;
	x28 = a2 | x27;
	x29 = x26 ^ x28;
	x30 = x1 | x8;
	x31 = x30 ^ x6;
	x32 = x5 & x14;
	x33 = x32 ^ x8;
	x34 = a2 & x33;
	x35 = x31 ^ x34;
	x36 = a5 | x35;
	x37 = x29 ^ x36;
	*out1 ^= x37;
	x38 = a3 & x10;
	x39 = x38 | x4;
	x40 = a3 & x33;
	x41 = x40 ^ x25;
	x42 = a2 | x41;
	x43 = x39 ^ x42;
	x44 = a3 | x26;
	x45 = x44 ^ x14;
	x46 = a1 | x8;
	x47 = x46 ^ x20;
	x48 = a2 | x47;
	x49 = x45 ^ x48;
	x50 = a5 & x49;
	x51 = x43 ^ x50;
	*out2 ^= x51;
	x52 = x8 ^ x40;
	x53 = a3 ^ x11;
```

```
        x54 = x53 & x5;
        x55 = a2 | x54;
        x56 = x52 ^ x55;
        x57 = a6 | x4;
        x58 = x57 ^ x38;
        x59 = x13 & x56;
        x60 = a2 & x59;
        x61 = x58 ^ x60;
        x62 = a5 & x61;
        x63 = x56 ^ x62;
        *out3 ^= x63;
}
```

As you see, all actions are done with bitwise AND, OR and XOR operators. While this may seem to be a very roundabout and inefficient way to replace a table lookup, the bitslicing technique comes with a huge bonus through parallelism: since all these operations are done with bitwise operators, then they naturally operate on several instances in parallel, at the same time.

For instance, if you use that code with 64-bit integer types, then every time this function is invoked, it computes the first DES S-box on the bitsliced input in bits 0 of parameters `a1` to `a6`, but it also does so on the input stored in bits 1 of parameters `a1` to `a6`, and so on. Thus, this code, which consists in 56 bitwise operations, actually computes 64 parallel instances of the S-box, each with its own input data. On average, this means a cost of less than one operation per S-box instance, which compares favourably to a table lookup.

An additional bonus from bitslicing is in the implementation of bit permutations. There are several such operations in DES. In a classical implementation, this needs to be computed with either sequences of shifts and masks and recombinations, or with more lookups in precomputed tables. In a bitslice representation, though, permuting the bits really means using the "right" variables in the next step; this is mere data routing, which is resolved at compile-time, with no cost at all at runtime! This characteristic, combined with the efficient computation of DES S-boxes, made bitslicing very tempting for computational tasks involving a lot of parallel DES instances, namely cracking passwords which were hashed with the traditional DES-based Unix `crypt()` function.

In our goal of achieving constant-time implementations, bitslicing offers an interesting characteristic, in that it replaces table lookups (that may leak secret information) with sequences of bitwise operations that are naturally constant-time. Bitslicing, though, comes with some extra costs, which can be summarised as follows:

- Bitslicing uses a lot of variables, usually many more than the number of registers offered by a typical CPU (x86 CPU have 7 or 15 general-purpose registers, depending on operation mode; PowerPC raise that number to 30 or so). On big, modern platform, this is not that much of an issue: extra variables will be stored on the stack, and the exchange of data between stack and registers will be interleaved with the bitwise binary operations. Since such CPU are superscalar, theses data exchanges will be basically "free": the ALU will be fully occupied, and there will be enough free slots for exchanges to be done without slowing down computations. However, the same is not true on reduced platforms, which are not necessarily able to run a bitwise operation and a memory access in the same clock cycle.
- Bitslicing tends to require large code. A rule of thumb is that the code that replaces a lookup table will be about as large as the table that it replaces, but this depends a lot on the architecture. On platforms where elementary opcodes are 32 bits each (e.g. PowerPC, SPARC...), bitslice code will typically be bigger. Thus, bitslicing tends to be incompatible with a goal of extreme code footprint reduction.
- Bitslicing works over orthogonalised data[3]. This operation can be implemented relatively efficiently, but still has some non-negligible overhead.
- Bitslicing really shines in highly parallel contexts. For instance, if you bitslice DES on a machine with 64-bit registers, then best performance is achieved when you do have 64 instances of DES to run at the same time. This is possible with, for instance, decryption in CBC mode; but not with encryption in CBC mode. In CBC encryption, the input to each DES invocation depends on the output of the previous invocation, which prevents parallelism. This would basically divide computation speed by 64.

Some of these issues can be worked around to some extent by using mixed strategies. For instance, an AES encryption round involves applying 16 identical S-boxes on 16 different input bytes; these are good candidates for being executed in parallel in bitslice mode, while they still relate to the same AES invocation instance, and other parts of the algorithm do not offer that level of parallelism.

Constant-Time in BearSSL

AES

There are five implementations of AES in BearSSL, called "big", "small", "ct", "ct64" and "x86ni".

- The "big" implementation is a classic, table-based implementation, which is not constant-time. It offers good performance (about 170 MB/s, or 18 cycles per byte, on a 3.1 GHz Ivy Bridge core from 2012), but it is not constant-time.
- The "small" implementation is a straightforward translation of the AES specification, aiming at a very small code size, at the expense of performance. It is small, slow, and not constant-time.
- The "ct" implementation uses a mixed bitslice strategy to offer constant-time processing. It is the default implementation on 32-bit platforms.
- The "ct64" implementation is similar to "ct" but with 64-bit variables, thereby being almost twice as fast on 64-bit architectures when the encryption mode allows for parallelism. It is the default implementation on 64-bit platforms.

- The "x86ni" implementation uses the AES-NI opcodes present in recent x86 CPU. It is much faster than all other implementations, and constant-time, and yields small code. However, it is specific to x86 processors, and not all of them support the relevant opcodes, thereby needing a "fallback" mechanism on one of the other implementations.

The bitslicing technique relies on a circuit implementing the AES S-box and described by Boyar and Peralta in 2009 (https://eprint.iacr.org/2009/191). The resulting code (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/aes_ct.c;h=66776d9e206c92cbbf3fa799c651a3d3652bd75a;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l27). takes as inputs eight 32-bit words, and computes the AES S-box (that takes eight input bits) on all 32 instances in parallel. Since an AES round involves 16 S-boxes, this means that the S-boxes for two parallel AES instances can be processed at the same time.

This implementation strategy is similar to that of the Käsper-Schwabe implementation (https://eprint.iacr.org/2009/129), although there are some differences:

- Their code uses SSE2 registers, which are 128-bit each, and they can thus process eight AES instances in parallel, for really good performance (about twice as fast as the best table-based implementations) while still being constant-time. BearSSL's "ct" implementation is for 32-bit systems, thus only 32 bits per register.
- Since the implementation strategy runs AES S-boxes from the same instance, the "transverse" operations (ShiftRows, MixColumns) require some explicit bit/byte mangling. Käsper and Schwabe leverage special SSE2 opcodes to do that efficiently; BearSSL cannot use such luxuries since the code aims at portability on platforms that have only basic operations. The layout of bits in words is also different.
- BearSSL supports CTR but also CBC encryption, which means that an AES decryption implementation must be provided as well (CTR mode uses the block cipher in encryption mode only, which makes things simpler).

Doing all the circuit optimisation for the AES S-box in the decryption direction seemed to be a lot of work, so I did not do it. Moreover, an optimised circuit in bitslice code implies a large footprint; a system using AES in CBC mode will need encryption and decryption, so it would make sense to try to mutualise part of the computation. Therefore, BearSSL's code reuses the AES S-box code to compute the inverse S-box. The trick is that AES S-box can be expressed algebraically as:

$$S(x) = A(I(x)) \text{ xor } 0x63$$

where I() is inversion in the finite field GF(256) (0 is formally defined to be its own inverse), and A() is a specific linear transform. Since inversion is an involution, one can express the inverse S-box as:

$$S^{-1}(x) = A^{-1}(S(A^{-1}(x \text{ xor } 0x63)) \text{ xor } 0x63)$$

The bitslice code corresponding to A() and $A^{-1}()$ is small with regards to the code that computes the inverse I(), so this formula allows reusing the bulk of the implementation of the S-box to compute the inverse S-box. This yields this code (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/aes_ct_dec.c;h=7f32d2bd446da4e1fce668da8c5f476b5ce85e86;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l27).

This trick implies a slight performance overhead for decryption. However, CBC decryption is parallel, while CBC encryption is not. Therefore, in the "ct" implementation where two AES instances can be run in parallel, CBC decryption is inherently faster than CBC encryption, so it makes sense to push the extra overhead on the decryption engine.

The "ct64" implementation is similar to "ct" but with 64-bit words. On a 64-bit platform, this yields a substantial performance enhancement. Here are some figures, for an Intel 3.1 GHz Ivy Bridge, compiled with GCC in 64-bit mode:

Implementation	CBC (enc)	CBC (dec)	CTR (enc/dec)	size (CBC)	size (CTR)
big	161.40	177.02	169.93	5994	3050
small	39.05	22.48	38.54	3207	1879
ct	28.15	43.02	53.87	5803	3604
ct64	27.39	77.05	90.90	6549	4535
x86ni	679.76	2369.34	2420.70	3290	2750

All speed values are in megabytes per second, for a 128-bit key. The "size" values correspond to the total code footprint for CBC and CTR implementations, respectively (in bytes, for encryption and decryption together).

Note that this specific machine has 100Mbit/s connectivity (megabits, not megabytes) so even with the slowest of these implementations (the "small" code, in CBC decryption mode), the encryption cost at full bandwidth uses less than half of one of the four CPU cores. It shall also be taken into account that these values were obtained on a "big" CPU with a 64-bit architecture; I will provide benchmarks on a "small" system (say, a low-power 32-bit ARM) when I have them.

DES

BearSSL includes two implementations of DES, called "tab" and "ct". The "tab" implementation uses tables and is not constant-time. The "ct" implementation uses a mixed bitslice strategy, and is constant-time (but also about three times slower than the "tab" implementation).

Bitslicing was originally described and optimised for DES, but in a massively parallel context (password cracking). In SSL, use of DES (really, 3DES) is for CBC encryption and decryption; CBC encryption is not parallel, hence a mixed strategy is needed.

The difficulty in such an endeavour is that DES uses eight different S-boxes, each with its own circuit. To allow for computing all S-boxes of one DES round in parallel, the following technique was used:

- Formally split each S-boxes (6 bits input, 4 bits output) into four S-boxes that take 6 bits of input and yield 1 bit of output. We now have 32 boxes to implement (let's call them "T-boxes").

- A generic function that takes some bits of input and provides one bit of output can be built with a recursive tree of multiplexers. Namely, to implement a function that takes 6 bits of input, use the first bit to select between two results:

  T(a, b, c, d, e, f) = MUX(a, T(0, b, c, d, e, f), T(1, b, c, d, e, f))

  Applying recursively on the two sub-functions, this ends up with a tree structure with 6 levels; one input bit is used at each level, and the actual function definition is the set of 64 leaves, each being a 1-bit constant.

- Now that the 32 T-box functions are represented with the same tree-like circuit, differing only in the 64 constants they feed on, we make these constants extra inputs: the circuit now takes 70 inputs, and all 32 functions are identical.

- Bitslicing occurs to evaluate these 32 functions in parallel with our 32-bit words.

The resulting code can be observed there (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/des_ct.c;h=581c0ab294557614efeac6004a216aa082b23010;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l235). Though the performance is not great (6.53 MB/s on my 3.1 GHz Intel Ivy Bridge CPU with 3DES, vs 20.26 MB/s for the "tab" implementation), it can still be serviceable in some situations where 3DES must be used for backward compatibility. Notably, some embedded systems have weak CPU but even slower bandwidth, e.g. 9600 bauds serial lines, making even 3DES comparatively cheap. In any case, BearSSL puts by default 3DES at the very bottom of its cipher suite lists, thereby using it only when there is no other choice.

GHASH (for GCM)

GHASH is a "universal hash function", an animal quite distinct from usual hash functions such has SHA-256. The GCM encryption mode combines a block cipher in CTR mode (usually AES) and GHASH into an Authenticated Encryption with Additional Data mode (AEAD), which provides both confidentiality (through encryption) and checked integrity (with a MAC), properly assembled and running on the same secret key.

At its core, GHASH is a polynomial evaluation in the finite field GF(2^{128}); almost all of the cost and complexity is, for each 128-bit block of data, computing a multiplication in that finite field. This is a field of characteristic 2, which means that the multiplication is "special": addition is replaced with bitwise XOR, so it has been called "carryless". Some recent CPU have dedicated opcodes for that (the `pclmulqdq` opcode was added to recent x86 CPU, along with the opcodes that implement AES rounds in hardware), but "small" CPU do not have such an opcode, and it would not be accessible through portable C code anyway.

Classical implementations of GHASH rely on interpreting the multiplication as a linear operation in a vector space, and breaking it down in a number of table lookups, to process the input by chunks of 8, 12 or 16 bits. Such tables must be generated from the secret key, so they imply some non-negligible overhead, and they use up RAM, not ROM. Alternatively, ROM tables for generic carryless multiplication can be used, and Karatsuba decomposition is used to extend these tables into a full 128-bit carryless multiplication. Either way, tables mean non-constant-time, unless you push the idea to the extreme of tables with only two entries (for 0 and 1). Käsper and Schwabe do just that in their implementation, with 128-bit SSE2 registers to speed up the process. They still need about 2 kB of tables (key-dependent, so RAM-based), and even with SSE2 the performance is unconvincing, in that their constant-time GHASH is twice slower than their AES-CTR code.

BearSSL uses a different implementation strategy. I have never seen it described anywhere else, but it seems rather obvious so it is improbable that I am the first to think about it. The idea is to use integer multiplications, with "holes" to avoid carry spilling. To see how it works, consider an integer multiplication of two 4-bit values, let's say 13 and 11. Doing the multiplication "by hand" in binary, you get something like this:

```
          1 1 0 1
        x 1 0 1 1
          -------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
---------------
1 0 0 0 1 1 1 1
```

Basically, we get shifted versions of the multiplicand (corresponding to the bits equal to 1 in the multiplier), that then get added together. This addition involves carry propagation; the carryless multiplication would use a XOR, without any carry propagation. How do we turn integer multiplication into carryless multiplication? The solution is to spread apart the data bits with "holes" (sequences of zeros) between them to allow carries to spill harmlessly. With the values above, this would look like this:

```
                  1 0 0 1 0 0 0 0 0 1
                x 1 0 0 0 0 0 1 0 0 1
                  -------------------
                  1 0 0 1 0 0 0 0 0 1
                0 0 0 0 0 0 0 0 0 0
              0 0 0 0 0 0 0 0 0 0
            1 0 0 1 0 0 0 0 0 1
          0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0 1
-------------------------------------
1 0 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 1
^     ^     ^     ^     ^     ^     ^
```

The "^" signs indicate the bits we are interested in. Indeed, if we keep only these bits, we get `1111111`, which is the carryless product of `1101` by `1011`. The extra carry that occurred near the middle pushed a 1 into a hole, that we can ignore through masking.

Extending that process into 32-bit words, we can use 8 data bits per word, with three zero-bits between any two successive data bits. We need holes that large because with 8 data bits in the multiplier and multiplicand, we may need to add togeter up to 8 bits of value `1`, which needs three extra bits to spill without altering the next column. If we used only two bits for the hole, then we could not use more than 7 data bits in either the multiplier or the multiplicand.

The end-result can be observed in `src/hash/ghash_ctmul.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/hash/ghash_ctmul.c;h=362320252f3b5930379635c1e59d2b6f1eba0121;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l160), in a function which is reproduced here:

```
static inline void
bmul(uint32_t *hi, uint32_t *lo, uint32_t x, uint32_t y)
{
        uint32_t x0, x1, x2, x3;
        uint32_t y0, y1, y2, y3;
        uint64_t z0, z1, z2, z3;
        uint64_t z;

        x0 = x & (uint32_t)0x11111111;
        x1 = x & (uint32_t)0x22222222;
        x2 = x & (uint32_t)0x44444444;
        x3 = x & (uint32_t)0x88888888;
        y0 = y & (uint32_t)0x11111111;
        y1 = y & (uint32_t)0x22222222;
        y2 = y & (uint32_t)0x44444444;
        y3 = y & (uint32_t)0x88888888;
        z0 = MUL(x0, y0) ^ MUL(x1, y3) ^ MUL(x2, y2) ^ MUL(x3, y1);
        z1 = MUL(x0, y1) ^ MUL(x1, y0) ^ MUL(x2, y3) ^ MUL(x3, y2);
        z2 = MUL(x0, y2) ^ MUL(x1, y1) ^ MUL(x2, y0) ^ MUL(x3, y3);
        z3 = MUL(x0, y3) ^ MUL(x1, y2) ^ MUL(x2, y1) ^ MUL(x3, y0);
        z0 &= (uint64_t)0x1111111111111111;
        z1 &= (uint64_t)0x2222222222222222;
        z2 &= (uint64_t)0x4444444444444444;
        z3 &= (uint64_t)0x8888888888888888;
        z = z0 | z1 | z2 | z3;
        *lo = (uint32_t)z;
        *hi = (uint32_t)(z >> 32);
}
```

The first few lines split the two input words into multiple "words with holes". Since each 32-bit word becomes four words with holes, we end up computing 16 multiplications (the `MUL()` macro merely multiplies two `uint32_t` values, with a `uint64_t` result). We then mask out the holes in the `z*` values (because carries may have spilled into these holes), and we finally assemble the bits back into the final result.

This function computes a carryless multiplication of two 32-bit words. For the full GHASH, Karatsuba decomposition is used: a carryless multiplication of two 128-bit words is broken down into nine carryless multiplications of 32-bit words.

A 64-bit version of that algorithm is implemented in `src/hash/ghash_ctmul64.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/hash/ghash_ctmul64.c;h=a46f16fee977f6102abea7f7bcdf169a013c3e8e;hb=5f045c759957fdff8c85716e6af99e10901fdac0). This one has an extra twist: there is no portable way to obtain a 128-bit integer type. Some compilers have it, but not all of them. Thus, the multiplication of two 64-bit values will yield only a 64-bit result. In order to get the upper 64 bits, we use bit reversal. If we define that $\mathrm{REV}_N(x)$ reverses the order of bits in the N-bit word x (that is, bit 0 becomes bit N-1, bit 1 becomes bit N-2, and so on), then we have the following property:

$$\mathrm{REV}_{127}(a \times b) = \mathrm{REV}_{64}(a) \times \mathrm{REV}_{64}(b)$$

In other words, if we bit-reverse the two operands, this bit-reverses the result (why this works is "obvious" if you refer to the "by hand" multiplication algorithm described above — of course it would not work as well with integer multiplication, because carries propagate to the left, not to the right). So, if we have a function that yields the low 64 bits of the result, then it suffices to invoke that function twice, with the operands and again with the bit-

reversed operands, to obtain the full result.

One should note that then using 64-bit words with three-bit holes, there are 16 data bits in each word, so the integer multiplication may end up adding up to 16 bits of value 1 together, for a result that does not fit in the allocated hole. However, this may happen only for the top bits in the 64-bit words, so the extra carry will be truncated anyway. This would be a problem if we used 128-bit integer results, though.

The `src/hash/ghash_ctmul32.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/hash/ghash_ctmul.c;h=362320252f3b5930379635c1e59d2b6f1eba0121;hb=5f045c759957fdff8c85716e6af99e10901fdac0) file implements the bit-reversal technique with 32-bit integers. That implementation thus uses no 64-bit integers. This code is faster than the "ctmul" implementation on some architectures:

- On architectures that support pipelining of operations, the multiplication opcode may yield the upper word of the result in a dedicated register, creating contention and thus preventing the pipelining to operate properly (on x86 systems in 32-bit mode, the `mul` opcode always targets registers `eax` and `edx`, while the `imul` opcode, which provides only the low 32 bits of the result, can use arbitrary registers).
- Some architectures simply do not produce the high 32 bits, so any multiplication with a 64-bit result will internally use several multiplications and additions with carry propagation. An example of such an architecture is the low-power ARM Cortex M0.

Some basic measures on x86 (32-bit mode) show that "ctmul32" is slower than the plain "ctmul", but this does not necessarily hold for all architectures.

An important caveat that bears repeating is that all these implementations are constant-time only as long as the underlying multiplication opcodes are constant-time. As was noted previously, some historically widespread CPU had multiplications with varying execution time. On the other hand, all recent designs, even "small" CPU, have constant-time multiplications[4].

Another constant-time implementation strategy for GHASH is to use bitslicing, reducing the multiplication to its elementary circuit. On binary fields, a remarkable multiplication method due to Cantor and Kaltofen (http://www4.ncsu.edu/~kaltofen/bibliography/91/CaKa91.pdf) uses discrete Fourier transform to achieve O(n log n log log n) asymptotic complexity, that works especially well when working in a finite field GF(2^k) with k being a power of 2 (which is exactly the situation for GHASH). A 2014 article by Bernstein and Chou (https://eprint.iacr.org/2014/729) uses that algorithm to make an extremely fast GHASH implementation. However, that speed result is achieved through full bitslicing and 128-bit SSE2 registers; the code has a large footprint and a substantial overhead which makes it less desirable for small messages (e.g. SSL records are at most 16 kB in length).

Whether this Cantor-Kaltofen multiplication can be implemented in a mixed strategy (bitslicing for parallel evaluation of components which are part of the same instance), and whether the result yields good performance for small architectures (taking into account code size, RAM usage, and CPU cost), remains to be investigated.

BearSSL-0.3 adds an extra GHASH implementation that simply uses the `pclmulqdq` opcode, which is implemented in recent x86 CPU for that very purpose. Resulting code is constant-time, compact, and much faster than all other strategies explained here. The code (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/hash/ghash_pclmul.c;h=c70988933df551395cb5bb07fa120beb2c59d933;hb=5f045c759957fdff8c85716e6af99e10901fdac0) uses the "intrinsics", which are functions provided by recent compilers to allow access to such opcodes without resorting to inline assembly.

The downside of that implementation is that it is present only on x86 systems, and not all of them provide support for these opcodes. A dedicated function called `br_ghash_pclmul_get()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/hash/ghash_pclmul.c;h=c70988933df551395cb5bb07fa120beb2c59d933;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l237) is used to safely test for support (both compile-time and runtime) and use another implementation as fallback.

ChaCha20

The ChaCha20 stream cipher is "naturally" constant-time: the implementation relies only on bitwise word operations, additions, and rotations by a fixed amount. The straightforward implementation in BearSSL (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/chacha20_ct.c;h=9961eb119f217eb818ee137e29d4dffbc82d1d25;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l27) is constant-time and decently fast (e.g. it is faster than the non-constant-time AES implementation called "big"); it is also very compact.

Poly1305

The Poly1305 MAC algorithm is akin to GHASH except that it works in GF(2^{130}-5), i.e. with integers modulo a 130-bit prime. Integer multiplication opcodes are the most natural implementation strategy; however, care must be taken with regards to carry propagation in order to achieve constant-time execution[5].

BearSSL includes three implementations of Poly1305. The generic "32-bit" implementation is in `src/symcipher/poly1305_ctmul.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/poly1305_ctmul.c;h=150e610a9c9b6c5433daf9e6f1700b534c0fb3d0;hb=5f045c759957fdff8c85716e6af99e10901fdac0). That code is closely inspired from a well-known public domain implementation (https://github.com/floodyberry/poly1305-donna) by Andrew Moon. The gist of the code is to break the 130-bit values into five 26-bit words, stored in 32-bit variables. The 6 extra bits can be used to contain spilling carries, thereby making less frequent carry propagations across words. Processing a single 128-bit block will require 25 multiplications (32-bit operands, 64-bit result), which compares favourably to the 144 multiplications used by the "ctmul" GHASH implementation.

In practice, this Poly1305 implementation is much faster than ChaCha20 (about four times as fast on my 3.1 GHz Intel CPU), so while it could probably be sped up with 64-bit multiplications and/or SSE2 tricks, there is no urgency to do so: in the context of SSL, ChaCha20 and Poly1305 are used on almost the same amount of input data.

Another Poly1305 implementation is in `src/symcipher/poly1305_ctmul32.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/poly1305_ctmul32.c;h=15d9635dd4405c955188a4d7670b2203e987f3bf;hb=5f045c759957fdff8c85716e6af99e10901fdac0). It uses the same techniques as the "ctmul" code, but with 13-bit words instead of 26-bit words. This allows the use of 32→32 multiplications only, which greatly speeds up things on ARM Cortex M systems, and also allows constant-time operations on "uncooperative" architectures (like the M0 and M3).

A third Poly1305 implementation is defined in `src/symcipher/poly1305_i15.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/poly1305_i15.c;h=6f89212193d1af5a8acd952f026d0b22c684daf6;hb=5f045c759957fdff8c85716e6af99e10901fdac0). It uses the generic "i15" big integer code. It is thus quite slow, but can share code with RSA and EC. This is meant mostly for tests, or for applications that are desperate for code size.

Hash functions

Hash functions implemented by BearSSL (MD5, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512) consist in bitwise logical operations and additions on 32-bit or 64-bit words, naturally yielding constant-time operations.

HMAC

HMAC is naturally as constant-time as the underlying hash function. The size of the MACed data, and the size of the key, may leak, though; only the contents are protected.

We still want to do something more advanced, in order to better support the MAC verifications on SSL records. SSL, famously, got things in the wrong order: it does MAC-then-encrypt instead of encrypt-then-MAC (that is, the MAC is computed over the plaintext, appended to the plaintext, and finally encrypted along with the plaintext). A number of padding oracle attacks have been demonstrated against SSL records, thereby requiring use of constant-time processing techniques. An important element of these techniques is the ability to compute HMAC without revealing the exact size of the input data.

In BearSSL this is implemented in `src/mac/hmac_ct.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/mac/hmac_ct.c;h=3237885d7afcc1ba1a025651bb63f203afb6db1f;hb=5f045c759957fdff8c85716e6af99e10901fdac0). The basic principle is imported from a description by Adam Langley (https://www.imperialviolet.org/2013/02/04/luckythirteen.html), that works for Merkle-Damgård functions (i.e. MD5 and the SHA-1/SHA-2 family, which are the functions used in SSL). The idea is to use the constant-time primitives to assemble the proper block values for the hash functions, with the MD padding when appropriate, and saving the right output. Invocation of `br_hmac_outCT()` provides the input data, its exact length, as well as the minimal length (`min_len`) and maximal length (`max_len`). The code will take care to execute the exact same instructions for all possible values of `len` between `min_len` and `max_len`. Most of the complexity is in the handling of the variants between functions (MD5 uses little-endian, the SHA functions use big-endian; SHA-384 and SHA-512 use 128-byte blocks and a 128-bit counter in the MD padding, while the other functions use 64-byte blocks and a 64-bit counter in the MD padding).

CBC padding

In order to thwart padding oracle attacks, including the Lucky Thirteen attack (http://www.isg.rhul.ac.uk/tls/Lucky13.html) which is a padding oracle based on a timing leak, the processing of an incoming record should use a fully constant-time code, from the moment the record has been obtained from the network, to the production of the plaintext along with the boolean result that qualifies the correctness of the MAC and padding.

There again, BearSSL follows Adam Langley's tracks. The code can be seen in `src/ssl/ssl_rec_cbc.c` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/ssl/ssl_rec_cbc.c;h=c0806049e3cdc8b7135ea4c9968b60e99a7f4770;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l97). The processing outline is the following:

1. The record is decrypted, including MAC and padding. This step does not check the padding.
2. The padding length is obtained (from the last byte), from which the actual plaintext length is computed. Note that a short, invalid record may claim a longer padding than possible, so the code must take care not to be fooled. A running flag (called `good`, of type `uint32_t`) is used to keep track of any error.
3. The padding contents are checked. All bytes that may be part of padding are read (padding size may be anywhere between 1 and 256 bytes in length, provided that it leaves enough room for the MAC). All padding bytes shall have the same value (which is one less than the total padding length); if any of the bytes does not have the right value, the `good` flag is cleared, but processing continues.
4. The MAC value is extracted. Since the position of the MAC depends on the padding length[6], the extraction must read all bytes that could conceivably be part of the MAC.
5. The extraction step results in a "rotated" MAC value in the buffer. In order to restore it to its rightful position, we must use a constant-time buffer rotation.

   Here we slightly optimise over Langley's solution. He uses an integer division (remainder operator in C) to get the right offsets for each byte, which implied a small leak since the division opcode is not constant-time (he had to add a work-around); moreover, he uses two nested loops for a rotation cost in $O(n^2)$ (for a MAC length of n bytes). In BearSSL, the two nested loops have cost only O(n log n), and there is no integer division: we simply use the bits of the rotation count as enablers for a conditional rotation by a fixed amount.
6. The actual HMAC value is recomputed with the constant-time function `br_hmac_outCT()`.
7. The extracted-and-rotated HMAC value, and the recomputed value, are compared. The comparison is constant-time and hits all bytes, using a XOR and a constant-time comparison with zero to update the `good` flag.
8. Finally, the resulting plaintext length is compared with the maximum allowed length for a SSL record plaintext (encrypted records can be larger, because of the MAC and the padding).

When all of this is done, the "critical section" is finished, and we can use the value of the `good` flag to characterise the success or failure.

RSA

When timing attacks were first published, targeting RSA, a masking countermeasure was deployed in most implementations: a random mask was multiplied before the exponentiation, and the corresponsing unmasking value was multiplied after the exponentiation. As has been noted above, whether the masking is a complete fix is not proven; and it requires a source of randomness, which can be a tough requirement.

BearSSL, instead, uses a fully constant-time implementation of RSA. It provides its own constant-time implementation of big integers. The internal representation of an integer includes an "announced bit length"; any integer value which is modulo a value p will use as "announced bit length" the bit length of p, thereby masking its actual bit length as an integer.

No conditional jump depends on the contents of an integer, only on its announced bit length. This means that in some occasions, operations must be done twice. For instance, after a Montgomery multiplication, the result may exceed the modulus and require an extra subtraction. The corresponding code (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_montmul.c;h=b953aa3f42d514da680b825b826cb432785e8b74;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l92) handles that subtraction like this:

```
br_i31_sub(d, m, NEQ(dh, 0) | NOT(br_i31_sub(d, m, 0)));
```

The "inner" call to `br_i31_sub()` computes the carry resulting from the subtraction, but does not actually store the subtraction result (that's what its third parameter controls). The "outer" call to `br_i31_sub()` will perform the same operations, and store the result if and only if its control parameter is 1.

The modular exponentiation uses a square-and-multiply algorithm. The multiplication is always computed; its value is then used if and only if the corresponding exponent bit is non-zero. A constant-time conditional copy (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_modpow.c;h=4ef3f5d5ac871a3086a0b76f9b25540de58765b1;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l61) is used.

The "i15" implementation uses a slightly more complex mechanism in that it will use a larger window when possible, i.e. as long as it fits within our stack buffers (BearSSL tries to keep stack usage within less than 4 kB, and still support 4096-bit RSA). The `br_i15_modpow_opt()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i15_modpow2.c;h=37073a425ac41b6c2caed27b1288f34886a53372;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l27) function precomputes up to 32 window values, in order to handle exponent bits by groups of five. This of course requires a constant-time lookup.

The RSA implementation may thus leak the lengths of the prime factors, but not their contents. Prime factor lengths are not secret; in practice, it is a fair bet that in most 2048-bit RSA keys, the two prime factors will have length 1024 bits exactly.

Current RSA implementations are still relatively poor in performance and should be enhanced with extra optimisations, possibly using vector opcodes or assembly when available.

Elliptic Curves

BearSSL includes several implementations of elliptic curves. Some use the same generic big integer functions as RSA ("i15" and "i31" code), and thus inherit their constant-time characteristics; other include specialised code which is made faster by exploiting the special format of the involved field modulus.

Some points are worth mentioning, for the implementations of NIST curves:

- For the NIST curves, input points are validated upon entry. Since all supported curves have a prime order, validation is simple: it suffices to check the curve equation with the provided coordinates. But even if the point is invalid (in a non-obvious way, i.e. its encoded length is correct), computations are still performed. The point validity is kept as a flag.
- Projective Jacobian coordinates are used: a curve point is represented by a triplet (X,Y,Z), the two actual coordinates being X/Z^2 and Y/Z^3. The "point at infinity" is represented by a triplet where Z is zero.

  Note that comparing two points for equality can be expensive since they do not necessarily have the same Z, so this entails field multiplications. Testing whether a point is the point at infinity, however, is cheap.
- The generic point doubling equations happen to work for all valid points, including the "point at infinity".
- The generic point addition equations do not properly handle two cases: when one of the points is the point at infinity (but not the other), and when both operands are the same point (in which case this should be a doubling).

  Fortunately, since the curve order is prime and the point multiplier can be guaranteed to be non-zero and lower than the curve order, it can be proven that the last occurrence (adding two points together and finding out they are identical) cannot happen in our double-and-add algorithm (with its 2-bit window). The case with the point at infinity is handled with a dedicated flag.
- Point doubling requires 8 field multiplications while point addition needs 16. To perform less additions, we use a 2-bit window, so there is a single addition every two doublings. Constant-time conditional lookup is done to select the right value. This can be observed in the code (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/ec/ec_prime_i31.c;h=ce4d4abdb88dcdfd9d0bb36fceaa0ff304bcacb8;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l549).
- ECDSA signature verification entails computing aG+bQ where a and b are two integers (modulo the curve order), G is the conventional generator, and Q is the public key. Classic implementations mutualise the doublings in the two double-and-add instances; however, this implies a larger table for window optimisation: if using 2-bit windows, then the aggregate table must have all combinations of G and Q with multipliers up to 3, so we

are in for at least 13 extra values (apart from G and Q themselves). Each such point uses 216 bytes (three coordinates large enough for the P-521 curve, over 31-bit words with an extra "bit length" word) so such a window would use up almost 3 kB of stack space. We cannot afford that within BearSSL goals.

Alternatively we could use 1-bit windows, but then the savings would be slight: for each multiplier bit, we would compute one doubling and one addition, but since we can no longer guarantee that none of these additions is a doubling-in-disguise, then we must throw in another doubling "just in case" (constant-time operation forbids conditional execution, so the doubling must always be done). The end result is a bigger code, more complex but not faster than simply computing the two point multiplications separately.

The final addition must still handle the special cases, so it also entails a doubling and some conditional copying; see the implementation (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/ec/ec_prime_i31.c;h=ce4d4abdb88dcdfd9d0bb36fceaa0ff304bcacb8;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l756).

- The specialised implementations for curve P-256 include some "fixed-point" optimisations, by which precomputed multiples of the conventional generator (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/ec/ec_p256_m31.c;h=0631a135f7d1a0a3b0e1d39e34c967fd884341d2;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l1138) are used to save on point additions. A constant-time lookup (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/ec/ec_p256_m31.c;h=0631a135f7d1a0a3b0e1d39e34c967fd884341d2;hb=5f045c759957fdff8c85716e6af99e10901fdac0#l1237) is used to avoid leakage.

Curve25519 allows for much simpler code (for instance, there is no need for validation, since all sequences of 32 bytes are by definition valid), and the "natural" Montgomery ladder is constant-time.

Future Developments

Future work in BearSSL, regarding constant-time implementations, will focus on the following:

- Architecture-specific AES implementations, especially using the dedicated AES instructions on architectures that offer them, or the vector instructions such as SSSE3 (following the techniques described by Michael Hamburg (https://shiftleft.org/papers/vector_aes/vector_aes.pdf)). In particular, a fast, non-parallel constant-time implementation of AES would benefit CBC encryption.
- Using Cantor-Kaltofen multiplication algorithm for a constant-time GHASH with a mixed bitslice strategy.
- Better big integer code, in particular using Karatsuba decomposition for moduli small enough that the extra stack space can be afforded (so basically for elliptic curves, perhaps not for RSA).
- Benchmarks and optimisations on the Cortex M0 or M0+, where the fast multiplication but lack of opcode yielding the high 32 bits is likely to challenge our assumptions.

---

1. Except the 16-bit `char` type.↩
2. For a more advanced experience, try writing a compiler. This is very pedagogical.↩
3. Mapping source bits into their proper emplacements within internal variables is akin to matrix transposition, i.e. mirroring along the diagonal, hence the name "orthogonal code".↩
4. For instance, the Cortex M0 and M0+ both exist in two versions, one with a 32-cycle multiplier, the other with a 1-cycle multiplier; of course, the 1-cycle multiplier is largely preferable for performance, but even the 32-cycle multiplier is constant-time.↩
5. Constant-time execution is also good for performance in that case, because carry propagation with conditional jumps would likely induce branch stalls.↩
6. It would have helped a bit if the padding had been located between plaintext and MAC value, not after the MAC value. But, of course, an encrypt-then-MAC setup would have been even better.↩

(https://twitter.com/BearSSLnews)

Please report any problem with this web site to: <pornin@bolet.org> (mailto:pornin@bolet.org?subject=BearSSL web site)