ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluating Constant-Time Languages and Compilers

A direct comparison of FaCT and Jasmin

Bachelor Thesis

B. Kaufmann

May 21, 2022

Advisors: Prof. Dr. Kenny Paterson, Jan Gilcher

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Timing side-channels have been a constant issue in implementations of cryptographic algorithms since their discovery in the mid 90's by Paul Kocher. To prevent them a programming paradigm usually called constant-time programming emerged, where implementors manually take care that timing behavior of the implementation is independent of any secrets. In recent years several different tools and approaches have been developed to aid programmers in this challenging task. Among these, the approaches using domain specific languages (DSL) and specialized compilers and toolchains seem the most promising solution. Existing work has shied away from directly comparing itself to its competition. This thesis aims to create a basic framework for comparing such tools. Accordingly the FaCT and Jasmin toolchains are juxtaposed against the libsodium library in metrics of performance and storage. The algorithmic assortment of curve25519, sha2 and ed25519 has been chosen to effectively compare FaCT and Jasmin to the libsodium baseline. By combining the results of our benchmark with our experience in coding both FaCT and Jasmin, we conclude Jasmin offers a completly verified constant-time toolchain at the cost of performance and a limited scope of application. For FaCT we find reasonable performance and accessibility at the expense of verified constant-time preservation only upto the LLVM IR.

Contents

Contents **iii**

1 Introduction **1**
1.1 Domain Specific Languages 2
1.2 Libsodium 2

2 Translation **5**
2.1 Fundamental Translations 5
2.2 FaCT Specific Challenges 7
2.2.1 One Dimensional Arrays of a Base Type Only 7
2.2.2 64-bit Literals and Static Arrays 8
2.2.3 Array Accesses with Run-Time Values 10
2.3 Jasmin Specific Challenges 11
2.3.1 Compile-Time Values Only 11
2.3.2 For-Loops Are Unrolled 12
2.3.3 Limit on Input Parameters and Static Variables 13
2.4 Usability Report 14
2.4.1 FaCT 15
2.4.2 Jasmin 15

3 Benchmarking **17**
3.1 Platform 17
3.2 Performance 18
3.3 Storage 19
3.4 Top-Level Functions 19
3.5 Libsodium Compilation 20
3.6 Code Examples 20

4 Results **23**
4.1 Performance 23

4.1.1 Ed25519 24
4.1.2 Sha2 26
4.1.3 Curve25519 29
4.1.4 Synopsis 31
4.2 Storage 32

5 Conclusion **37**

A Appendix **40**
A.1 Valgrind Massif Plots 41
A.1.1 Ed25519 41
A.1.2 Sha2 44
A.1.3 Curve25519 47
A.2 Benchmark of Jasmin's Sha2 50

Bibliography **53**

Chapter 1

Introduction

Security in a Computer Science perspective is as important as ever. With the continued digitalization and interconnection through the internet of every imaginable aspect in our lives security algorithms will only gain in importance. Establishing secure connections and guarding secrets, private networks, and private data should be a high priority for anyone with access to the modern internet, man or machine. While of course computational-security is necessary, it has been shown there are real vulnerabilities in practical implementations of computational-secure ciphers. Timing side-channels attacks can pose a real security threat as documented by Daniel J. Bernstein [3]. To prevent such attacks the notion of constant-time programming has emerged, where programmers manually enforce timing independence with respect to any secrets. So far this has been achieved with the use of a set of rules to transform vulnerable code fragments (e.g. if-statements depending on secrets, memory access patterns etc.) into code which executes the same instructions on assembly level irrespective of input.

Since programmers are only human and even the most proficient expert on any topic is still susceptible to making errors, there has been a drive to develop tools for aiding constant-time programming. Dynamic analysis, static analysis, code verification and domain specific languages (DSL) are some of the distinctive approaches these tools have taken to help the programmer. DSLs often incorporate several of the mentioned techniques into their toolchain. Additionally, since timing behaviour is outside of the specification for general purpose programming languages, compilers can even introduce timing-issues in otherwise seemingly constant-time code. Therefore DSLs with their specialized compilers, which guarantee constant-time preservation, seem the most promising.

1.1 Domain Specific Languages

Two promising tools are FaCT [6] and Jasmin [1]. Both use a DSL accompanied by specialized compilers which in turn produce intermediary files to be used in further compilation steps. FaCT is specifically intended to be used with the C programming language and produces an object file on compilation. It can further produce a header file on request to efficiently include exported functions from the FaCT source code in your C code. Jasmin on the other hand simply compiles to an assembly file capable of being processed further by any general purpose compiler.

The FaCT language is purposefully designed to be as C-like as possible. A choice which goes hand in hand with the intended use in conjunction with C. Further it lowers the hurdle for newcomers already familiar with C and can benefit from C's core principles which have withstood the test of time remarkably well. There are several crucial differences though. Firstly FaCT's type system includes a secrecy label. Each declared variable has either the secret or public label assigned to it by the programmer. This does include function parameters and return types. Secondly FaCT does not allow the use of pointers, they are a FaCT internal construct only. Although FaCT allows one layer of referencing through the ref primitve. To compensate for the absence of pointers FaCT offers the ability to operate on slices of an existing array with the *view* primitive.

Jasmin's design meanwhile is far closer to an assembly language as it takes direct inspiration from qhasm. Jasmin does have a layer of high-level primitives such as *while*- and for-loops, if-statements and variables. Variables are either registers, stack allocated or inlined. Inlined variables are resolved at compilation. Importantly address offsets for memory accesses have to be inlined variables or literals. As a consequence address offsets can not be run-time resolved values. Whereas *while*-loops are implemented through jump instructions, for-loops are unrolled in the compilation process by Jasmin and therefore only work over inlined variables.

1.2 Libsodium

To compare FaCT and Jasmin in a benchmark several cryptographic algorithms were chosen. We decided to use a cryptographic library which would source all algorithms. The reasoning behind this decision are as follows: A popular modern library would provide algorithms which are up-to-date and used in practice. Further a modern cryptographic library offers a certain standard of quality across all implementations both in terms of performance and the absence of bugs.

In the end the libsodium library was chosen. Libsodium has a broad offering of algorithms, is relative popularity for real world applications and has a research related background in the NaCl library [7].

As for the specific algorithms chosen to benchmark the two tools and the original libsodium library implementations, we first have the scalar multiplication of the elliptical group provided by curve25519 from now on called simply curve25519. Secondly the sha2-512 hash algorithm from now on referenced as sha2 and lastly the sign function of the signature scheme ed25519 which is based upon the curve25519 elliptical group from now on mentioned by ed25519 only.

Curve25519 and ed25519 have found widespread adoption in the current internet landscape. Both algorithms are in the public domain and offered improved performance at the time of their introduction by Bernstein and co [4] [5]. In part because of those two advantages these algorithms have become a standard choice for the Diffie-Hellman key-exchange and signature ciphers respectively. As such it is an easy and logical choice to use these algorithms as a baseline in our benchmark.

Hash algorithms are an important part of crypthography and an obvious addition. While sha2 is not the most modern hash algortihm, it is still widely used in practice and is specifically used by libsodium in its implementation of ed25519. As such sha2 is a natural choice for the hash in the baseline.

Originally there was a further desire to include an hmac in the baseline, but unfortunately, due to time constraints and difficulties with Jasmin, the implementation of libsodium's sha2-based hmac was only completed in FaCT. Thus it has been left out in the final benchmark. Similarly at the beginning an AES implementation was discussed but due to time-constraints and the absence of a non-intrinsic implementation in libsodium it was scrapped as well.

Now libsodium has three different implementations of curve25519 and the underlying fundamental curve25519 functions. Ostensibly relevant for curve25519 itself but also for ed25519 which is built upon on curve25519 core functions like addition, subtraction, multiplication, toBytes etc. A closer look reveals there are two versions of the *ref10* implementation and a singular version of the *sandy2x* implementation. As *sandy2x* is built upon assembly vector instructions it was eliminated from contention early on. A non-specialized implementation was desired for the baseline as to not create a situation where a selected tool lacks the expressiveness for a translation. The two *ref10* variants differ in the use of 128-bit arithmetic operations. Again to pursue a general-purpose reference implementation the version without 128-bit operations was chosen. With this we have determined our final baseline.

Chapter 2

Translation

When translating these algorithms to both FaCT and Jasmin, we were faced by a respectively unique set of challenges. This chapter will explore these challenges and the solutions we found. Take note these solutions might not be optimal but merely our best effort. We will highlight some of the encountered quandaries on a few examples.

2.1 Fundamental Translations

FaCT's C-like character made translation in many aspects straightforward. Operators, casts, arrays and assignment operators are in essence identical. Further many aspects of C have a similar equivalent in FaCT i.e. *uint64_t* vs *uint64*, *constant* vs *mut*, *structs*, function definitions etc.

```
int32_t q;
int32_t carry0, carry1, carry2, carry3, carry4,
        carry5, carry6, carry7, carry8, carry9;

q = (19 * h9 + ((uint32_t) 1L << 24)) >> 25;
q = (h0 + q) >> 26;
q = (h1 + q) >> 25;
q = (h2 + q) >> 26;
q = (h3 + q) >> 25;
q = (h4 + q) >> 26;
q = (h5 + q) >> 25;
q = (h6 + q) >> 26;
q = (h7 + q) >> 25;
q = (h8 + q) >> 26;
q = (h9 + q) >> 25;
```

Figure 2.1: C source code

```
secret mut int32 q = int32(
   (19 * uint32(h9) + (uint32(1) << 24))
    >> 25
);

q = (h0 + q) >> 26;
q = (h1 + q) >> 25;
q = (h2 + q) >> 26;
q = (h3 + q) >> 25;
q = (h4 + q) >> 26;
q = (h5 + q) >> 25;
q = (h6 + q) >> 26;
q = (h7 + q) >> 25;
q = (h8 + q) >> 26;
q = (h9 + q) >> 25;
```

Figure 2.2: Translated FaCT code

Unlike C in FaCT a variable has to be initialized on declaration. Libsodium's implementation features several local variables which are only declared but not initialized. Thus in our FaCT translation all these variables are initialized to either a value assigned later in the source code or, if not possible, to zero.

We can see an outcome of this kind of transformation in figure 2.1 and 2.2. As an immediate example contrast the first initialization of the variable q. Notice moreover FaCT does not support implicit casting. However on the whole the translation is quite straight forward here.

```
tmp32 = h9;
tmp32 *= 19;
tmp32 += shift24;
tmp32 >>s= 25;


tmp32 += h0;
tmp32 >>s= 26;
tmp32 += h1;
tmp32 >>s= 25;
tmp32 += h2;
tmp32 >>s= 26;
tmp32 += h3;
tmp32 >>s= 25;
tmp32 += h4;
tmp32 >>s= 26;
tmp32 += h5;
tmp32 >>s= 25;
tmp32 += h6;
tmp32 >>s= 26;
tmp32 += h7;
tmp32 >>s= 25;
tmp32 += h8;
tmp32 >>s= 26;
tmp32 += h9;
tmp32 >>s= 25;
```

Figure 2.3: Translated Jasmin code of figure 2.1 without the variable declarations

Compare and contrast this to the Jasmin code in figure 2.3 seemingly a whole different animal. Apparent is the closeness to assembly since ”complex” mathematical expressions such as the first assignment of the tmp32 variable, the Jasmin equivalent to the q variable in the C code, needs to be partitioned into their basic instructions line by line. Further the programmer does need to handle the explicit loading and storing of stack and memory storage in and out of registers. While both C and FaCT do not need to return any variables for their void type functions, Jasmin functions of the same functionality need to explicitly return changed variables even if they are input parameters.

```
stack u32 h0;
stack u32 h1;
stack u32 h2;
stack u32 h3;
stack u32 h4;
stack u32 h5;
stack u32 h6;
stack u32 h7;
stack u32 h8;
stack u32 h9;
stack u32 q;
reg u32 tmp32;
reg u32 tmp2_32;

inline u32 shift24;
inline u32 shift25;
inline u32 shift26;
shift24 = 1 << 24;
shift25 = 1 << 25;
shift26 = 1 << 26;
```

Figure 2.4: Jasmin variable declarations for the function seen in figure 2.1

Jasmin takes a distinctive approach to variable declarations compared to both C and FaCT. Inline with the assembly-like nature of Jasmin all local variables need to be declared and only declared at the beginning of a function. In figure 2.4 we have the complete variable declarations of the same function all the other figures in this section are taken from.

Two register variables, *tmp32* and *tmp2_32*, are declared to actually execute arithmetic instructions as seen in figure 2.4. It lies in the programmer's responsibility to avoid overloading register variables, therefore we tried to use minimal register variables without introducing unnecessary load and store operations.

2.2 FaCT Specific Challenges

2.2.1 One Dimensional Arrays of a Base Type Only

Multidimensional arrays are not available out of the box in FaCT. Fortunately multidimensional arrays in C are only logical. Assuming we have an array *arr* declared as follows `int arr[N][M];` with $N, M \in \mathbb{N}$, then C does nothing more than create an array of length $N * M$. Likewise C simply translates `arr[i][j]` to `arr[i*N+j]` when accessing array elements. By using the same technique in FaCT, one is able to express multidimensional arrays.

Unfortunately there is a further problem because arrays of structs and structs with fields of struct type are simply not supported by FaCT. The latter case only appeared in the hmac implentation and was easily bypassed by just disassembling the struct. Meaning, instead of passing the struct to the function as a parameter, each field was passed as its own function parameter.

```
typedef struct {
        fe25519 YplusX;
        fe25519 YminusX;
        fe25519 Z;
        fe25519 T2d;
} ge25519_cached;
```

```
typedef int32_t fe25519[10];
```

Figure 2.5: Defintion of ge25519_cached in C

In libsodium's ed25519 implementation there are structs which consist of three or four identical fields. The fields of these structs are all arrays of type *int32_t* and have length 10. For example we look at ge25519_cached in Figure 2.5.

```
static void
ge25519_cmov8_cached(
        ge25519_cached *t,
        const ge25519_cached cached[8],
        const signed char b
)
```

(a) C code

```
void
ge25519A_cmov8_cached(
        mut ge25519_cached t,
        secret int32[320] cached,
        secret int8 b
)
```

(b) FaCT code

Figure 2.6: Defintion of ge25519_cmov8_cached

As seen in figure 2.6 (a) ge25519_cmov8_cached has an array of length eight with type ge25519_cached as an input parameter. To circumvent the lack of arrays of structs in FaCT we decompose the whole framework down to the base type. Such an array basically consists of $4 * 10 * 8 = 320$ entries with type *int32_t*. Now in FaCT we instead have an array of type *int32* with length 320, as seen in figure 2.6 (b). If we access the array logically we just read out the 40 relevant entries into a placeholder of the right struct type.

2.2.2 64-bit Literals and Static Arrays

On more than one occasion the C source code features static arrays. Some of which are initialized with 64-bit literals. This presents a problem on two fronts. On one hand FaCT only recognizes literals of size up to 32 bits. On the other hand FaCT does not have anything similar to static or global variables in C.

```
krnd[0] =  uint64(0xd728ae22);
krnd[0] |= uint64(0x428a2f98) << 32;
```

Figure 2.7: Loading 64-bit Literals in FaCT

The literals are relatively easy to solve. First the lower 32 bits of the literal get initialized into the desired 64-bit variable and the upper 32 bits get initialized into a temporary variable which then gets left-shifted by 32. Finally we perform a bitwise XOR-operation with the temporary variable and the lower 32 bits of the original value. The resulting value of this operation is now equal to the full 64-bit literal. In Figure 2.7 the 64-bit literal $0x428a2f98d728ae22$, given in hexadecimal, gets loaded into array *krnd* at position 0 using the aforementioned method.

```
static const ge25519_precomp base[32][8] = { /* base[i][j] = (j+1)*256^i*B */
        #ifdef HAVE_TI_MODE
                # include "fe_51/base.h"
        #else
                # include "fe_25_5/base.h"
        #endif
};
```

Figure 2.8: Declaration of the base array in libsodium

A simple solution to static variables is to just use an equivalent function-local variable. Of course this solution incurs performance penalties as well as expanded memory usage. A finer solution would be to define and initialize these static variables in the header file and then pass them to the necessary functions as an input parameter. This way we outsource the problem and in turn the 64-bit literals no longer pose a problem.
As a poignant example we have the *base* array in the C code. In figure 2.8 the declaration of the *base* array can be seen. As ge25519_precomp consists of 3 identical fields of type `int32_t[10]` we get an array of size $32 * 8 * 3 * 10 = 7680$ with type *int32_t*. An array of this size would obviously create heavy performance penalties if loaded new as local variable on every call. Therefore this array was declared in the FaCT header file as `static const uint32_t baseH[7680] = { /* content of array */};` and then passed to the top-level exported FaCT function as seen in figure 2.9.

```
/*public*/ int32_t _cryptoA_sign_ed25519_detached(
  /*secret*/ uint8_t sig[],
  /*public*/ uint64_t sig_len,
  /*secret*/ uint64_t * siglen_p,
  const /*secret*/ uint8_t m[],
  /*public*/ uint64_t m_len,
  /*public*/ uint64_t mlen,
  const /*secret*/ uint8_t sk[],
  /*public*/ uint64_t sk_len,
  /*public*/ int32_t prehashed) {

        return _cryptoC_sign_ed25519_detached(sig, sig_len, siglen_p,
                                              m, m_len, mlen, sk,
                                              sk_len, prehashed, baseH);
}
```

Figure 2.9: Using the FaCT header file to pass baseH

Consequently both techniques were used when deemed appropriate. Local variables where used for small and infrequently used static variables when performance overhead would remain miniscule and adding an additional input parameter to all dependant functions seemed superfluous.

To the contrary if the static variable was sizeable or used frequently then ostensively a local variable translation would incur significant performance penalties. Therefore in such a situation the variable was declared in the FaCT header file and passed to the FaCT source code as an input parameter to all dependant functions.

2.2.3 Array Accesses with Run-Time Values

For FaCT to compile successfully, its public safety checker must proof the impossibility of illegal accesses to secret variables. The public safety checker can resolve most of the accesses in the code automatically except for non-trivial array accesses inside a loop or input parameter dependent memory accesses. Hence FaCT provides the *assume* primitive which allows the programmer to pass loop invariants to the safety checker. With these loop invariants the public safety checker should be able to successfully complete the necessary safety proofs. The programmer has to ensure these invariants are valid as to not compromise public safety in their program.

```
assume( 240 <= len base - uint64(240*pos));
assume( uint64(240*pos) <= len base);

if((240 <= 7680 - 240 *pos) && (240*pos < 7680)) {
        ge25519A_cmov8(t,view(base, 240*pos,240) ,b);
}
```

Figure 2.10: Excerpt from ge25519_cmov8_base

In Figure 2.10 we have an example of a situation where the public safety checker has no chance of establishing a safety proof without the help of the programmer. The *base* array seen in figure 2.10 has already been mentioned in section 2.2.2. We know it has length 7680 and for FaCT to compile the public safety checker must proof that the *view* primitive does not introduce out-of-bounds accesses.

First we recall the use of the *view* primitive to get slices of an array. *view* takes three inputs, the first a reference to the array, the second the offset from the beginning of the array to the start of the slice and the third being slice length. We see *view* is dependent on the *pos* variable. Since *pos* is an input parameter FaCT cannot construct a valid proof on its own. As the programmer we know the ge25519_cmov8_base function is an internal function exclusively and receives only values from 0 to 7 for the *pos* variable from its caller.

To pass on this information to FaCT two calls of the *assume* primitive are used. The first *assume* expression guarantees the end of the slice is in range $[239, 7679]$ and the second one guarantees the beginning of the slice is in range $[0, 7439]$. Combined they ensure the slice is completely inside the bounds of the base array. With this new information the public safety checker is capable of completing the necessitated proofs for FaCT to compile.

2.3 Jasmin Specific Challenges

2.3.1 Compile-Time Values Only

Jasmin restricts offset values in memory accesses and *for*-loop bounds to inlined variables only. In other words such values must be known at compilation. This created problems in multiple ways. Because of the modular nature of libsodium, memory accesses and especially *for*-loop bounds are implemented sporadically as run-time resolved variables in the source code.

Most frequently sha2 featured dependencies on run-time resolved values in loop bounds. A significant margin of these appear because sha2 is designed to handle inputs of variable length. Thus resolving this problem with *for*-loops was inherently impossible. Jasmin does provide a *while*-loop primitive which operates over run-time variables and hence is an obvious substitute.

```
if (inlen < 128 - r) {
    for (i = 0; i < inlen; i++) {
        state->buf[r + i] = in[i];
    }
    return 0;
}
for (i = 0; i < 128 - r; i++) {
    state->buf[r + i] = in[i];
}
/* code continues */
```

(a) C code

```
tmp = 128;
tmp -= r;
if(inlen < tmp) {
    for i = 0 to 128 {
        if(inlen > i) {
            tmp8 = (u8) [in+i];
            tmp = r;
            tmp += i;
            sBuf[(int) tmp] = tmp8;
        }
        }
} else {
        for i = 0 to 128 {
        if(tmp > i) {
            tmp8 = (u8) [in+i];
            tmp2_64 = r;
            tmp2_64 += i;
            sBuf[(int) tmp2_64] = tmp8;
        }
    }
/* code continues */
```

(b) Jasmin code

Figure 2.11: For-loops inside function sha512_update

In the end a slightly different solution was implemented. If there was a logical upper bound on the run-time resolved loop bounds then in Jasmin the for-loop would have the aforementioned logical upper bound as its loop bound. if-statements were then used to simulate the original function in the source code.

A visualization of such a transformation can be seen in figure 2.11. Before the start of the code excerpt there was a modulo operation on the variable r with divisor 128. Therefore we know 128 is an upper bound on the number of iterations for both loops. The if-statements ensure their contents are only executed when the loop index is in the range of the original loop iteration values. Naturally the if-bodies contain corresponding code to the original loop body. The upper bounds on loop bounds for the affected for-loops were negligible, ergo a transformation of this kind would not incur significant performance penalties. While not used in this particular instance, $while$-loop transformations were used for a problem discussed in the following section 2.3.2.

On certain occasions the whole issue could be circumvented. In Jasmin not all functions are compiled down to assembly code, only exported functions are. Non-exported functions are inlined when called in other functions. For internal functions with problematic loop bounds a solution could be found if the critical for-loop bounds were dependant on an input parameter and the callee always assigned a literal to that parameter. Then these input parameters are transformed to an inlined variable and the internal function would also be defined as inlined. The only remaining hurdle was functional testing of the inlined functions which was done with hard coded values on the inlined parameters in auxiliary exported functions.

2.3.2 For-Loops Are Unrolled

As a result of compile-time resolved loop-bounds Jasmin is able to unroll for-loops in the compilation process. While this technique does ease the constant-time verification of programs it did create problems for us during the translation process. In particular in situations where loops featured calls to non-trivial functions.

The for-loop unrolling alone does no present a problem. However Jasmin only has stack or registers variables at run-time. Moreover the programmer is responsible for register allocation in Jasmin and therefore needs to ensure the prevention of overlapping register allocations. Since the amount of local variables needed far exceeded the number of available registers, stack variables had to be used frequently as local storage. Now combining the loop unrolling with numerous local stack variables inside the called functions lead to an stack overflow error when trying to compile higher level functions.

```
for (i = 1; i < 64; i += 2) {
        ge25519_cmov8_base(&t, i / 2, e[i]);
        ge25519_madd(&r, h, &t);
        ge25519_p1p1_to_p3(h, &r);
}
```

Figure 2.12: For-loop found in ge25519_scalarmult_base

In Figure 2.12 a *for*-loop found in the ed25519 source code can be seen. All called functions, especially ge25519_cmov8_base, are non-trivial and feature plenty of local stack variables. Hence unrolling this loop creates 32 times as many stack variables. As a result, when translating this exact loop as a *for*-loop to Jasmin, stack overflow errors are encountered when compiling.

```
j = 1;
while(j < 64) {
    tmp8 = e[(int) j];
    tmp8_stack = tmp8;
    tmp64 = j;
    tmp64 >>= 1;
    t_yplusx, t_yminusx, t_xy2d
      = ge25519A_cmov8_base(t_yplusx, t_yminusx, t_xy2d, tmp64 ,tmp8_stack , srcBase);
    r_X, r_Y, r_Z, r_T
      = ge25519A_madd(r_X, r_Y, r_Z, r_T, h_X, h_Y, h_Z, h_T, t_yplusx, t_yminusx, t_xy2d);
    h_X, h_Y, h_Z, h_T
      = ge25519A_p1p1_to_p3(h_X, h_Y, h_Z, h_T, r_X, r_Y, r_Z, r_T);
    j += 2;
}
```

Figure 2.13: For-loop of Fig. 2.12 translated to Jasmin

A solution can be found in replacing the *for*-loops with equivalent *while*-loops. Since *while*-loops are not unrolled a function call is compiled only once instead of 32 times as seen in the example. In figure 2.13 such a translation to a *while*-loop can be seen for the code of figure 2.12. Transformations of this type had to be done a couple of times to successfully compile ed25519 without stack overflow errors.

2.3.3 Limit on Input Parameters and Static Variables

Jasmin only allows registers as input parameters for exported functions. As we know there are at most 6 registers to be used as parameters on function calls in x86-64. Thus as a consequence Jasmin only allows for 6 input parameters in exported functions. Similar to FaCT large and/or frequently used static arrays are allocated in C an the passed to Jasmin's exported functions. The same is done for some local storage space. Unfortunately this did lead to more than six theoretical function parameters in some cases.

```
uint8_t storage[31594];
memcpy(storage + 234, KrndA, 840);
memcpy(storage + 874 , baseA, 30720);
memcpy(storage + 192, DOM2PREFIX, 34);
```

Figure 2.14: Allocation of local storage and static arrays in C to be passed to Jasmin

To solve this issue all static arrays and the mentioned local storage spaces are combined into one large array in C and then passed to the Jasmin function. This reduced the needed input parameters to 6. To read out the desired "fields" of this combined array its address, which is stored on the stack, can be used in combination with the known offsets to set a register variable with the address to the desired variable.

```
tmp2_64 = (u64) [srcLocalStorage + 226];
tmp64 = srcLocalStorage;
tmp64 += 192;
tmp3_64 = srcLocalStorage;
tmp3_64 += 234;
sState, sCount, sBuf =
    _cryptoB_sign_ed25519_ref10_hinit(sState, sCount, sBuf, tmp64, tmp2_64, tmp3_64);
```

Figure 2.15: Access to C allocated local storage and static arrays through offsets in Jasmin

A practical application of this process can be seen in figure 2.14 where the *storage* array is allocated and partly initialized with static arrays in C. In figure 2.15 we showcase the procedure to access static arrays and local storage inside *storage* through known offsets. Note in this Jasmin code the address to the aforementioned *storage* array has been previously stored in the *srcLocalStorage* variable.

2.4 Usability Report

Usability is not on top of priority charts for researchers and developers of these tools, for obvious and understandable reasons. Nonetheless for the adoption of FaCT and Jasmin in practice, by developers of cryptographic libraries, factors like usability, project integration and platform availability are relevant.

This report is written from the perspective of a complete newcomer to both FaCT and Jasmin. Further the bulk of the work was in porting C code to the respective languages. While this task provides a useful insight it can not be compared one-to-one with the experience of writing complete new cryptographic algorithms.

2.4.1 FaCT

The creators of FaCT included a usability study in their paper [6]. Although a rather small study it found FaCT to be approximately as intuitive to understand as C. Likewise coding correct constant-time code in FaCT was achieved by a slightly higher margin of participants compared to coding in C. In general we would echo the findings of the report. After a period of acclimatization programing in FaCT is very similar to C. Especially once one understands how to use slices with the *view* and *assume* primitives instead of pointers when needed.

FaCT's integration with C is nearly seamless with the exception of structs. The structs inside the FaCT generated header filer are declared without any fields. To utilize these structs in the C code (e.g. for passing input parameters to the FaCT functions) one has to manually complete the struct definition in the header.

There are some drawbacks. Resources for learning FaCT are scarce which is expected. The paper itself does a relative poor job of bringing across the language to a programmer. While Jasmin's paper likewise did not divulge to much information on utilization, Jasmin does provide some supplementary material unlike FaCT. Starting to learn FaCT is very much a trial and error affair even in comparison to Jasmin.

Moreover there are some restrictions which hinder ease-of-use. As discussed it is not possible to have struct fields of struct type or an array of structs. While this does not technically restrict expressiveness it does create the need for time-consuming work-arounds to achieve the modularity of C code.

Overall FaCT is quite useable after getting over the introductory hurdle. The compilation error messages are basic but workable, compilation times remain low with increasing code size, modularity is achievable and language primitives are on the whole intuitive.

2.4.2 Jasmin

Jasmin's design is somewhere between an assembly- and high-level programming language. As a consequence of having far less experience in programming assembler(e.g. x86-64, MIPS) than high-level languages (e.g. C, C++, Java, Haskell etc.) learning Jasmin proved to be a bit harder. On the other hand Jasmin does have more resources to learn. The github repository has a documentation which has several short chapters explaining different topics (e.g. Arrays, Global Variables etc.). In addition Jasmin provides small programs meant to showcase the functionality of Jasmin in different ways. Jasmin's integration into C is good but not as seamless as FaCT since headers with function declarations need to be written manually for effective use of the compiled code.

Unsurprisingly as a result of Jasmin's closeness to assembly it is more cumbersome to implement high-level language algorithms in Jasmin compared to a high-level language like FaCT. A significant problem arose however. Jasmin's compilation time is glacial compared to FaCT and even became a significant problem when trying to implement ed25519 and the hmac. Both algorithms rely heavily on sha2 and its subfunction init, update, final. On top of that ed25519 utilizes the curve25519/ed25519 subfunctions reduce, mul, scalarmult_base etc. All these functions in of itself had compilation times of 5 to 25 minutes. When combining them for the sign function of ed25519 compilation times were in the range of 4-6 hours if not longer.

Now this is after the fact unsurprising as Jasmin seems to be designed to be used only to compile either smaller subfunctions of high-level cryptographic algorithms or specifically lightweight algorithms. Regardless this makes Jasmin suboptimal for coding complete sophisticated and modular algorithms. Even coding parts of the libsodium library seem unimaginable. The uncompleted hmac threw further a wrench into benchmarking plans. And more importantly it raises a question. If Jasmin verified code only represents a portion of a cryptographic suite, how significant is the resulting verification gap for constant-time guarantees? Assuming good constant-time practice in surrounding code this might not be substantial problem. However the goal of these new tools is to abandon reliance on good practice and instead to depend on constant-time verification.

Chapter 3

Benchmarking

The Goal of this thesis is to compare performance of different DSLs ergo the focus of this benchmark is on performance but it is not the only aspect.

The benchmark is written in C. Two contributing factors for this decision are FaCT's intended use in conjuction with C, plus the fact that libsodium is implemented in C. For Jasmin C-compilers are more than adept in linking assembler files. Altogether C is a logical choice.

3.1 Platform

An important aspect of every benchmark is to know the underlying hardware and software upon which the benchmark is run. We used a desktop computer with the following specifications:

CPU	Intel Xeon CPU E3-1231 v3 @ 3.40GHz × 4
GPU	NVIDIA GeForce GTX 1080 Ti
RAM	15.6 GiB @ Corsair Vengeance 4GB DDR3-1600 x 4
Main Storage	OCZ Vertex 460 SATA-III SSD 120 GB
OS	Ubuntu 20.04.4 LTS 64-bit

3.2 Performance

Performance can be measured in several ways but this paper focuses on the number of CPU clock cycles. While absolute time is an important metric, it is obviously not comparable across different platforms. The number of CPU clock cycles should have less variance across systems of the same architecture. In this case the architecture is x86-64 which of course in 2022 is as widespread as ever. Specifically a measure of cycles passed between the moment precisely before the function call and the moment just after the end of its execution is taken.

Inputs are randomly generated by an implementation of the Mersenne Twister [8]. Then we perform a cache warm up by invoking the function for ~2000 iterations on the to-be-tested variables. Finally to benchmark the function we use a method described by the intel paper "How to Benchmark Code Execution on Intel IA-32 and IA-64 Instruction Set Architectures" [10]. This method makes use of the *RDTSC* and *RDTSCP* instructions to read the timestamp register before and after the execution of the function. *RDTSC* and *RDTSCP* both read the clock cycle register. *RDTSCP* furthermore guarantees the complete termination of all code before its reading of the clock cycle register. *move* instructions are used to store timestamps into the appropriate registers. Additionally *CPUID* is used to create a barrier between the function call and the rest of the program. This barrier prevents out-of-order execution of instructions associated with high-level code before and after the two *CPUID* instructions. Due to the barrier all instructions executed by the program between the *RDTSC* and *RDTSCP* instruction correspond to the call of the benchmarked function.

This procedure is repeated for 16000 iterations. For each repetition the absolute difference of both measured timestamps is stored to ultimately compute the average and variance over all iterations. Both ed25519 and sha2 have variable input length, therefore their benchmarks have the option of passing the desired length in Bytes as an execution argument. If this option is not utilized the length is set to a standard value of 1024 Bytes. We decided to run the benchmarks for ed25519 and sha2 for a power of two selection of arguments. The selection contains all power of twos between 64 and 8192 i.e. the set $\{64, 128, 256, 512, 1024, 2048, 4096, 8192\}$.

For curve25519 we will simply output all measured clock cycles on a new line. On the other hand for both sha2 and ed25519 we will compute mean and variance, for the given input length, inside the benchmark. As output we then have mean and variance on the same line separated by a comma.

3.3 Storage

Storage space is another facet of benchmarking. Less relevant than performance for this thesis. Storage can be divided in storage used while running the program and the file size of the final binary. File size of binaries is straightforward. The bash shell is used in combination with the command *ls -l* to read out file sizes in Bytes.

The analysis of run-time storage requires more sophisticated tools. There are several bash commands such as *top*, *ps* and *pmap* which provide functionality to monitor processes including their memory usage. However these tools only capture snapshots when run in the bash shell. Luckily there are several tools available to profile memory usage of a program. Probably the most known tool is valgrind and its memory profiler massif [9]. Although massif standard functionality is to profile heap memory specifically, massif can also be used to monitor all memory used by the program. One has to pass appropriate flags to extend functionality. In this case concretely the flags *–stacks=yes* and *–time-unit=i* are supplied to enable the profiling of stack memory as well.

For this benchmark a slightly different code is used to run with valgrind. First compared to the performance benchmark inlined instructions and computation of mean and variance have been removed. The cache warm-up loop has been removed as well and the loop bound has been reduced to 500 iterations. In essence only the generation of the random input, the function call itself and the loop structure are left.

3.4 Top-Level Functions

For curve25519 the benchmark runs scalar multiplication on two random inputs which is implemented by the crypto_scalarmult_curve25519_ref10 function. For sha2 the benchmark tests the top-level function crypto_hash_sha512 which receives an input of random value and given size. For these algorithms this selection is straightforward since both have a clear top-level function which execute their intended singular task.

On the contrary ed25519 is a non-trivial suite of top-level functions which serve to fulfil several distinctive tasks i.e. verifying signatures, creating signatures and generating keypairs. The chosen task to benchmark is creating signatures. To start with verifying signature does not handle secrets when implemented correctly. Secondly generating keypairs is an interesting issue but not a unique problem to ed25519 and was not the intended rationale behind choosing ed25519 as a baseline. Moreover creating signatures is the heart of a signature algorithm and therefore for this thesis the most interesting part.

The libsodium implementation of ed25519 has four top-level signing functions, three of which are intended to be called by users of libsodium. The three user-accessible functions are crypto_sign_ed25519_detached, crypto_sign_ed25519 and crypto_sign_ed25519ph_final_create.
_crypto_sign_ed25519_detached is the internal top-level function which all three of the previous mentioned functions are based upon on. To avoid needlessly complicating the translation crypto_sign_ed25519_detached is benchmarked.

3.5 Libsodium Compilation

As discussed before libsodium provides three different implementations of curve25519 and its core functions. The computer this benchmark is run on does support 128-bit architecture and vectorized assembly instructions needed for the unsought implementations. When building the libsodium library through the standard process presented by the documentation the availability of these instructions are recognized and thus compiles cruve25519 and ed25519 using undesired implementations. Additionally libsodium uses pthreads on default settings.

Two of these problems can be dealt with through passing the *–without-pthreads* and *–disable-asm* flags when running configure. *–without-pthreads* is self-explanatory and *–disable-asm* prevents libsodium of using vectorized assembly instructions and consequently the *sandy*2*x* implementation. Now the remaining problem of 128-bit operators must be handled manually. To this end we modified the configure and configure.ac files, from the libsodium source code, by hand and in such a way that the Makefiles created by running ./configure indicate the unavailability of 128-bit instructions.

3.6 Code Examples

In this section we provide three figures to visualize the benchmark. Particularly we showcase code segments responsible for implementing the previously discussed processes.
First we look at the use of the Mersenne-Twister (MT) in figure 3.1. We generate a random 32-bit number through the MT supplied function. Subsequently we use the left shift operator to discard the later 24 bits and store the leading 8 bits into the message array *mO*. As described variations of this exact technique are used to generate all needed random inputs.

```
for(uint32_t j = 0; j < mlen0; ++j) {
        m0[j] = genrand_int32() >> 24;
}
```

Figure 3.1: Use of the Mersenne-Twister for generating random messages in the ed25519 benchmark

In figure 3.2 on the other hand the method for extracting the desired CPU clock cycles is pictured. The call to the sha2 top-level function is sandwiched between the assembly instructions. This technique as mentioned is copied one-to-one, except for the sha2 function call, from the Intel paper [10]. The result is then stored inside the *store* array. This *store* array is then used to calculate the mean and variance in the case of ed25519 or sha2. Conversely for curve25519 the *store* array is simply used to print out all measured CPU clock cycles. What is more the *Krnd* variable in the sha2 function call is a further example of an externally declared static array passed on to Jasmin for internal use.

```
asm volatile ("CPUID\n\t"
        "RDTSC\n\t"
        "mov %%edx, %0\n\t"
        "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
        "%rax", "%rbx", "%rcx", "%rdx");

cryptoB_hash_sha512(out0,in0,length,Krnd);

asm volatile("RDTSCP\n\t"
        "mov %%edx, %0\n\t"
        "mov %%eax, %1\n\t"
        "CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1)::
        "%rax", "%rbx", "%rcx", "%rdx");

start = ( ((uint64_t)cycles_high << 32) | cycles_low );
end = ( ((uint64_t)cycles_high1 << 32) | cycles_low1 );
store[i] = end - start;
```

Figure 3.2: Intel-based use of instructions for measuring cycles

Finally in figure 3.3 we have the complete code in charge of the cache warm-up. As one can discern the sha2 function is simply called 2000 times for randomly generated input. Importantly the same variables are used as input parameters in the warm-up procedure and during the definite measurement. The whole C file responsible for benchmarking the sha2 implementation in Jasmin has been provided in the appendix under section A.2. All three snippets seen in this section originate from said C file.

```
for(uint32_t i=0; i<2000; ++i) {
        uint64_t length = lengthIn;
        uint64_t lengthb = lengthIn;

        for(uint64_t j=0; j<length; ++j) {
                unsigned char t = genrand_int32() >>24;
                inO[j] = t;
                inA[j] = t;
        }

        cryptoB_hash_sha512(outA,inA,lengthb,Krnd);
}
```

Figure 3.3: Example of the whole cache warm-up process for Jasmin's sha2 implementation

Chapter 4

Results

Like the name of this chapter implies we will have a look at the results of our benchmark described in chapter 3. We will go over them in chapter 3's described order of importance i.e. performance, run-time storage and lastly binary size.

4.1 Performance

Since curve25519 produce a slightly different benchmark in comparison to ed25519 and sha2 we will have different visualization in its subsection. For the sections on ed25519 and sha2 both contain analogous charts. First to simply display the data gathered by the benchmark we have a lin-log plot with the input sizes on the logarithmic x-axis and mean clock cycles on the linear y-axis. Figures of this type are 4.1 and 4.4.

Next to show relative performance in comparison to the baseline, we have two identical lin-log plots, one for FaCT and Jasmin respectively. The two plots are combined into one figure. Identical to before input sizes are on the logarithmic x-axis and on the linear y-axis we have mean clock cycles given as a percentage of libsodium's mean clock cycles. Figures 4.2 and 4.5 are of this nature.

Lastly to visualize performance evolution with increasing input size, we have a further lin-log plot. On the linear y-axis the difference in mean clock cycles between current and preceding input length is given as a percentage. The usual first entry of the x-axis, 64, has been removed because it does not have a preceding input length. Apart from this change the x-axis is identical to the ones in prior plots. Figures 4.3 and 4.6 fall into the aforementioned category.

As indicated beforehand the section on curve25519 has distinct plots. Since the benchmark for curve25519 gives us all measured clock cycles we can display boxplots for these data sets. The boxplots indicate both mean and median and the box itself spans from the 25th percentile to the 75th percentile (denoted as Q_1 and Q_3 respectively). The whiskers are set to $Q_1 - 1.5 \cdot IQR$ and $Q_3 + 1.5 \cdot IQR$ for the lower and upper whisker respectively. IQR denotes the interquartile range defined as $IQR = Q_3 - Q_1$. To correspondingly compare relative performance against the libsodium baseline a table is used. Inside the table mean and median clock cycles are given as percentage of libsodiums counterpart for both FaCT and Jasmin.

4.1.1 Ed25519

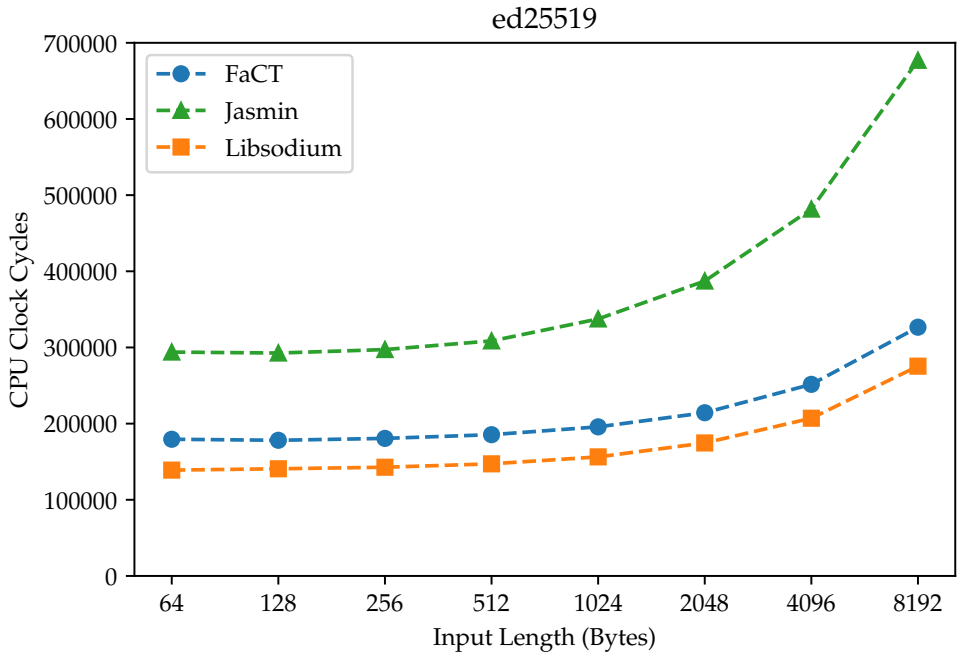
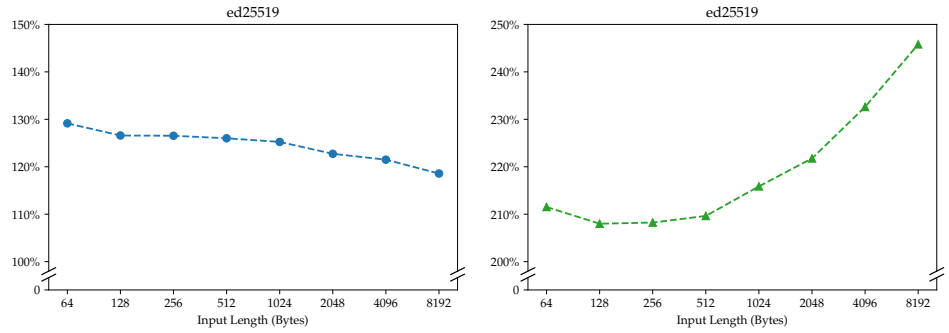


Figure 4.1: Mean clock cycles of all implementations for selected input lengths

First we'll have a look at ed25519. When studying figure 4.1, even on a simple glance, one observes that independent of input length there is a clear hierarchy of performance. With the original libsodium implementation having the lowest mean clock cycles executed and Jasmin having the highest while FaCT is situated between the two. However the distance between FaCT and libsodium is significantly smaller than the gap between Jasmin and libsodium.



(a) FaCT (b) Jasmin

Figure 4.2: Mean clock cycles given as a percentage of the baselines mean clock cycles

Overall these results are somewhat in line with expectations before running the benchmark. As Jasmin does not and can not perform assembly code optimizations when compiling its code. Unlike FaCT and standard C compilers, such as Clang and GCC, optimization through the passing of optimization flags is not supported. Jasmin as a language requires the programmer to implement code optimization by hand. Furthermore for FaCT Cauligi et al.'s performance studies [6] indicate a slight performance loss when porting code to FaCT from C.

For clearer measure of relative performance to the libsodium library we have figure 4.2. In Jasmin's plot one observes its comparative performance increases with growing input length. This is unlike FaCT where we see the comparative performance gets closer to the baseline as input length increases. Jasmin's declining performance with increased input length can even be seen in figure 4.1 as the green line seems to start ″curving″ earlier and steeper than both the blue and orange line of FaCT and the source code respectively. On the contrary in the same figure FaCT's improvement of comparative performance can not be seen by the naked eye.

A deeper look in the performance evolution of all implementations can be seen in figure 4.3. Quite apparent is the similarity in performance evolution between FaCT and libsodium, though FaCT seems to have slightly lower increases. Jasmin however seems to handle the increase in Bytes comparatively poor as from 1024 onwards its percentage increase is the largest by a noticeable margin. Nonetheless the pattern is very similar across the board.

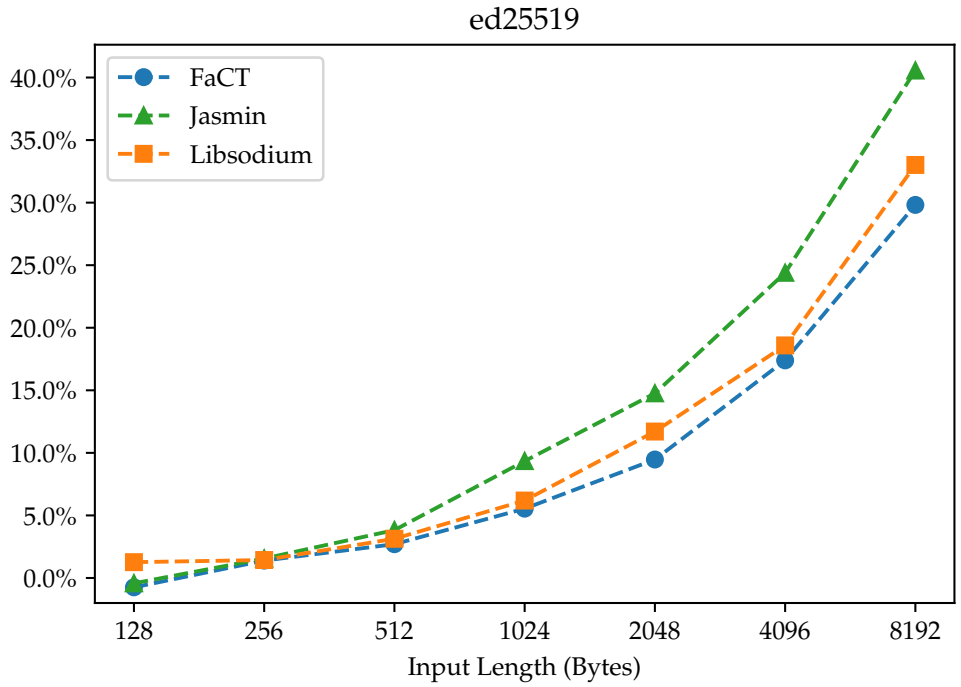


Figure 4.3: Change of mean clock cycles to the preceding input length as a percentage

4.1.2 Sha2

For sha2 we have a similar situation to ed25519. We see in figure 4.4 FaCT and the baseline are fairly close and then Jasmin has a significant gap to both of them. The reasons for this gap are identical to ed25519 as Jasmin does not perform assembly level optimizations unlike the other two. Albeit FaCT is seemingly even closer to the baseline in comparison to ed25519. All in all nothing to surprising in this figure.

In figure 4.5 however we have a deviation from ed25519. This time we have a matching tendency with a gradual increase in relative performance for both in the comparison to libsodium. For both FaCT and Jasmin their comparative performance worsens with increased input length. FaCT's percentages operate in a notably smaller range compared Jasmin and, in contrast to Jasmin, FaCT's later increase is much flatter when compared.

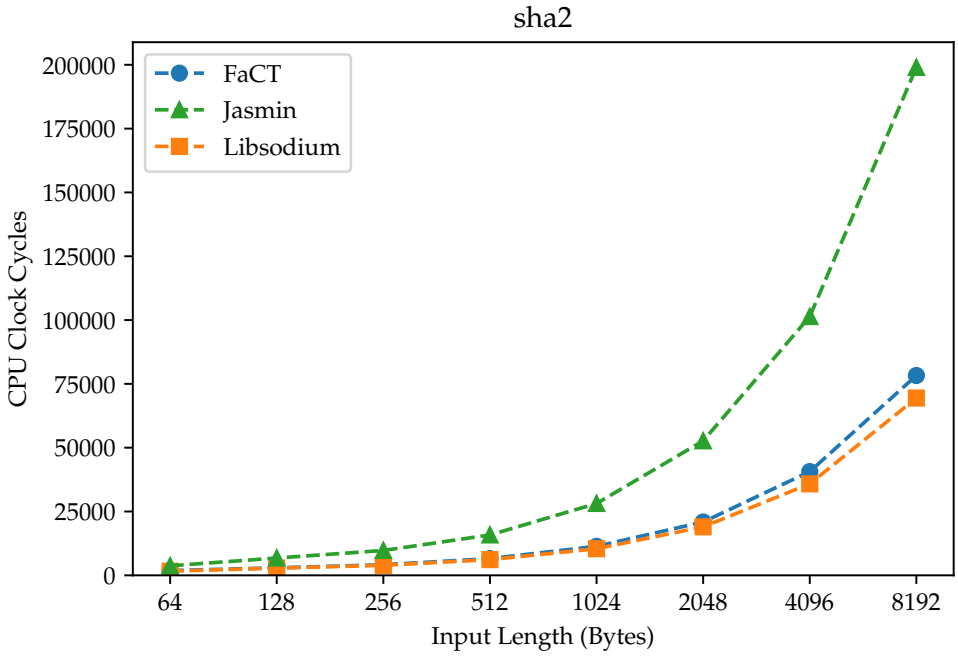
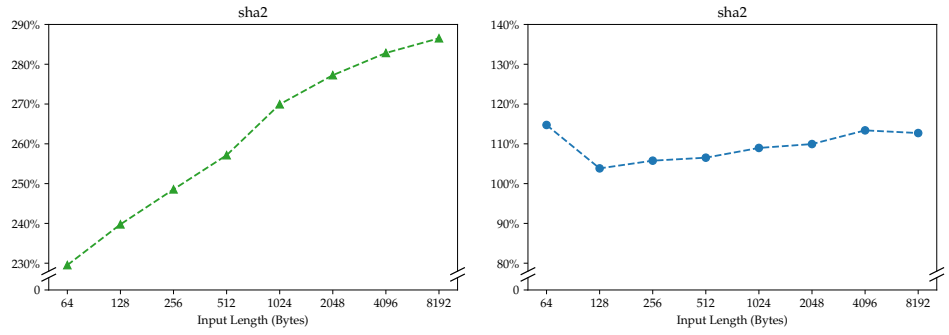


Figure 4.4: Mean clock cycles of all implementations for selected input lengths

An interesting observation is the fact that for both FaCT and Jasmin their relative best performance points are at the lower end of input sizes. Which indicates the source code seems to handle increased input size better than both FaCT and Jasmin. The same phenomenon can also be seen in figure 4.6 as the baseline suffers consistently the lowest cycle increases from 512 Bytes onward.



(a) FaCT (b) Jasmin

Figure 4.5: Mean clock cycles given as a percentage of the baselines mean clock cycles

Figure 4.6 furthermore showcases an intriguing pattern for all three lines. All of them exhibit a "bounce" around 256 Bytes. This is not outside of our expectations as the structure of this algorithm has a crucial if statement depending on input length.

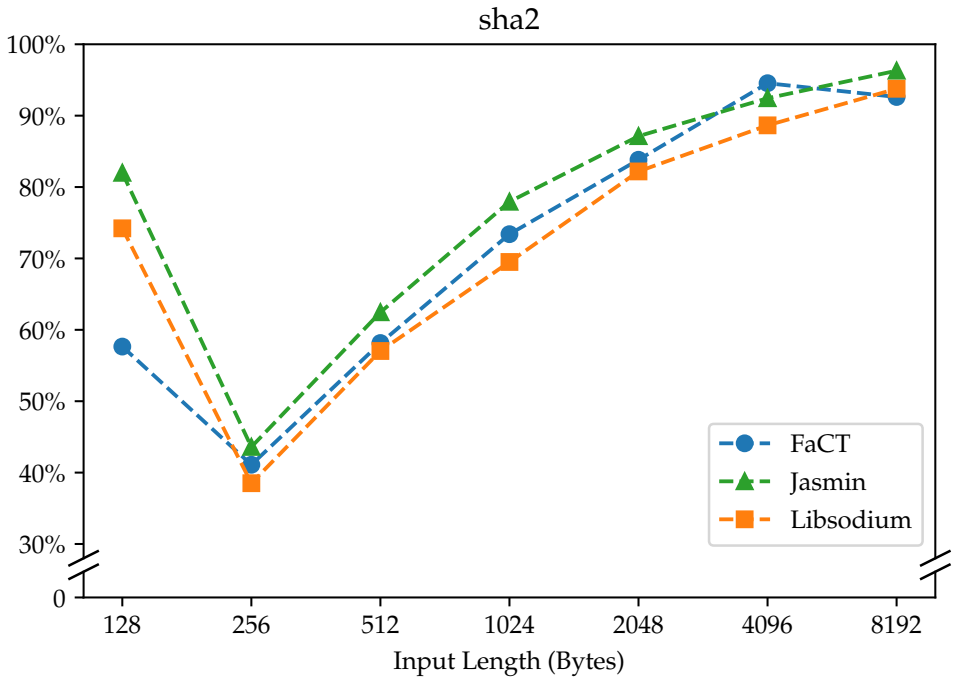


Figure 4.6: Change of mean clock cycles to the preceding input length as a percentage

When the input length is greater equal than 128 Bytes the algorithm, thanks to the mentioned if-statement, enters a part of the code which executes far more clock cycles than for the smaller input sizes. So when going from 64 to 128 Bytes we incur this penalty but afterwards the underlying executed code stays the same. Thus we are left with linear growth, attributed to number of loop iterations depend on input length, from 256 Bytes onward.

4.1.3 Curve25519

As discussed before in this section we will analyse the boxplots foreach implementation. These boxplots can be found in figures 4.7, 4.9 and 4.8 for libsodium, FaCT and Jasmin respectively.

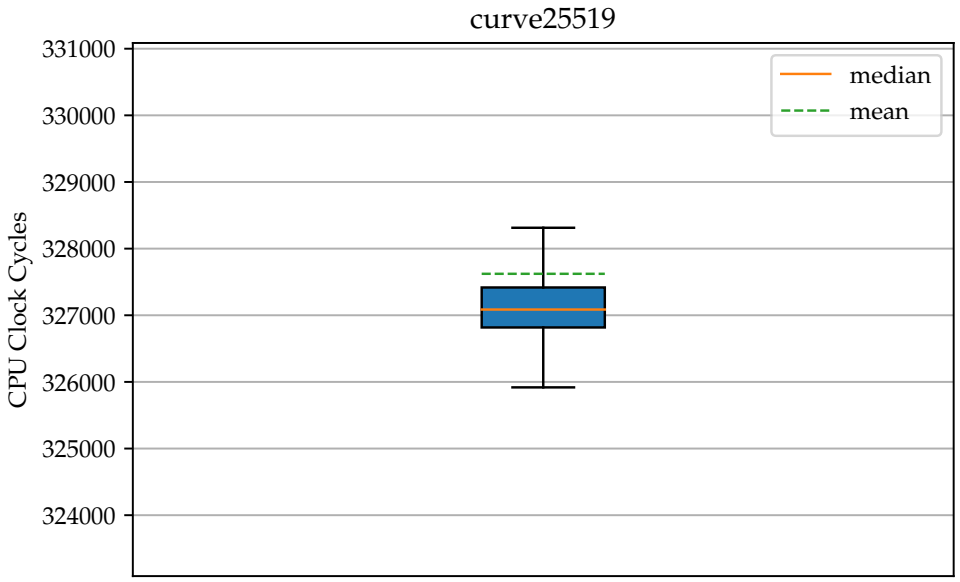


Figure 4.7: Boxplot of libsodium's data

Once again we have the familiar ordering of Jasmin, FaCT and libsodium from highest to lowest in both median and mean clock cycles. Although there is a noteworthy difference. For both ed25519 and sha2 FaCT's code ran significantly closer to the original than to Jasmin. Here however as we can deduce from the boxplots, all three are nearly equidistant in terms of both median and mean. Both the spacing between FaCT and the baseline and between FaCT and Jasmin is around 150000 cycles. We can see this detail demonstrated by means of percentages in table 4.1.

This is thanks to Jasmin's improvement and Fact's decline in relative performance compared to the previously discussed algorithms. The cause behind Jasmin's improved performance might have to do with curve25519's design as seemingly less of the code lends itself to compiler optimization. As to why FaCT's performance worsened is hard to say. Our translation might have unknowingly introduced an unintended overhead or FaCT just simply might not handle the optimization as well.

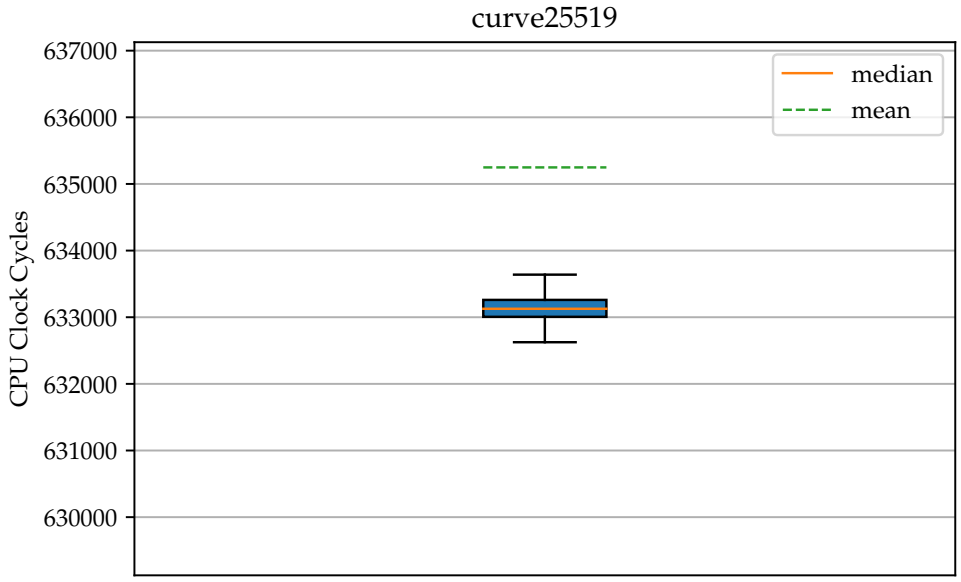


Figure 4.8: Boxplot of Jasmin's data

Another peculiarity can be seen in figure 4.8. Jasmin's boxplot is the only of the three were the mean falls decidedly outside of the whiskers. Furthermore notice the interquartile range (IQR) for Jasmin is the smallest just below libsodium's while FaCT has the largest IQR by a margin. The aforementioned fact is probably the reason why the mean is an outlier for Jasmin. The narrow interquartile range in combination with the outlying mean indicates Jasmin's data points are comparatively bunched up with large outliers.

DSL	**Percentage of libsodium's**	
	Mean	Median
FaCT	145.32%	145.52%
Jasmin	193.56%	193.689%

Table 4.1: Mean and median clock cycles in relation to libsodium

Inline with the observations from above the span of boxes behave similarly to the whiskers. A further observation reveals that the mean is always higher than the median. As we run the benchmark the OS will interfere with the running process and therefore add unrelated clock cycles to the measurements. When the OS interrupts the benchmark for a comparatively long time heavy outliers are created which skew the mean upwards.

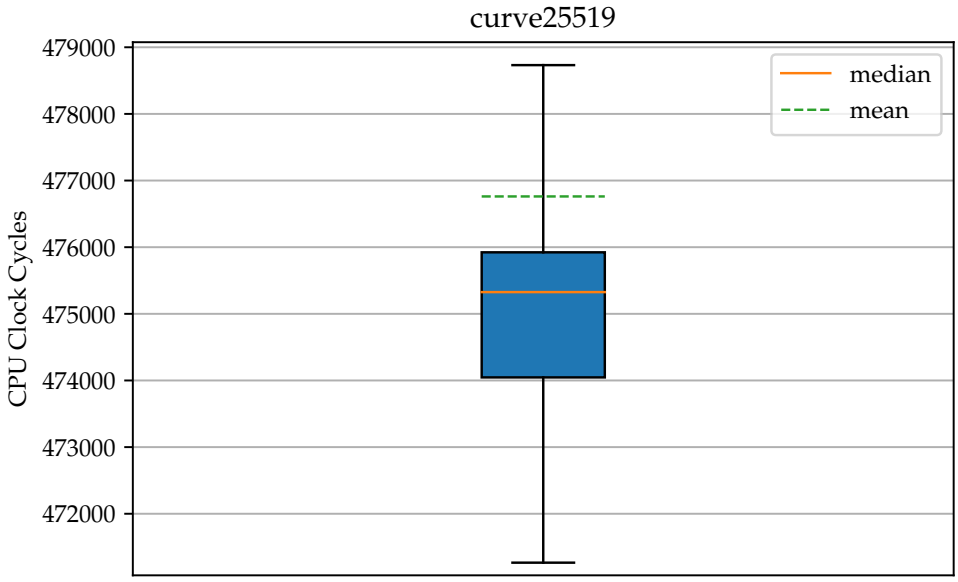


Figure 4.9: Boxplot of FaCT's data

4.1.4 Synopsis

As seen and discussed plentiful in the preceding sections the original libsodium implementation is always the best performer while Jasmin is always the worst. FaCT slots itself consistently between the two but how competitive it is in relation to libsodium is algorithm dependent. For example for sha2 FaCT is highly competetive nearly matching the baseline. On the contrary for curve25519 it virtually incurs a 50% penalty.

Jasmin's poor performance is a consequence of its own design. Jasmin compiles its code to assembler as unchanged as possible to give the programmer greater influence over the compiled code. As a consequence any type of optimization needs to be done by the programmer explicitly and the quality of such optimizations therefore depend on the programmer's expertise. Since we attempted to translate the source code with the least amount of alterations possible, no special augmentations of the code for performance optimization were done. Expectedly Jasmin's performance suffered as a result. On the other side FaCT's optimization options appear to be reasonably competitive with their equivalent on modern C compilers.

Quite obvious is the difference between the design philosophies behind both of these DSLs. FaCT is built analogous to a modern high-level programming language which outsources much of its performance optimization to the compiler instead of the programmer. Jasmin meanwhile is designed more closely to an assembler language which emphasizes the ability of programmer to greatly influence the inner workings of the end product and therefore leaves performance optimizations to the programmer. Moreover through direct translation to assembly code Jasmin guarantees the compilation step preserves constant-time [2]. FaCT meanwhile guarantees the conservation of constant-time for the full transformation to LLVM intermediate representation (LLVM IR) but does not guarantee constant-time preservation from LLVM IR to the final assembly code [6].

Scaling with input length had no clear leading performer. As FaCT seemed to scale better than the source code for ed25519. On the contrary for sha2 the inverse was true. Since the underlying curve25519 core functions used in ed25519 are independent of input length, this curiously would imply that FaCT's sha2 function calls inside ed25519 scaled better than libsodiums. This seems contradictory and could warrant a further investigation but is outside the scope of this paper. None of the results were outside the expected.

4.2 Storage

As outlined in chapter 3 valgrind's massif tool was utilized to profile heap and stack memory. The visualized results of this procedure can be found in the appendix under section A.1 in the corresponding segments for each algorithm. To alleviate tables have been provided in each section where both peak and floor stack sizes have been given as a percentage of the baseline's. Important to note is that anomalous values at the begging and end of the plot have been ignored in determining and discussing the relevant data points. Plots which are discussed in the following sections will be referenced explicitly from the appendix. For the segment on binaries a simple table is given in the section itself.

Ed25519

DSL	Percentage of libsodium's stack size	
	peak	floor
FaCT	100.92%	100.00%
Jasmin	122.68%	122.68%

Table 4.2: Stack size peak and floor during the execution of ed25519 compared to the baseline in percentage

When comparing all three plots there are two key observations. To start with Jasmin's recorded data, in figure A.2, hovers around 160kB while both FaCT's and libsodium's data, in figure A.1 and figure A.3 respectively, ranges inbetween 130kB to 135kB. Plainly said Jasmin uses around 30kB more storage. The same information can be seen in percentages in table 4.2. Secondly Jasmin's stack size stays constant. On the other hand both FaCT's and Jasmin's stack size oscillates over the whole execution. Seemingly FaCT and the source code constantly allocate and deallocate storage while Jasmin allocates only at the beginning.

Sha2

DSL	Percentage of libsodium's stack size	
	peak	floor
FaCT	101.72%	100.00%
Jasmin	100.57%	100.00%

Table 4.3: Stack size peak and floor during the execution of sha2 compared to the baseline in percentage

On the whole three plots in figures A.4, A.5) and A.6, for FaCT, Jasmin and libsodium respectively, might as well be as similar as possible. This is very much reflected in table 4.3 since the maximal observed difference is only 1.72%. There is neither a clear stand-out performer nor does any implementation show any deviation in the general plot pattern. In this regard the only noteworthy point is there being nothing to meaningfully differentiate between them.

Curve25519

DSL	Percentage of libsodium's stack size	
	peak	floor
FaCT	145.95%	168.33%
Jasmin	122.30%	57.33%

Table 4.4: Stack size peak and floor during the execution of curve2551 compared to the baseline in percentage

This time there is a clear pecking order as libsodium, seen in figure A.9, clearly uses the least amount stack space. Then comes Jasmin, seen in figure A.8, which is closely followed by the last placed FaCT implementation, seen in figure A.7. Interestingly both FaCT and libsodium oscillate with a high frequency and low amplitude while Jasmin oscillates with a low frequency and a high amplitude. Indeed as seen in table 4.4 Jasmin's floor is by far the lowest, but Jasmin's peak is still higher than libsodium's peak.

Binaries

DSL	Algorithm		
	Curve25519 *(Bytes)*	Sha2 *(Bytes)*	Ed25519 *(Bytes)*
FaCT	16'512	9'240	46'064
Jasmin	162'200	206'728	1'521'816

Table 4.5: Size of the produced binaries in Bytes

Jasmin evidently produces binaries of larger size by quite a margin. Jasmin's files are larger by a factor of 10, 20 and 30 respectively for curve25519, sha2 and ed25519. Jasmin was assumed to perform worse in this metric. The programmer has by design a much greater influence on the produced assembly code in Jasmin and Jasmin does not perform any optimization in general. Consequently code which is not optimized for small size, like Jasmin's in this case, does not fare well against compiler optimized code, like FaCT's implementation. Further a significant factor will be the loop unrolling Jasmin does on compilation. Since there appear several for loops in all Jasmin translations it is factor responsible for exacerbating the situation. Having said that factors of 10, 20 and 30 times are fairly significant.

Synopsis

Overall it is fair to say in terms of stack size during execution libsodium performs the best since it is either the best outright or at worst functionally equivalent to the other two. For the comparison between FaCT and Jasmin there is no clear winner in terms of size minimization but there is a clear difference in the patterns of size variations over the execution. Jasmin's oscillations have an observable lower frequency than both FaCT and libsodium. As for binaries FaCT clearly creates far smaller files than Jasmin. Again we come back to the difference in design philosophies between FaCT and Jasmin as discussed in section 4.1.4. FaCT's compiler induced code optimizations lead to shorter binaries. To get similar results in Jasmin a programmer has to have extensive knowledge on manual code optimizations and further to correctly apply it.

Chapter 5

Conclusion

FaCT and Jasmin are two DSLs with very differing core design principles. Jasmin sacrifices ease-of-use and easily attainable performance for greater control over the resulting program and verified constant-time preservation. While this seems to be a reasonable sacrifice it has created potential problems in usage. Thanks to the weighty challenges faced when trying to implement modular and interconnected code suites it is not reasonable to write a whole library such as libsodium inside Jasmin. Thus only parts of said library could be written in Jasmin with a high-level language, like C, used as connecting glue thereby introducing a verification gap at the top in such a practical application.

Conversely FaCT attains the ease-of-use and automatic optimizations of a modern compiler. As a result it is very much possible to implement extensive, interconnected and modular code suites without any crucial challenges. This comes however at the cost of only ensuring the preservation of constant-time up to the conversion to LLVM IR, leaving a verification gap at the lower end of the tool.

Because of these design philosophies it is apparent Jasmin is made to be used by experienced and knowledgable cryptographic programmers only. FaCT meanwhile is, after getting over the introductory hurdle, quite natural to use for programmers with intermediate experience. Although some cryptographic expertise is probably required to write cryptographic sound code.

Appendix A

Appendix

A.1 Valgrind Massif Plots

A.1.1 Ed25519

FaCT

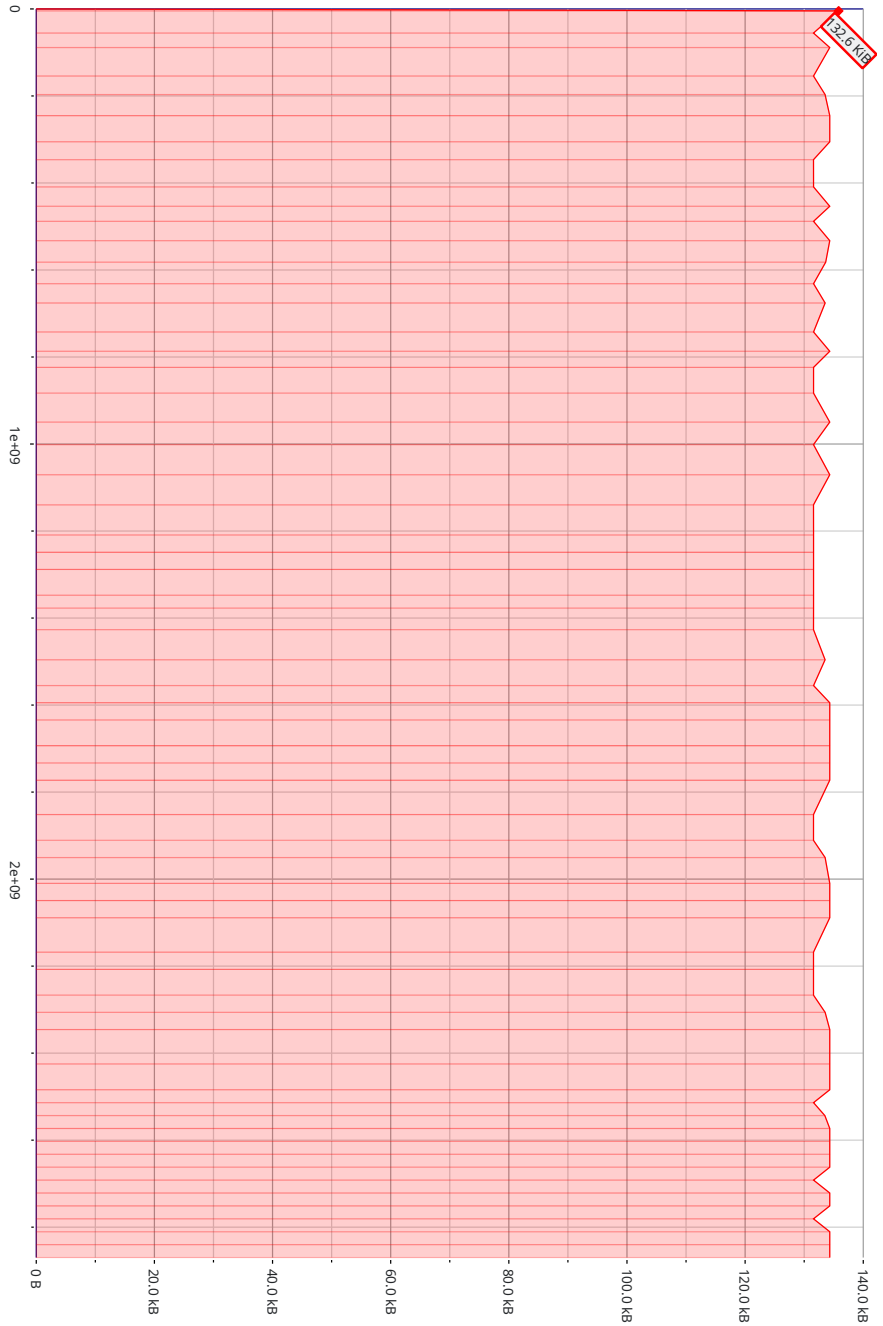


Figure A.1: Stack profiling of FaCT: X-axis: time in instructions, Y-axis: stack size

Jasmin

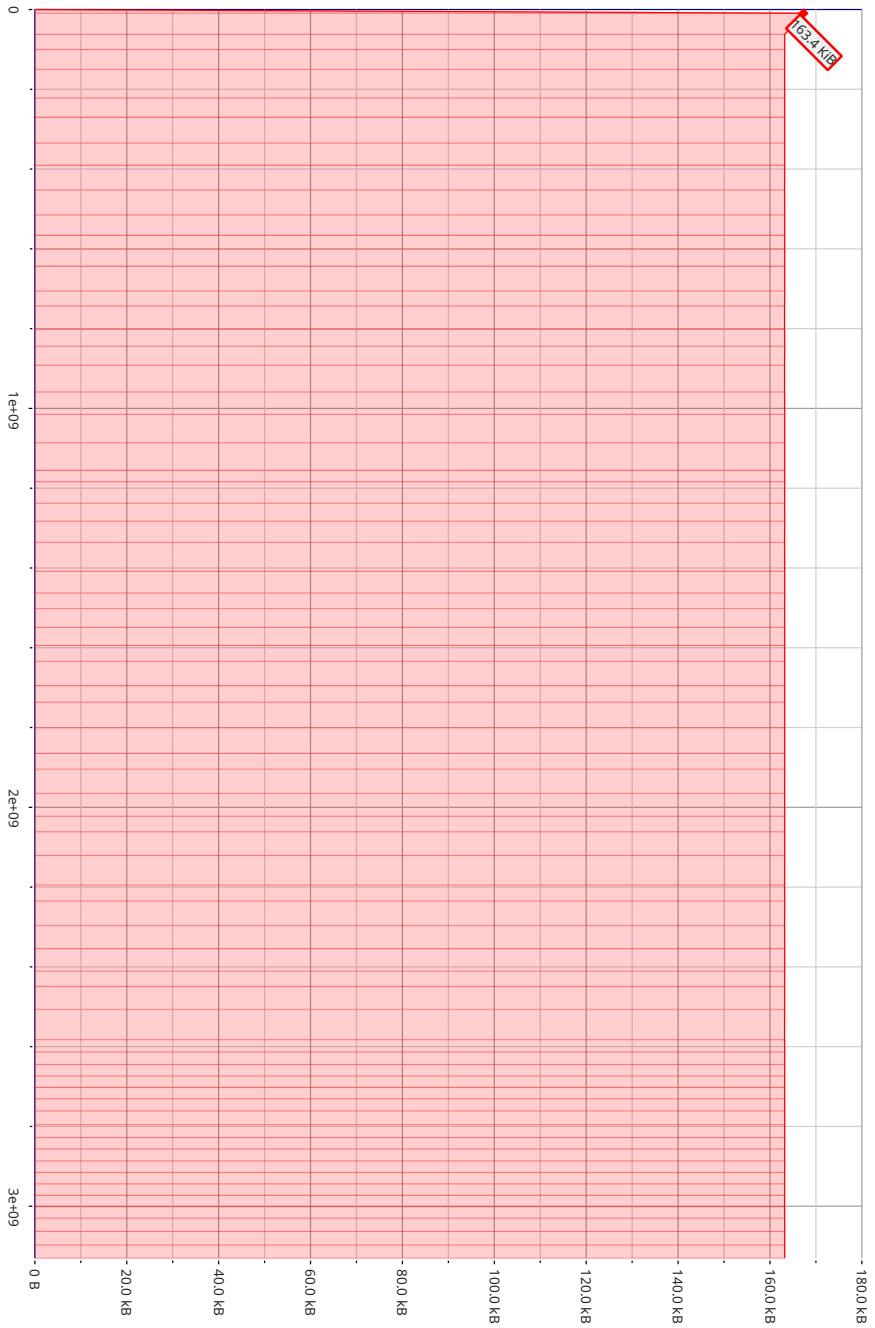


Figure A.2: Stack profiling of Jasmin: X-axis: time in instructions, Y-axis: stack size

Libsodium

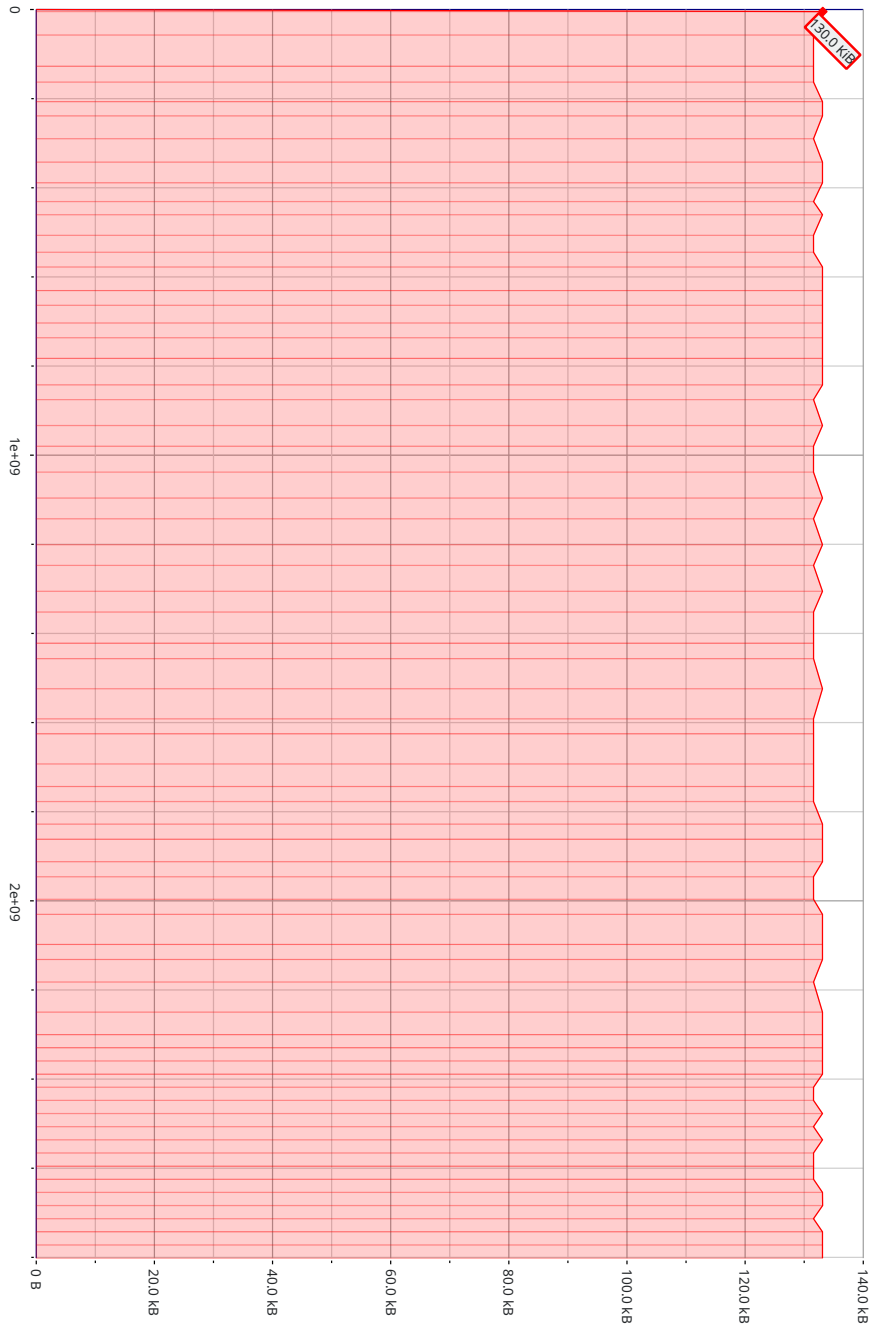


Figure A.3: Stack profiling of libsodium: X-axis: time in instructions, Y-axis: stack size

A.1.2 Sha2

FaCT

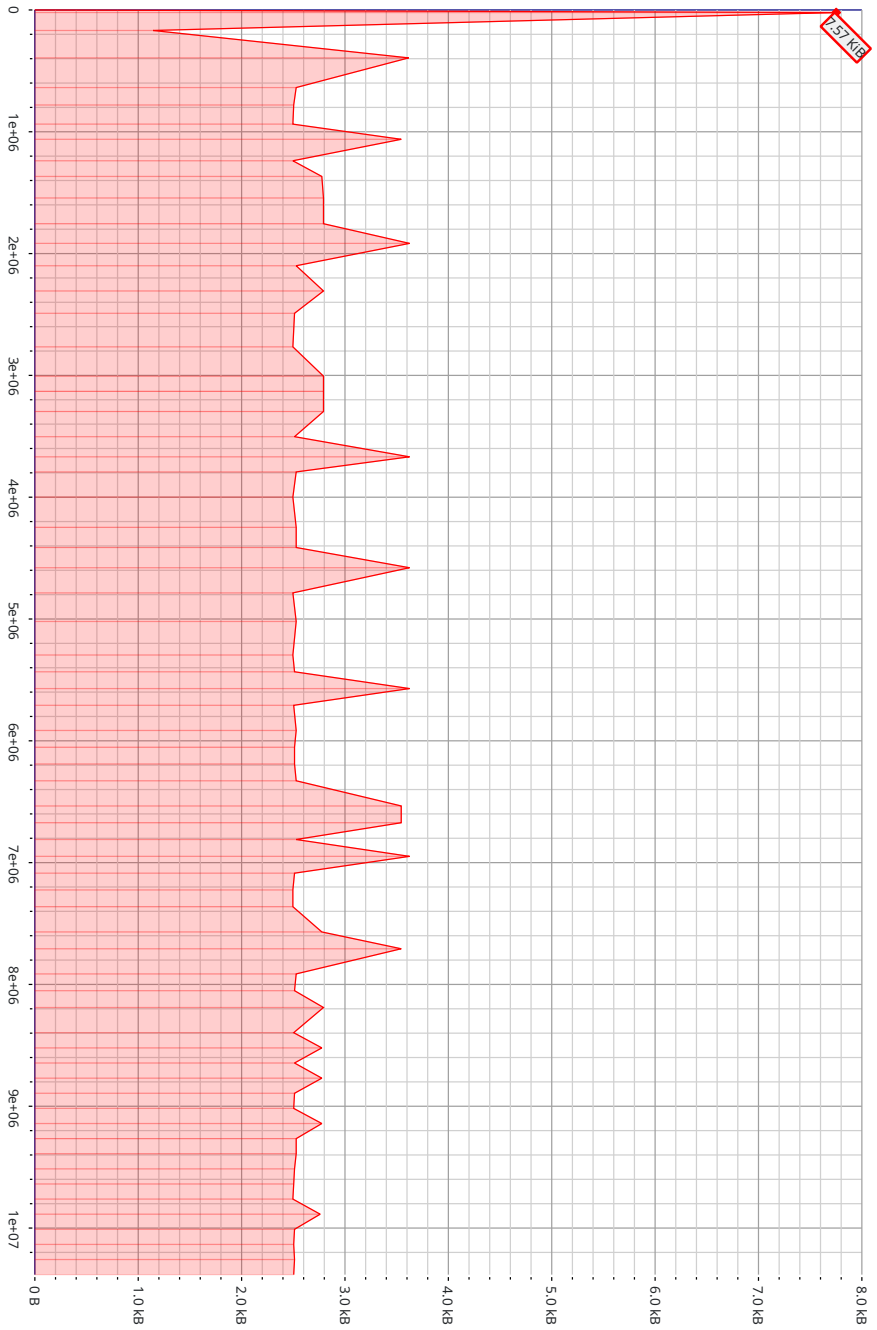


Figure A.4: Stack profiling of FaCT: X-axis: time in instructions, Y-axis: stack size

Jasmin

Figure A.5: Stack profiling of Jasmin: X-axis: time in instructions, Y-axis: stack size

Libsodium

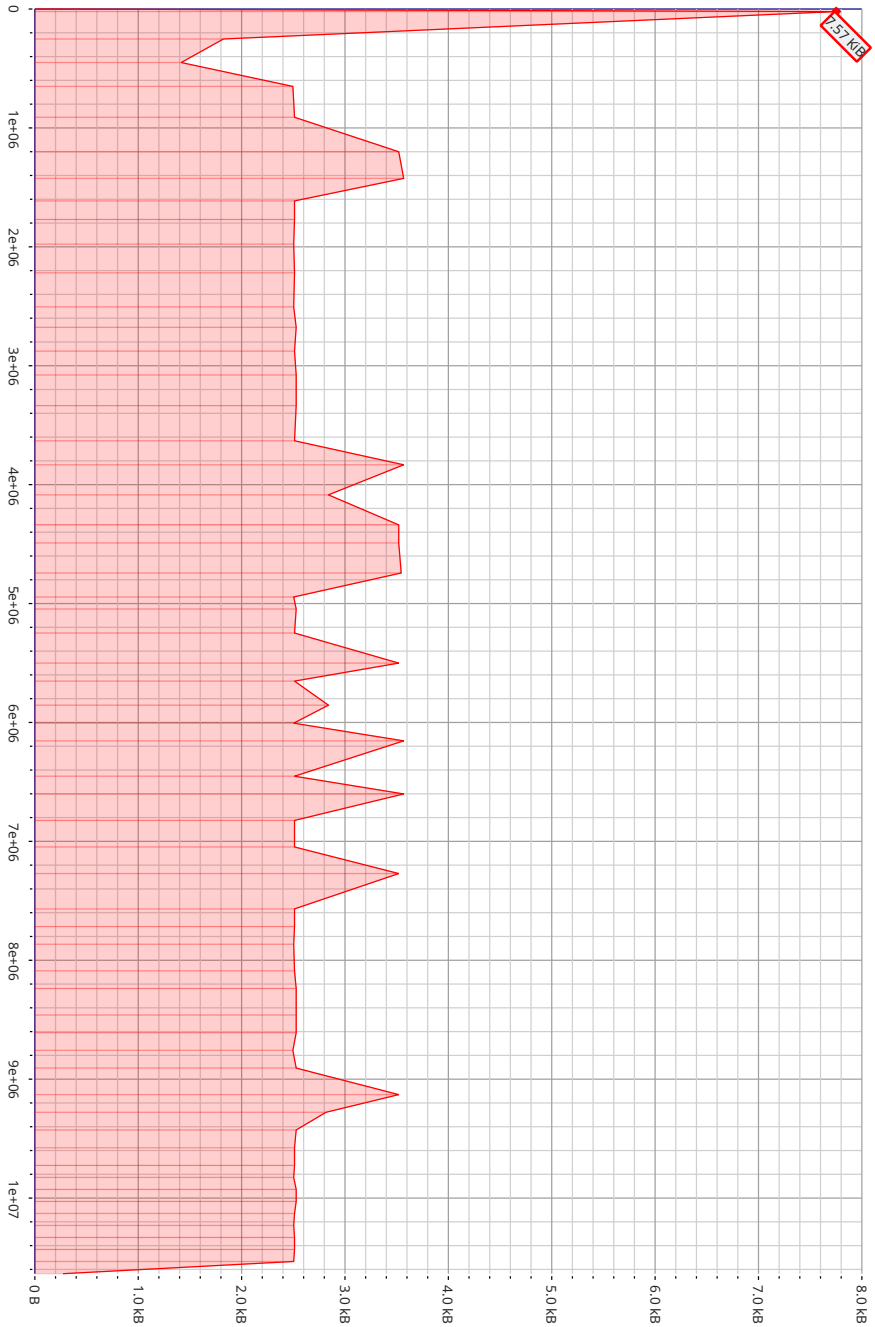


Figure A.6: Stack profiling of libsodium: X-axis: time in instructions, Y-axis: stack size

A.1.3 Curve25519

FaCT

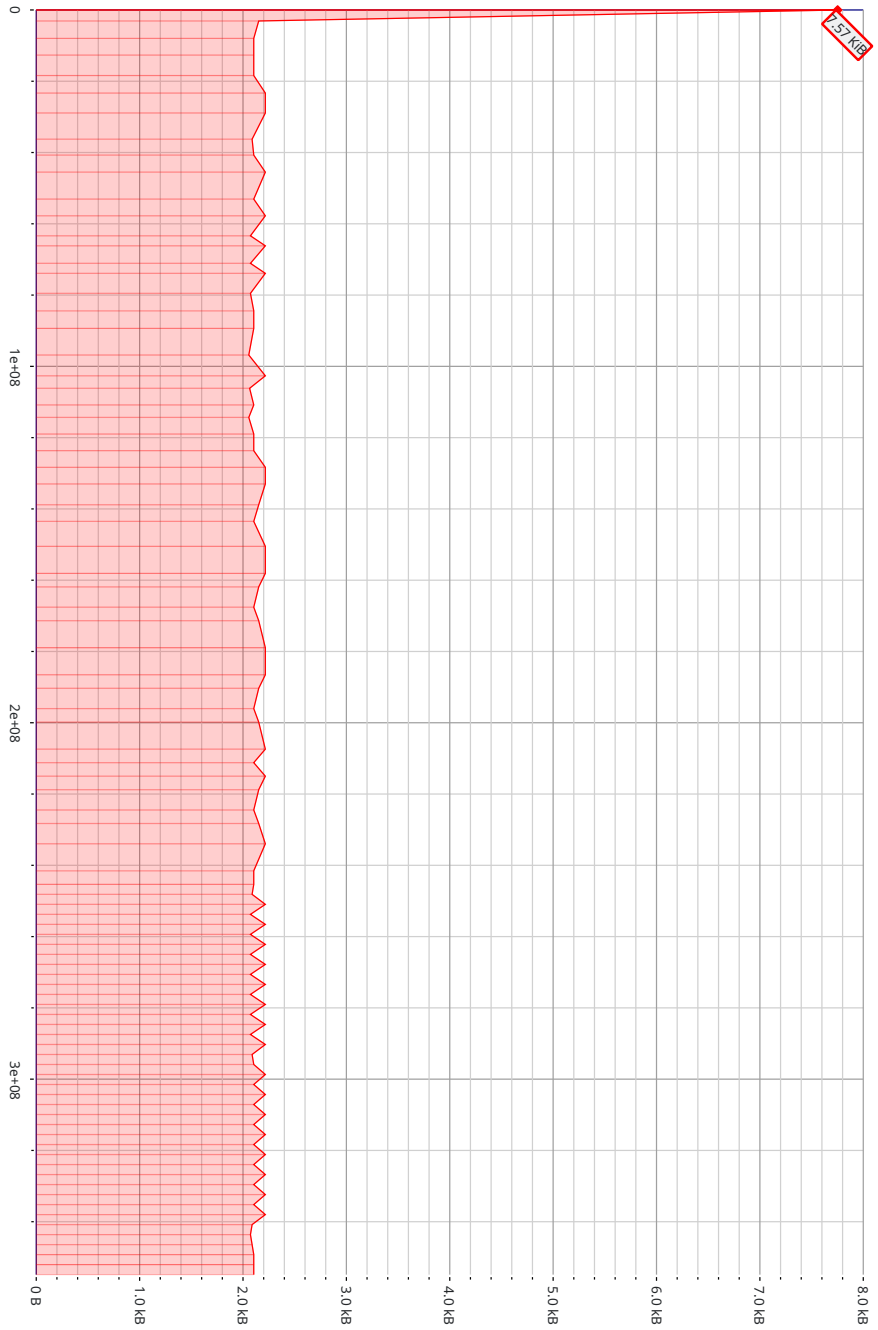


Figure A.7: Stack profiling of FaCT: X-axis: time in instructions, Y-axis: stack size

Jasmin

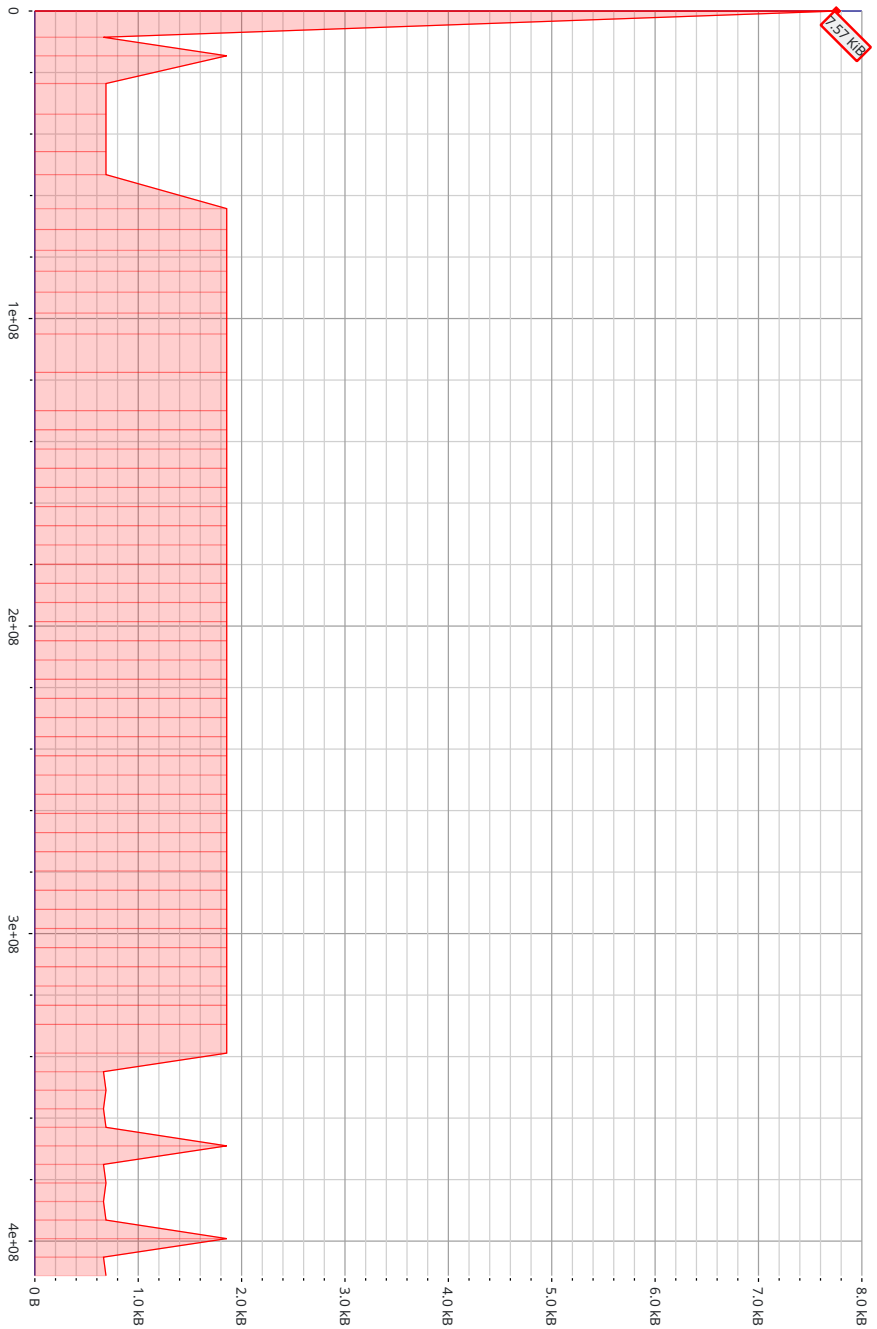


Figure A.8: Stack profiling of Jasmin: X-axis: time in instructions, Y-axis: stack size

Libsodium

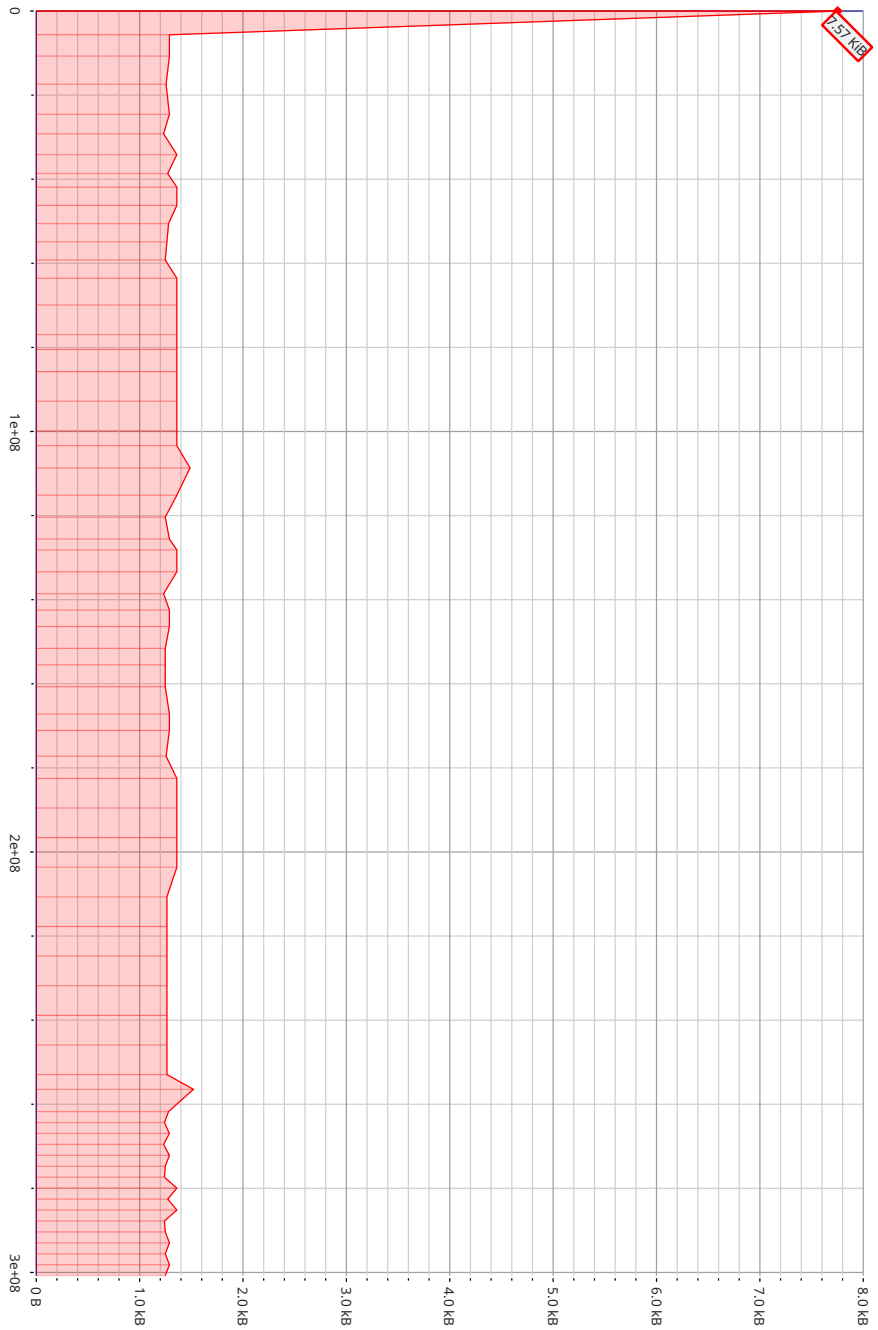


Figure A.9: Stack profiling of libsodium: X-axis: time in instructions, Y-axis: stack size

A.2 Benchmark of Jasmin's Sha2

```
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "../ed25519/plain_C_source/mt19937ar.c"

static const uint64_t Krnd[80] = {
    0x428a2f98d728ae22ULL, 0x7137449123ef65cdULL, 0xb5c0fbcfec4d3b2fULL,
    0xe9b5dba58189dbbcULL, 0x3956c25bf348b538ULL, 0x59f111f1b605d019ULL,
    0x923f82a4af194f9bULL, 0xab1c5ed5da6d8118ULL, 0xd807aa98a3030242ULL,
    0x12835b0145706fbeULL, 0x243185be4ee4b28cULL, 0x550c7dc3d5ffb4e2ULL,
    0x72be5d74f27b896fULL, 0x80deb1fe3b1696b1ULL, 0x9bdc06a725c71235ULL,
    0xc19bf174cf692694ULL, 0xe49b69c19ef14ad2ULL, 0xefbe4786384f25e3ULL,
    0x0fc19dc68b8cd5b5ULL, 0x240ca1cc77ac9c65ULL, 0x2de92c6f592b0275ULL,
    0x4a7484aa6ea6e483ULL, 0x5cb0a9dcbd41fbd4ULL, 0x76f988da831153b5ULL,
    0x983e5152ee66dfabULL, 0xa831c66d2db43210ULL, 0xb00327c898fb213fULL,
    0xbf597fc7beef0ee4ULL, 0xc6e00bf33da88fc2ULL, 0xd5a79147930aa725ULL,
    0x06ca6351e003826fULL, 0x142929670a0e6e70ULL, 0x27b70a8546d22ffcULL,
    0x2e1b21385c26c926ULL, 0x4d2c6dfc5ac42aedULL, 0x53380d139d95b3dfULL,
    0x650a73548baf63deULL, 0x766a0abb3c77b2a8ULL, 0x81c2c92e47edaee6ULL,
    0x92722c851482353bULL, 0xa2bfe8a14cf10364ULL, 0xa81a664bbc423001ULL,
    0xc24b8b70d0f89791ULL, 0xc76c51a30654be30ULL, 0xd192e819d6ef5218ULL,
    0xd69906245565a910ULL, 0xf40e35855771202aULL, 0x106aa07032bbd1b8ULL,
    0x19a4c116b8d2d0c8ULL, 0x1e376c085141ab53ULL, 0x2748774cdf8eeb99ULL,
    0x34b0bcb5e19b48a8ULL, 0x391c0cb3c5c95a63ULL, 0x4ed8aa4ae3418acbULL,
    0x5b9cca4f7763e373ULL, 0x682e6ff3d6b2b8a3ULL, 0x748f82ee5defb2fcULL,
    0x78a5636f43172f60ULL, 0x84c87814a1f0ab72ULL, 0x8cc702081a6439ecULL,
    0x90befffa23631e28ULL, 0xa4506cebde82bde9ULL, 0xbef9a3f7b2c67915ULL,
    0xc67178f2e372532bULL, 0xca273eceea26619cULL, 0xd186b8c721c0c207ULL,
    0xeada7dd6cde0eb1eULL, 0xf57d4f7fee6ed178ULL, 0x06f067aa72176fbaULL,
    0x0a637dc5a2c898a6ULL, 0x113f9804bef90daeULL, 0x1b710b35131c471bULL,
    0x28db77f523047d84ULL, 0x32caab7b40c72493ULL, 0x3c9ebe0a15c9bebcULL,
    0x431d67c49c100d4cULL, 0x4cc5d4becb3e42b6ULL, 0x597f299cfc657e2aULL,
    0x5fcb6fab3ad6faecULL, 0x6c44198c4a475817ULL
};


void cryptoB_hash_sha512(uint8_t *out, uint8_t *in, uint64_t inlen, uint64_t *srcKrnd);


int main(int argc, char **argv) {

        init_genrand(19876168137);

        static uint64_t sampleSize = 32000;

        uint64_t store[sampleSize];

        double mean = 0.0;
        double hold = 0.0;

        unsigned char outO[64];
        unsigned char outA[64];

        memset(outO,0,64);
        memset(outA,0,64);

        uint64_t lengthIn = 1024;

        if(argc > 1) {
                lengthIn = strtoul(argv[1],NULL,10);
        }

        unsigned char inO[lengthIn];
        unsigned char inA[lengthIn];

        memset(inO,0,lengthIn);
        memset(inA,0,lengthIn);
```

```
        for(uint32_t i=0; i<2000; ++i) {

                uint64_t length = lengthIn;
                uint64_t lengthb = lengthIn;

                for(uint64_t j=0; j<length; ++j) {
                        unsigned char t = genrand_int32() >>24;
                        inO[j] = t;
                        inA[j] = t;
                }

                cryptoB_hash_sha512(outA,inA,lengthb,Krnd);

        }


        for(uint32_t i=0; i<sampleSize; ++i) {
                unsigned cycles_low, cycles_high, cycles_low1, cycles_high1;
                uint64_t start, end;

                uint64_t length = lengthIn;
                uint64_t lengthb = lengthIn;

                for(uint64_t j=0; j<length; ++j) {
                        unsigned char t = genrand_int32() >>24;
                        inO[j] = t;
                        inA[j] = t;
                }

                asm volatile ("CPUID\n\t"
                        "RDTSC\n\t"
                        "mov %%edx, %0\n\t"
                        "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
                "%rax", "%rbx", "%rcx", "%rdx");

                cryptoB_hash_sha512(outO,inO,length,Krnd);

                asm volatile("RDTSCP\n\t"
                "mov %%edx, %0\n\t"
                "mov %%eax, %1\n\t"
                "CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1)::
                "%rax", "%rbx", "%rcx", "%rdx");

                start = ( ((uint64_t)cycles_high << 32) | cycles_low );
                end = ( ((uint64_t)cycles_high1 << 32) | cycles_low1 );
                store[i] = end - start;

                hold = store[i];
                hold /= sampleSize;
                mean += hold;
        }

        double var = 0.0;
        for(uint32_t i = 0; i < sampleSize; ++i) {
                hold = pow((store[i] - mean),2);
                hold /= sampleSize-1;

                var += hold;
        }
        var = sqrt(var);

        printf("%lu,%f,%f\n", lengthIn, mean, var);

        return 0;
}
```

Bibliography

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-assurance and high-speed cryptography". In: *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1807–1823.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations ". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 965–982.

[3] Daniel J. Bernstein. *Cache-timing attacks on AES*. 2005.

[4] Daniel J. Bernstein. *Curve25519: new Diffie-Hellman speed records*. 2006.

[5] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. *High-speed high-security signatures*. 2012.

[6] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. "Fact: A dsl for timing-sensitive computation". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 174–189.

[7] *Libsodium Documentation*. URL: https://doc.libsodium.org/ (visited on 05/21/2022).

[8] Makoto Matsumoto. *Mersenne Twister Home Page*. URL: http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/MT2002/emt19937ar.html (visited on 05/21/2022).

[9] Nicholas Nethercote and Julian Seward. *Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation*. 2007.

[10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel © IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation. Sept. 2010. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (visited on 05/21/2022).