SPARKSkein

A Formal and Fast Reference Implementation of the Skein Hash Algorithm

Roderick Chapman[1], Eric Botcazou[2], Angela Wallenburg[1]

[1]Altran Praxis Limited, Bath, UK. [2]AdaCore, Paris, France.

rod.chapman@altran-praxis.com

Background

Skein is a cryptographic "hash" algorithm - one of the basic building blocks of integrity and authentication in digital communications. Skein is one of the final five candidate algorithms in competition to become the new international standard hash algorithm "SHA-3".

SPARK has a truly unambiguous semantics, making it a very portable language. SPARK verification tools offer a full-blown implementation of Hoare-Logic style verification, supported by both an automatic and an interactive theorem prover. There seems to be a belief that "formal is slow" in programming languages, thus justifying the continued use of low-level and type-unsafe languages like C in anything that is thought to be in any way real-time or performance critical. This work aims to provide evidence to refute this view.

This poster describes SPARKSkein - a new reference implementation of the Skein algorithm.

Aims

This work aimed to prove that a "low-level" hash algorithm like Skein could be realistically implemented in SPARK. The goals of the implementation were as follows:

Readability – Strike a balance between readability and performance. The code should be "obviously correct" to anyone familiar with the Skein specification and/or the existing C reference implementation.

Portability – Single code-base that was portable and correct on all target machines of any word-size and endian-ness.

Performance – SPARK code close to or better than the existing C reference implementation.

Formality – Prove at least type-safety (i.e. "no exceptions") on the SPARKSkein code.

Empirical - Make the experiment *empirical* in that all the code, data, and tools are freely available to the scientific community.

Method

1. Implement Skein in SPARK, based on mathematical Skein specification and existing C reference implementation.
2. Establish full proof of type-safety using the SPARK tools.
3. Test for correctness, back-to-back against C implementation, using designers' reference test vectors.
4. Structural "white box" coverage test.
5. Portability test on a variety of architectures and operating systems.
6. Performance test, using Skein designers' performance testing approach.
7. Improve GCC compiler to address any performance issues with SPARK, depending on results of step 6.
8. Report results.

Results - Implementation

Mostly easy. SPARK's mathematical semantics isolates meaning of code from low-level representation issues. Same code "just works" on all machines, with *no* macros, "ifdefs" or conditional compilation.

"Low level" functions like shift, rotate, "xor" are *very efficient* in SPARK using GCC. These "function calls" are *intrinsic* so emit just a single instruction in the generated code on 64-bit machines.

Results - Proof

A Bug in C, Detected by SPARK...

One Verification Condition (VC) for the "Finalization" algorithm would not prove. The code computes the number of bytes of output required from the requested number of bits:

```
Byte_Count := (Hash_Bit_Len + 7) / 8;
```

where the "+" is "mod 2^{64}" and the "/" is integer truncating division. The "Byte_Count" is then used to compute how many cipher block chaining mode iterations are required to compute the required number of blocks of output.

SPARK generates a VC to show that this loop must iterate at least once (or else *no* output would be generated), thus:

```
Hash_Bit_Len >= 1 and
Hash_Bit_Len <= 2^64 - 1
 ->
((Hash_Bit_Len + 7) mod 2^64) / 8 > 0 .
```

Discovery. This VC is *not true* for very large values of Hash_Bit_Len (near 2^{64}, the "+ 7" overflows to produce a number near 0, which *is zero* when divided by 8!)

Root cause. This bug exists in the C implementation, and was naively copied into the initial SPARK code.

Impact. Initializing the algorithm with a request for very large hash output could result in *no* output, with the algorithm returning an *undefined* block of memory. Unlikely, *unless* you had previously computed the hash output length, and that computation suffered an *underflow*.

Correction. Very simple in SPARK – either just add a special case in the loop iteration calculation, or eliminate the problem entirely with

```
subtype Hash_Bit_Length is U64 range 0 .. U64'Last - 7;
```

After that, proof of type safety was successfully completed.

Main points. Finding the correct invariants is still far too hard! They need to be like Three Bears' porridge – not too weak, not strong, but "just right". In SPARKSkein, the invariants involve many *non-linear* inequalities (e.g. "mod" and "/" operators), so our tools struggle with them.

Proof Summary:

```
Total VCs by type:
                      ----------Proved By Or Using-----------
                Total  Examiner Simp(U/R)  Checker Review False Undiscgd
Assert or Post:    65       22      35          8      0     0        0
Precondition check:21        0      12          9      0     0        0
Check statement:   31        0      26          5      0     0        0
Runtime check:    244        0    243(   2)    1      0     0        0
Refinement VCs:     6        2      4(   4)    0      0     0        0
Inheritance VCs:    0        0      0          0      0     0        0
================================================================================
Totals:           367       24    320(   6)   23      0     0        0
% Totals:                    7%    87%(   2%)   6%     0%    0%       0%
===================== End of Semantic Analysis Summary =========================
```

What about SMT Provers?

23 VCs out of 367 (6.26%) required proof with the Checker – our interactive prover. These were almost all VCs with non-linear conclusions. Could a modern SMT solver do better?

Yes...Using Jackson's SPARK-to-SMTLib translator (ViCToR), both Z3 and Alt-Ergo are able to prove all 23 remaining VCs, resulting in 100% automation for this proof.

Note: the SPARK Programming Language is not sponsored by or affiliated with SPARC International Inc and is not based on the SPARC® architecture

Results – Reference Test Vectors and Portability

Official Skein test vectors from the specification. Initial attempt failed!

Why? One rotation constant had been entered incorrectly when typing in the code. "34" instead of "43".

Type system declared that these constants had to be in range 1 .. 63. so this is "type safe" but still wrong.

After correction: all test vectors pass on all test machines, including 32- and 64-bit machines, little- and big-endian architectures, and many operating systems.

Moral: code review is still important!

Results – Structural Coverage Test

Reference test vectors plus a small number of "white box" additional tests were derived. This results in 100% statement and branch coverage.

Conclusion: not much! Coverage analysis is only really interesting when you *don't* get 100%.

Results - Performance

Summary: With contemporary versions of GCC and common compiler options, SPARK offers better or equivalent performance when compared with the C reference implementation.

Environment: reproduce the Skein designers' performance testing program in SPARK, to enable a like-for-like performance comparison between the C and SPARK implementations. GNAT Pro is the GCC build maintained by AdaCore – it is highly stable, and subject to a far more rigorous quality-control that the standard GCC builds from FSF. GNAT Pro compiles both SPARK and C, so use we can use the same compiler back-end and various common optimization settings to assess performance.

Test machine: Intel Core i7-860 @ 2.8GHz running 64-bit Debian GNU/Linux.

Options tested: -gnato (*enable* all dynamic checks, including overflows), -gnatp (*disable* all dynamic checks), -gnatn (*enable* inlining of subprograms), plus optimization levels 0, 1, 2, and 3.

Round 1 Analysis – GNAT Pro 6.3.2 (GCC 4.3.5)

- At –O0, both languages are "deliberately awful" – a side-effect of GCC's design.
- At –O1, SPARK outperforms C owing to significantly more aggressive inlining in the Ada compiler front-end. Inlining leaves *no* residual overhead. Observation – no need in SPARK to sacrifice decomposition and readability for performance.
- At –O2, C leads by a small margin owing to a few missed optimization opportunities in the SPARK code.
- At –O3, margin remains the same, but better overall owing to auto loop unrolling in both languages.

Round 2 Analysis – GNAT Pro 6.4.0 (GCC 4.5.0)

- Huge improvement across the board, reflecting the new SSA-based back-end in GCC 4.5.0. –O0 is far less awful!
- Basic pattern the same, except at –O0 where SPARK leads.

Round 3 Analysis – GNAT Pro 6.4.1 (GCC 4.5.2)

- As a result of Round 2 results, we implemented various improvements to the Ada "middle end" to produce more "optimization friendly" IL for the back-end.
- Same pattern, but SPARK now offers identical performance at –O3.
- These improvements have been integrated into subsequent GNAT Pro products and the trunk of FSF GCC builds.

Table 1: Performance Results Clocks per byte hashed (Lower numbers are better)	Round 1 GNAT Pro 6.3.2 (GCC 4.3.5)		Round 2 GNAT Pro 6.4.0 (GCC 4.5.0)		Round 3 GNAT Pro 6.4.1 (GCC 4.5.2)	
Compiler Options	SPARK	C	SPARK	C	SPARK	C
-O0 –gnato	213.9	N/A	71.1	N/A	70.6	N/A
-O0 –gnatp	207.9	172.3	69.9	96.5	69.7	96.4
-O1 –gnatp	27.6	37.7	22.2	37.0	22.2	37.0
-O1 –gnatp –gnatn	26.8	37.7	20.7	37.0	20.5	37.0
-O2 –gnatp –gnatn	25.5	24.7	20.2	19.7	20.0	19.7
-O3 –gnatp –gnatn	20.4	20.1	13.4	12.3	12.3	12.3

Further Work

Further work could cover improving proof automation, completeness and performance in the SPARK tools, counter-example finding, and our interface to SMT-based provers.

For GCC, we might be able to identify additional SPARK-specific optimization opportunities.

Challenges

The SPARK reference implementation of Skein could be used as a "challenge problem" for other Ada-based verification tools.

The C reference implementation forms a challenge for C verifiers.

Conclusions

It can be done! Formal code can be fast, readable and portable.

Readability and Performance do not seem to conflict, given strength of compilation technology.

Room-for-Improvement in SPARK Simplifier, but SMT tools do much better.

Much help needed with non-linear VC structures, and with finding the "just right" loop invariants.

Obtaining SPARKSkein

SPARKSkein code, test programs, and proofs are available from www.skein-hash.info

GNAT and SPARK tools are available for the scientific community from libre.adacore.com. Industrial users should contact sales@adacore.com

Full paper: *SPARKSkein: A Formal and Fast Reference Implementation of Skein.* Roderick Chapman, Eric Botcazou, and Angela Wallenburg. Proceedings of the 14th Brazilian Symposium on Formal Methods (SBMF), Sao Paulo, Brazil, September 2011. Springer-Verlag LNCS Vol. 7021, pp. 16–27. Also available from the authors.

Acknowledgements

The authors would like to thank Doug Whiting and Jesse Walker of the Skein design team. Doug patiently answered questions about the C implementation. Jesse offered valuable comments on an early draft of the SPARKSkein paper.