SPARKNaCl: a verified, fast re-implementation of TweetNaCl

or...

“What I did during lockdown…”

Roderick Chapman,
Director, Protean Code Limited
Honorary Visiting Professor, Dept. of Computer Science, University of York
Contents

• TweetNaCl? Why bother?

• Goal and not goals...

• Proof

• Performance

• Further work
Contents

- TweetNaCl? Why bother?
- Goal and not goals...
- Proof
- Performance
- Further work
TweetNaCl?

- A complete implementation of the NaCl crypto API, written in C.
- Source code fits in 100 tweets!
- No comments at all in the code...
  - One 16-page technical paper describes how it works.
  - Assurance largely rests on dynamic test, lots of users and the formidable reputation of the authors.
TweetNaCl?

- Core algorithms:
  - Stream cipher: Salsa20
  - Authentication: Poly1305
  - Hash: SHA2-512
  - Signatures: Ed25519
  - Key exchange: X25519 (ECDH using Curve25519)
TweetNaCl?

- The code is very hard to understand...
  - No comments...
  - Cunning mathematical tricks to achieve acceptable performance...
  - “Constant Time” programming style to mitigate side-channel attack...
  - Somewhat terse overview in the paper...
- For example...
TweetNaCl?

• From the paper:

"Arithmetic modulo the group order. Signing requires reduction of a 512-bit integer modulo the order of the Curve25519 group, a prime \( p = 2^{252} + \delta \) where \( \delta \approx 2^{124.38} \). We store this integer as a sequence of limbs in radix \( 2^8 \). We eliminate the top limb of the integer, say \( 2^{504}b \), by subtracting \( 2^{504}b \) and also subtracting \( 2^{252}\delta b \); we then perform a partial carry so that 20 consecutive limbs are each between \(-2^7\) and \(2^7\). We repeat this procedure to eliminate subsequent limbs from the top. We similarly eliminate any remaining multiple of \( 2^{252} \), leaving an integer between \(-1.1 \cdot 2^{251}\) and \(1.1 \cdot 2^{251}\). We then multiply the final carry bit by \( p \) and add, obtaining an integer between 0 and \( p - 1 \), and carry in the traditional way so that each limb is between 0 and 255."

• Becomes...

There's a mistake in this paragraph... Can you spot it?
// Just for completeness...

typedef long long i64;

// Note that a “gf” is 1024 bits
// or 128 bytes
typedef i64 gf[16];

// Some tweet-saving abbreviations...
#define sv static void

#define FOR(i,n) for (i = 0; i < n; ++i)
sv modL(u8 *r, i64 x[64])
{
i64 carry, i, j;
    for (i = 63; i >= 32; --i) {
        carry = 0;
        for (j = i - 32; j < i - 12; ++j) {
            x[j] += carry - 16 * x[i] * L[j - (i - 32)];
            carry = (x[j] + 128) >> 8;
            x[j] -= carry << 8;
        }
        x[j] += carry;
    }
    x[i] = 0;
}
carry = 0;
FOR(j, 32) {
    x[j] += carry - (x[31] >> 4) * L[j];
    carry = x[j] >> 8;
    x[j] &= 255;
}
FOR(j, 32) x[j] -= carry * L[j];
FOR(i, 32) {
    x[i+1] += x[i] >> 8;
    r[i] = x[i] & 255;
}
}
Why Bother?

• There is lots of other excellent work on formal verification of Crypto libraries – MSR Everest, AWS, Galois’ Cryptol & SAW, Jasmin & EasyCrypt, HACSPEC, Fiat Cryptography etc. etc.

• There are several very highly respected implementations of NaCl out there, including LibSodium.

• Why bother with another one?
Why Bother?

• Can a modern deductive verification system and programming language cope with a library like NaCl?

  1. What can be proven anyway?

  2. What level of automation can be achieved?
Myth

“Formal is Slow…”

• So-called “Formal” languages have a reputation in industry for being “too slow” for production use.

• Put another way: “Our code has to go really fast, so we can only write it on C, C, C, or C... (or assembly language...)”

• Can a “formal” language like SPARK really compete?
Contents

- TweetNaCl? Why bother?
- Goal and not goals...
- Proof
- Performance
- Further work
Goals...

• A re-implementation of TweetNaCl in SPARK, covering the whole library and NaCl API.

• It shall pass all NaCl regression tests.

• Retain “constant time” programming style.

• Fully “auto-active” proof of at least type- and memory safety for the whole thing.
  
  • It’s all about the types... (and contracts...)

  • No “interactive” proof please...
Goals...

- Compatible with the “Zero Footprint” runtime library of GCC/GNAT
  - No COTS or libraries.
  - “Heap free” programming style, so will run on ”bare-metal” targets with no OS at all.
  - *Same code* will compile and run on *any* target CPU/OS.
Goals...

- Performance and code size competitive with TweetNaCl.

- Will we find any bugs? En-passant, will we find bugs in TweetNaCl too?

- We have form in this area: a subtle corner case bug was found in the Skein hash algorithm by re-implementing it in SPARK...
NOT Goals…

• Don’t try to compete with LibSodium’s performance…

• Don’t try to prove partial correctness (unless strictly necessary).

• Don’t try to prove mathematical security-related properties of Curve25519, Ed25519 and so on…
Contents

• TweetNaCl? Why bother?
• Goal and not goals...
• Proof
• Performance
• Further work
Some stats

- SPARKNaCl is 1786 logical lines of code, of which 664 statements and 1122 declarations.

- 111 subprogram bodies, comprising 74 procedures and 37 functions.
Contracts...

- Types and subtypes: LOTS!
- Precondition: 27
- Postcondition: 20
- Dynamic_Predicate on subtype: 8
- Assert: 51
Some stats

- Loop statements: 56
  - User-written loop invariant not required (auto-generated invariant is good enough): 25
  - User-written loop invariant required, but trivial: 18
  - User-written loop invariant, but hard: 13
  - “Hard” = “Hours or days of effort to understand the algorithm, and get the proof to automate OK”
The hard bits...

- The most “difficult” (for formal verification) loops are in:
  - “*” operator for type GF (256-bit large integer)
  - The “Carry-and-Reduce (Car)” functions for reducing a GF large integer modulo $2^{255}-19$.
  - The “Modulo L” operation (see above...) within the Ed25519 Sign and Open functions.
  - The first two are also the most performance-critical inner loops of the Sign algorithm...
What has been proven?

- "Type and Memory Safety"
  - No undefined behaviour
  - No dependence on unspecified behaviour
  - Nothing that would normally raise an exception: range violation, arithmetic overflow, buffer overflow, division-by-zero etc.
  - (Note: there are no pointers! And no heap...)
  - All object values satisfy all type declarations.
- All user-supplied contracts are true (pre, post, loop invariants, asserts...)
What has been proven?

- **Observations:**

  1. Types are cool. Better (i.e. “stronger”) types get you better proof. Never write a “Pre” or an “Assert” where a type could do the job just as well.

  2. Expressive use of types is the key to achieving proof automation and completeness.

  3. “Type safety” is not a “fixed price” concept in SPARK. More types -> More proof.

  4. Types serve as a (weak-ish) specification of correctness properties...

  5. Fully predicated types from Ada2012 (aka “Liquid Types” in Haskell and OCAML) are awesome.
Predicated types? Eh?

LM    : constant := 65536; -- Limb Modulus
LMM1  : constant := 65535; -- "LM Minus 1"
R2256 : constant := 38; -- We’ll come back to this one...
   -- "Maximum GF Limb Coefficient"
MGFLC : constant := (R2256 * 15) + 1;
   -- "Maximum GF Limb Product"
MGFLP : constant := LMM1 * LMM1;

subtype GF64_Any_Limb is I64 range -LM .. (MGFLC * MGFLP);
type GF64 is array (Index_16) of GF64_Any_Limb;
subtype GF64_Normal_Limb is GF64_Any_Limb range 0 .. LMM1;
subtype Normal_GF64 is GF64
   with Dynamic_Predicate =>
      (for all I in Index_16 => Normal_GF64 (I) in
       GF64_Normal_Limb);
What has been proven?

- The code generates 907 Verification Conditions.

- Final result: 100% automation/completeness for all VCs using Z3, CVC4 and Alt-Ergo provers.

- Of those 907
  
  - 777 are trivial and proven by all 3 provers.
  
  - 130 are “hard” in that at least one prover fails to prove it.
Proof Performance

• Proof of the whole library takes about 4.5 minutes from scratch on my laptop (using 16 threads).

• Results are cached to improve re-proof of units that haven’t changed.

• Generating lots of independent VCs parallelizes very well... Just throw more CPU cores at it...

• but...relies on strict modularity and contracts in SPARK...so “whole program” or “inter-procedural” proof is never required...
Contents

• TweetNaCl? Why bother?

• Goal and not goals...

• Proof

• Performance

• Further work
Performance

• Test setup:
  • HiFive 1 Rev B board
  • SiFive FE310 32-bit RISC-V
  • GNAT Community 2020 (GCC 8.4.1) for both C and Ada
  • Test an Ed25519 “Sign” operation on a 256 byte message using both SPARKNaCl and TweetNaCl.
Expectations...

- Why SPARK might be slower than C...
  - Functional programming style (and first-class array types) implies return-by-copy for function return values, which can be large blocks of data. Therefore: slow!
  - TweetNaCl only does two normalization calls after “*” on GF type, but this cannot be proven in SPARKNaCl which does three normalization calls.
  - SPARKNaCl sanitizes local variables before return from subprograms. TweetNaCl does not.
Expectations...

• Why SPARK might be faster than C...

  • SPARK is proven to be free of “runtime errors”, so can be compiled with no dynamic type checking – just like C!

  • SPARK has many intrinsic properties that make it amenable to optimization: no aliasing, no undefined behaviour, no side-effects in expressions etc.

  • So... Let’s experiment with several of the standard optimization levels...
Baseline results

- Numbers are Millions of CPU cycles, so *smaller is better.*

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>C</th>
<th>SPARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241.83</td>
<td>198.03</td>
</tr>
<tr>
<td>1</td>
<td>97.65</td>
<td>98.03</td>
</tr>
<tr>
<td>2</td>
<td>84.99</td>
<td>93.86</td>
</tr>
</tbody>
</table>
Baseline results

• SPARK wins at –O0! What happened?

• Inlining of expression functions and Intrinsic Shift/Rotate functions work properly in SPARK at –O0...

• Jason Donenfeld’s “Carry-and-Reduce” code from WireGuard is a bit faster (and easier to prove...)

• “Return-Slot Optimization” (even at –O0) removes the cost of functional return-by-copy in many cases.
Improvement

- The golden rule: improve performance, but *never break the proof!*
  - All improvements retain 100% completeness and automation of the proof.

- Proof actually stops me making dumb mistakes (e.g. making the code faster, but introducing a type-safety bug...)

- Proof can *suggest* or *guide* improvements (more of which later...)
  - (Optional skip to slide 75 here)
Round 1...

- Not enough time to go over all the optimizations...

- In the order I discovered them...
  
  - Optimal initialization of large objects (using proof rather than data-flow analysis)
  
  - Manual partial redundancy elimination.
  
  - Manual unroll of "*" inner loop
  
  - Removal of array "slices" in SPARK Code
  
  - Application of these changes to TweetNaCl as well as SPARK to get a “level playing field...”
Round 1 Results

- Numbers are Millions of CPU cycles, so smaller is better.

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Original C</th>
<th>Revised C</th>
<th>Original SPARK</th>
<th>Revised SPARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241.83</td>
<td>175.74</td>
<td>198.03</td>
<td>181.74</td>
</tr>
<tr>
<td>1</td>
<td>97.65</td>
<td>62.32</td>
<td>98.03</td>
<td>69.96</td>
</tr>
<tr>
<td>2</td>
<td>84.99</td>
<td>51.00</td>
<td>93.86</td>
<td>57.82</td>
</tr>
</tbody>
</table>
Round 2...

• Observation: TweetNaCl stores a 256-bit integer using 16 64-bit “digits” (so 1024 bits) to accommodate the worst-case range of the digits during a “*” operation.

• This makes for compact code, but

• ...is wasteful of storage...

• All mathematical operations on digits are 64-bit integer, which is really slow on 32-bit RISC-V (6 instructions for a multiplication instead of 1...)

3
Round 2...

- Observation 2: We have proved that 64 bits are required to accommodate the intermediate result of a "*" operation.

- BUT... Subsequent operations can be done in 32-bit arithmetic (e.g. the second call to “carry and reduce” can be all 32-bit...) and we can store a GF value in exactly 256 bits with no waste.

  - Assignments are faster. Possibly better performance from data cache too...

  - Use the proof system and predicated types to prove that all these transformations are OK...

  - I have NOT even tried to port these changes to the C code...
Round 2 Results

- Numbers are Millions of CPU cycles, so *smaller is better.*

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Original C</th>
<th>Revised C</th>
<th>Original SPARK</th>
<th>Revised SPARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241.83</td>
<td>175.74</td>
<td>198.03</td>
<td>167.48</td>
</tr>
<tr>
<td>1</td>
<td>97.65</td>
<td>62.32</td>
<td>98.03</td>
<td>32.60</td>
</tr>
<tr>
<td>2</td>
<td>84.99</td>
<td>51.00</td>
<td>93.86</td>
<td>27.43</td>
</tr>
</tbody>
</table>
Round 3...

- Introduce –O3 (but with explicit No_Unroll pragma applied to all loops)
- Introduce –Os
- Try RV32IMC (Compressed) instruction set.
  - Smaller code size, so possibly better I-Cache hit rate?
  - Penalty for branch to mis-aligned instruction, though (on the FE310 core...)
- So, try RV32IMC but with forced 4-byte alignment on all basic block headers.
Round 3 Results

- Numbers are Millions of CPU cycles, so *smaller is better*.

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Original C</th>
<th>Revised C</th>
<th>Original SPARK</th>
<th>Revised SPARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241.83</td>
<td>175.74</td>
<td>198.03</td>
<td>174.47</td>
</tr>
<tr>
<td>1</td>
<td>97.65</td>
<td>62.32</td>
<td>98.03</td>
<td>32.54</td>
</tr>
<tr>
<td>2</td>
<td>84.99</td>
<td>51.00</td>
<td>93.86</td>
<td>41.10</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>25.69</td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td>30.52</td>
</tr>
</tbody>
</table>
Round 3 – Summary

• Best performance: 25.69 Million cycles at –O3 and RV32IMC_align_4

• Smallest worst-case stack usage: 2512 bytes at –Os

• Smallest code size: 18280 bytes at –Os (for the whole library)
Round 4...

- Manual loop fusion in “+” and “-” functions with first “Car” operation on GF.

- Narrowing of integer “*” in multiplication of GF digits. (If all digits are always 16-bit unsigned, then multiplying two digits is always OK in 32-bit arithmetic, right?)

  - Proof system says “Yes”...

- Unroll first iteration of “*” on GF to avoid double-initialization of the digits of the intermediate result.
Round 4 Results

- Numbers are Millions of CPU cycles, so *smaller is better.*

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Original C</th>
<th>Revised C</th>
<th>Original SPARK</th>
<th>Revised SPARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241.83</td>
<td>175.74</td>
<td>198.03</td>
<td>109.20</td>
</tr>
<tr>
<td>1</td>
<td>97.65</td>
<td>62.32</td>
<td>98.03</td>
<td>30.21</td>
</tr>
<tr>
<td>2</td>
<td>84.99</td>
<td>51.00</td>
<td>93.86</td>
<td>24.52</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>23.58</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>27.99</td>
</tr>
</tbody>
</table>
Round 5

• New compiler! GNAT Community 2021 is GCC 10.3.1 and incorporates improvements to Return-Slot Optimization, so RSO gets enabled for more function calls in SPARK code...

  • Thanks to Eric Botcazou at AdaCore for this work...

  • ...and any other improvements that come along...
Round 5 Results

- Numbers are Millions of CPU cycles to form an Ed25519 Signature of a 256 byte message, so *smaller is better.*

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Original C</th>
<th>Revised C</th>
<th>Original SPARK</th>
<th>Final SPARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241.83</td>
<td>175.74</td>
<td>198.03</td>
<td>110.20</td>
</tr>
<tr>
<td>1</td>
<td>97.65</td>
<td>62.32</td>
<td>98.03</td>
<td>30.06</td>
</tr>
<tr>
<td>2</td>
<td>84.99</td>
<td>51.00</td>
<td>93.86</td>
<td>24.37</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>23.22</td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td>27.62</td>
</tr>
</tbody>
</table>
Observations

- Optimization comes in many flavours:
  - Language-specific (e.g. array slices and RSO in SPARK)
  - Micro-architecture specific (RV32IM or RV32IMC with or without instruction alignment?)
  - Algorithmic (with proof support)
  - “Back porting” of optimizations (such as PRE and loop unrolling) so the benefit of them appears at –O0...
Observations

• It’s very hard to predict what you’re gonna get... No substitute for measurement and experimentation...

• “Proof driven optimization” is great. It stops me making dumb mistakes and proves that transformations are semantics-preserving...

• For example, operator narrowing and storage compression of the GF type and its operators.
Observations

• Does your project still compile at –O0? Why? Optimization of SPARK is semantics-preserving!

• Loop invariants are still too hard to find for anything that’s remotely non-trivial...

• Reproducibility of results with SMT provers is poor...
  • New version of prover? Might break proof... 😞
  • Prover timeouts are a terrible idea! 😞
Contents

- TweetNaCl? Why bother?
- Goal and not goals...
- Proof
- Performance
- Further work
A few ideas...

- Do all the performance analysis again on an RV64 target.
- Do it all again using LLVM.
- Submit bug reports against CVC4, Z3 etc. for all the unproven VCs... Perhaps CVC5 will prove the whole thing alone?
- Prove that two applications of “Car” following “*” really is all you need. (Hint: this has been done in Coq so it must be possible, but can it be automated?)
- Verification of “Constant Time” property using Info-flow analysis of program-dependence graph. TBD...
Resources

- SPARKNaCl

  https://github.com/rod-chapman/SPARKNaCl

- Long explanations of performance optimization:

  https://blog.adacore.com/doubling-the-performance-of-sparknacI-again

- GNAT/GCC and SPARK Toolsets,

  https://www.adacore.com/download/more