# SPARKSkein: A Formal and Fast Reference Implementation of Skein

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

[1] Altran UK Limited, 22 St Lawrence Street
Southgate, Bath BA1 1AN, U.K.
[2] AdaCore, 46 rue d'Amsterdam,
75009 Paris, France.
rod.chapman@altran.com
botcazou@adacore.com, angela.wallenburg@altran.com

**Abstract.** This paper describes SPARKSkein[1] – a new reference implementation of the Skein cryptographic hash algorithm, written and verified using the SPARK language and toolset. The new implementation is readable, completely portable to a wide-variety of machines of differing word-sizes and endian-ness, and "formal" in that it is subject to a proof of type safety. This proof also identified a subtle bug in the original reference implementation which persists in the C version of the code. Performance testing has been carried out using three generations of the GCC compiler. With the latest compiler, the SPARK code offers identical performance to the existing C reference implementation. As a further result of this work, we have identified several opportunities to improve both the SPARK tools and GCC.

**Keywords:** Skein, Hash, SHA-3, SPARK, Theorem Proving, GCC, Optimization, Verification, Security.

## 1 Introduction

This paper describes SPARKSkein – a new reference implementation of the Skein cryptographic hash algorithm [1], written and verified using the SPARK[2] language and toolset.

This work started out as an informal experiment to see if a hash algorithm like Skein could be realistically implemented in SPARK. The goals of the implementation were as follows:

- Readability. We aimed to strike a reasonable balance of readability and performance. The code should be "obviously correct" to anyone familiar with the Skein specification and/or the existing C reference implementation.

---

[1] The full version of this paper appears here: http://link.springer.com/chapter/10.1007/978-3-642-25032-3_2

[2] The SPARK Programming Language is not sponsored by or affiliated with SPARC International Inc and is not based on the SPARC® architecture.

- Portability. SPARK has a truly unambiguous semantics, making it a very portable language. Therefore, we aimed for a single code-base that was portable and correct on all target machines of any word-size and endian-ness.
- Performance. We hoped that the performance of the SPARK code would be close to or better than the existing C reference implementation. The conjecture here is that code that is both correct and type-safe can also be fast. If we failed on the performance front, then we hoped to at least understand *why* as a way of promoting further work on compiler optimization for SPARK.
- Formality. The SPARK verification tools offer a full-blown implementation of Hoare-Logic style verification, supported by both an automatic and an interactive theorem prover. We aimed to prove at least type-safety (i.e. "no exceptions") on the SPARKSkein code. 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.
- Empirical. We aimed to make the experiment *empirical* in that all the code, data, and tools are freely available to the scientific community.

Could we do it? Could we produce code that is formal, provable, readable, portable *and* fast?

## 2 Skein

Skein [1] is one of the candidate algorithms in the third and final round of the competition to design the future standard hash algorithm that will become known as SHA-3 [12]. Skein is designed for cryptographic strength, portability, and performance, although it is particularly designed for efficiency on 64-bit little-endian machines, such as x86_64, which dominate in desktop computing. Skein is fully defined in [13] using an algorithmic specification accompanied by proofs of a number of key security properties.

## 3 SPARK

This section provides a brief overview of SPARK and its capabilities. SPARK-aware readers may skip ahead.

SPARK is a contractualized subset of Ada. The contracts embody data- and information-flow, plus the classical notions of pre-condition, post-condition and assertions in code. The language is designed to have a wholly unambiguous semantics – there are *no* unspecified or undefined language features in SPARK – meaning that static analysis can be both fast and sound. The contract language is designed for wholly static verification through the generation of Verification Conditions (VCs) and the use of theorem proving tools. SPARK is well-known in the development of safety-critical systems, but is also being used in some high-grade secure applications, where its properties and verification system have proven useful.

As a subset of Ada, SPARK can be compiled by any standard Ada compiler. The contracts look like comments to an Ada compiler, but are an inherent part of the language as far as the verification tools are concerned. The unambiguous semantics also means that a SPARK program has the same meaning regardless of choice of compiler or target machine – endian-ness, word-size, and so on just don't matter at all.

There are four main tools. The *Examiner* is the main static analysis engine – it enforces the language subset and static semantics, and then goes on to perform information-flow analysis [2]. The Examiner includes a Verification Condition Generator (VCG) – essentially an implementation of Hoare's assignment axiom – that produces VCs in a logic suitable for an automated theorem prover called the *Simplifier*[3]. This is an heuristics-driven automated prover. For VCs that the Simplifier can't prove, we have the *Checker* – an interactive proof assistant based on the same core inference engine. Finally, a tool called *POGS* collates and reports the status of each VC for the entire program.

Further details about SPARK can be found in the SPARK textbook [3] and the Tokeneer on-line tutorial [4]. The GPL edition of the SPARK toolset is freely available under the terms of the GPL [5].

## 4 Implementing SPARKSkein

The current implementation delivers the main Skein hash algorithm with a 512-bit block-size. For the purposes of this exercise, the other block-sizes and uses of Skein were not relevant.

The coding was straightforward. The main challenge was in understanding the Skein specification and the existing C implementation in sufficient detail to produce a correct SPARK implementation.

One challenge arises in laying out the structure of the Skein "Tweak Words" record. In the C implementation, these are just an array of two 64-bit words, but in SPARK we chose to declare this as a record type with named fields for ease of reading. This means that the layout of this record type has to be different on big-endian and little-endian machines. To do this, a representation clause specifies the bit-numbering required, but depends on the constant System.Default_Bit_Order to get the correct order and layout for the target machine.

To illustrate the difference in coding style, consider the initialization of the hash context in Skein_512_Init. In the C reference implementation, this looks like a function call:

```
Skein_Start_New_Type(ctx,CFG_FINAL);
```

Closer inspection, though, reveals that this is actually a pre-processor macro:

[3] Not to be confused with Greg Nelson's better-known Simplify prover.

```
#define Skein_Start_New_Type(ctxPtr,BLK_TYPE)
{ Skein_Set_T0_T1(ctxPtr,0,SKEIN_T1_FLAG_FIRST |
SKEIN_T1_BLK_TYPE_##BLK_TYPE); (ctxPtr)->h.bCnt=0; }
```

This, in turn, refers to the macro Skein_Set_T0_T1. The whole thing expands out into:

```
{ { {(ctx)->h.T[0] = ((0));};
{(ctx)->h.T[1] = (((((u64b_t) 1 ) << ((126) - 64)) |
(((u64b_t) (( 4))) << ((120) - 64)) | (((u64b_t) 1 ) <<
((127) - 64))));}; };
(ctx)->h.bCnt=0;
};
```

which is actually 3 assignment statements, with the various shifting/masking constants picked to get the correct endian-ness for the target machine.

In SPARK, this code becomes a procedure, which treats the Context as a record object that can be assigned to. The whole thing comes out as two assignment statements:

```
Ctx.Tweak_Words :=
   Tweak_Value'(Byte_Count_LSB => 0,
                Byte_Count_MSB => 0,
                Reserved       => 0,
                Tree_Level     => 0,
                Bit_Pad        => False,
                Field_Type     => Field_Type,
                First_Block    => First_Block,
                Final_Block    => Final_Block);
Ctx.Byte_Count := 0;
```

which we argue is more readable. All the complexity of the endian-ness and the layout of the record are hidden in the representation clause, and the compiler takes care of generating the required shifting and masking instructions to construct the correct value.

SPARK naturally supports nesting of subprograms (as in all Pascal-family languages) so this allows a natural top-down decomposition of the main operations into local procedures. This decomposition aids readability, but has a negligible impact on performance, assuming a compiler is able to inline the local procedures.

As far as possible, the implementation follows the structure of the reference C implementation, so anyone familiar with that version should be able to read and follow the SPARK code.

We also added a package Skein.Trace that produces debugging output in *exactly* the same format as the functions in the C code's skein_debug.c, so automatic comparison of debug output would be possible. This proved very useful in testing the output of the SPARK version side-by-side with the C.

# 5 Verification of SPARKSkein

We have verified SPARKSkein in various ways: using the SPARK static verification tools, testing using the published reference test vectors, structural coverage analysis, testing for portability on as many differing machines that we could lay our hands on, and performance testing. These sections summarize the results of these activities.

## 5.1 Static Verification and Proof

The SPARKSkein code passes all the analyses and verification implemented by the Examiner with no errors. Additionally, we generated VCs for *type-safety*. This means we prove that a program could never raise an exception at run-time through the failure of a type-safety check, such as a buffer overflow, division-by-zero, numeric overflow and so on. The proof of type-safety essentially proves that a program remains in a well-defined state and would never raise exceptions for any possible input data that meets the stated top-level pre-conditions. A benefit of type-safety proof is that it can detect subtle corner cases such as this.

The implementation produces 367 verification conditions, of which 344 (93.7%) are proven automatically by the Examiner or Simplifier. Of these 344, 6 require the insertion of user-defined lemmas into the theorem-prover. Such user-defined lemmas must be subject to careful review, or validation with the Checker, as they have the potential to introduce unsoundness into the proof. The remaining 23 verification conditions are proved using the Checker, requiring some human assistance.

**Prover Says No – a Bug is Discovered**
The subprogram Skein_512_Final caused some problems, and led to the discovery of a subtle corner-case bug.

The finalization algorithm uses the number of bits of hash requested to compute how many bytes of hash are required, and therefore how many blocks of data are needed. A loop then iterates to generate the required number of blocks. This loop has to iterate at least once, or else no output would result. This requirement was expressed as a type-invariant in SPARK in that the number of output blocks has to be at least one.

The offending fragment of code is:

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

where the "+" operator is modulo $2^{64}$.

The need to have at least one block comes out as a VC with conclusion:

$$((Local\_Ctx.H.Hash\_Bit\_Len + 7) \bmod 2^{64}) / 8 > 0$$

which the theorem prover refused to prove for our first implementation – most obviously because it's not true!

The problem is that if the requested Hash_Bit_Len set by Skein_512_Init is sufficiently large (i.e. near $2^{64}$), then the "+ 7" overflows to be near zero which, when divided by 8, *is* zero.

This bug is unlikely to happen in reality, based on the assumption that no-one would ask for a hash nearly $2^{64}$ bits long, but it does illustrate the theorem-prover's ability to sniff out such subtle corner cases that typically elude testing, review or other forms of verification.

The correction is simple enough – we simply limit the range of acceptable hash bit lengths to a maximum of $2^{64} - 8$, so the overflow is avoided. This is encoded in SPARK as a subtype called Hash_Bit_Length, declared in the package specification and then used as the parameter for Skein_512_Init.

In the C reference implementation, this bug persists and the code produces no blocks of output (returning a pointer to an undefined block of memory) for this case.

**Reflections on the Proof**

The 344 automatically discharged proofs were harder (and slower) than expected. This owes to the prevalence of "modulo N" arithmetic in the VCs. Crypto algorithms tend to do most things using "unsigned" or (in SPARK terminology) "modular" types, which exhibit modular operators like "+" that wrap-round. In the world of proof, this generates VCs that have "mod N" appended to the end of nearly every expression. We also chose to index array types with modular integers, so even innocuous operations like incrementing an array index variable resulted in a "mod N" appearing in the VC. Theorem-provers are notoriously poor with such things - ours included – so the 93.7% of VCs proved automatically is acceptable but offers some room for improvement.

The 23 VCs that remained undischarged by the Simplifier were difficult to prove in the Proof Checker. The main problem is finding a sufficiently strong pre-condition or loop-invariant for the offending code. These tend to have a Goldilocks-like tendency – they mustn't be too strong, mustn't be too weak, but "just right." When the Simplifier fails to prove a VC, it is not always clear why. Perhaps there really is a bug in the code, and the VC has a counter-example? Perhaps the VC really is true, but the Simplifier is just not clever enough to find the proof? Perhaps the VC is unprovable because a pre-condition or invariant is too weak? This last case is particularly annoying – a long session with the Checker can result in merely finding that a VC isn't provable at all. It then isn't always clear exactly what change to the invariant might help. These weaknesses will be considered further in section 6.

**A taste of SMT**

In parallel with this work, Jackson [6] has produced ViCToR – a tool that translates SPARK VCs into SMTLib so that they can be processed by contemporary SMT-based solvers. Encouragingly, we have found that both Z3 [7] and Alt-Ergo [8] are capable of proving *all* of the VCs arising from SPARKSkein automatically, with Z3 offering by far the better runtime performance at present. We are currently working on integrating the SPARK tools with ViCToR to offer the option of using the Simplifier, one or more SMT-based prover(s), or some combination of both.

### 5.2 Reference Test Vectors

The test case in the main program “spec_tests” runs the 3 reference test cases given in version 1.2 of the Skein specification.

The first attempt to run this test case failed – the resulting hashes were wrong, illustrating that type-safe code is not necessarily correct. This problem was traced to a simple typing error in the value of the shifting constant R_512_6_3 which had the incorrect value 34 instead of 43.

With that correction in place, the results were as in the Skein specification. No further defects were discovered.

### 5.3 Platform Testing

The “spec_tests” program has also been added to AdaCore’s regression test suite for GCC. This suite is run nightly on all architectures (both big-endian and little-endian) and operating systems supported by AdaCore.

Target architectures and operating systems include 32-bit x86 (Windows, Linux, FreeBSD, and Solaris), x86_64 (Windows, Linux, Darwin), SPARC (32- and 64-bit Solaris), HP-PA (HP Unix), MIPS (Irix), IA64 (HP Unix, Linux), PowerPC (AIX), and Alpha (Tru64).

The test passes on all platforms.

### 5.4 Coverage Analysis

The main program “covertest” is designed to exercise boundary values and structural coverage of the hash algorithm. In particular, these test cases are designed to exercise the Skein_512_Update code with various combinations of data blocks of length less than 1 block, exactly 1 block, between 1 and 2 blocks, exactly 2 blocks and more than 2 blocks. This case also tests various sequences of these blocks to cover the cases where a short block results in data being “left over” in the hash context buffer.

This program can be compiled with GCC’s coverage analysis options switched on, and analysed with gcov. The project file “covertest.gpr” builds the program with these options enabled. A single run of “covertest”, followed by “gcov skein.adb” shows 99.7% statement coverage, with a single warning for exactly 1 uncovered line of code. This line is a type declaration which has no object code associated with it, so this must be a false-alarm from gcov.

## 5.5 Performance Testing

Achieving acceptable performance, but without sacrificing readability and portability, was a major goal of this experiment. This section reports our findings, comparing the performance of the SPARK code against the existing reference implementation in C.

There is a view that anything "formal" must be "slow." Languages like Ada with their run-time type checking are often criticized for being "slower than C" and therefore not appropriate for time-critical code such as this. Is this really true?

One conjecture we sought to investigate is that type-safe SPARK code should also be fast. SPARK has several properties that make it suitable for hard real-time programming. Furthermore, SPARK code *should* be amenable to more aggressive optimization than other imperative languages. In particular, in SPARK:

- Functions are always *pure* – they have no side-effects.
- There is absolutely no *aliasing* via pointers or names of variables.
- If type-safety has been proven statically (as in this case), then we can safely compile with all runtime checking disabled and more optimistic assumptions about data validity – hopefully making the generated code smaller, faster and simpler.

These properties *should* be taken advantage of by a compiler – where, for example, an optimization pass could make more optimistic assumptions about SPARK code than it could for C. Is this really true? Can a current version of GCC actually find and exploit these properties of SPARK?

**Method**

These tests were run on a standard PC with an Intel core i7 860 processor running at 2.8GHz. The machine was running 64-bit GNU/Linux (Debian 5.0.5). We chose a 64-bit OS (and compiler) since Skein is designed to perform well on such machines.

The test case "perftest" was written to mimic the testing strategy and performance measurement approach of the "skein_test" program that is supplied with the reference implementation. In this way, we hoped to get results that were reasonably comparable for the C and SPARK implementations.

We also chose to compile the C and the SPARK with the *same* compiler, in this case GNAT Pro 6.3.2 – a stable derivative of GCC 4.3.5 that compiles both SPARK and C through the same back-end. We compiled the C code at various levels of optimization and took the results for Skein_512 hashing a block of 32768 bytes as our base-line for comparison.

When compiling SPARK, GCC offers some additional options that we can take advantage of, so we exercised these to see the effect. In particular, we used the following Ada-specific options:

-gnato – this compiles with *all* the run-time type checking required by the Ada LRM, including checks for arithmetic overflow. This typically generates the slowest code, so was useful as a base-line for the SPARK code.

-gnatp – this option suppresses *all* run-time type checks in the generated code. This is reasonable for us, since we had, of course, already proved that the code was type-safe – effectively showing that run-time checks could never fail. This gives a run-time and code-generation model close to that of C, so we expected comparable performance of the SPARK and the C with –gnatp at the same level of optimization.

-gnatn – enables inlining of subprograms in the back-end of the compiler.

Results were measured in clocks (measured by the x86's rdtsc instructions) per byte hashed, as per the reference skein_test program. Lower numbers indicate better performance:

| GNAT Pro 6.3.2 (GCC 4.3.5) | | |
|---|---|---|
| **Options** | **SPARK** | **C** |
| -O0 -gnato | 213.9 | N/A |
| -O0 -gnatp | 207.9 | 172.3 |
| -O1 -gnatp | 27.6 | 37.7 |
| -O1 -gnatp -gnatn | 26.8 | 37.7 |
| -O2 -gnatp -gnatn | 25.5 | 24.7 |
| -O3 -gnatp -gnatn | 20.4 | 20.1 |

**Going further – GNAT Pro 6.4.0w**
We then re-ran the experiment with a GNAT Pro wavefront (6.4.0w – a derivative of GCC 4.5.0 built on the 28th July 2010). GCC 4.5.0 includes significant improvements across all phases of the back-end, so we expected to see improvement for both C and SPARK. The results were:

| GNAT Pro 6.4.0w | | |
|---|---|---|
| **Options** | **SPARK** | **C** |
| -O0 -gnato | 71.1 | N/A |
| -O0 -gnatp | 69.9 | 96.5 |
| -O1 -gnatp | 22.2 | 37.0 |
| -O1 -gnatp -gnatn | 20.7 | 37.0 |
| -O2 -gnatp -gnatn | 20.2 | 19.7 |
| -O3 -gnatp -gnatn | 13.4 | 12.3 |

**Analysis – GNAT Pro 6.3.2 vs GNAT Pro 6.4.0w**
Coming from compilers based on back-ends separated by two complete cycles of GCC development (roughly two years), these results are significantly different. It is, however, possible to identify a few common patterns.

First of all, the results are uniformly better with the newer compiler and, at -O1 or above, come from improved alias analysis and dead store elimination. The -O0 level is peculiar: for years, the GCC back-end had been known for its totally unoptimized code generation at this level; this was changed in GCC 4.5 and the effect is clearly visible here. This also explains why SPARK gained so much at -O0: being a more expressive language than C, its raw intermediate representation is more verbose and used to be replicated almost verbatim in the generated code at -O0, thus masking the actual merits of the code. To eliminate this old effect, we'll exclude the results at -O0 in the following comparison of SPARK and C.

SPARK is far ahead at -O1 because, even at these low optimization levels, the Ada compiler generates single-instruction inline code for bitwise operators (e.g. shifts and rotates) on scalars. Being rather conservative at these levels, the standard optimization heuristics prevent such operators from being inlined in C.

The next optimization level, -O2, essentially bridges the inlining gap, with C nudging slightly ahead of SPARK with both compilers.

Level –O3 introduces automatic loop unrolling. This is responsible for the big boost in both languages at -03. This leads to roughly equivalent performances with 6.3.2, but not quite so with 6.4.0w because other effects are exposed with the newer compiler. Specifically, it appears that the improved partial redundancy elimination in loops is more efficient on the C code. The SPARK code also suffers from slightly inferior scalarization of composite types and from too limited store/copy propagation.

Finally, it's worth noting that the big boost at -O3 can be partially retrofitted at lower optimization levels in both languages by manually unrolling the single loop in the procedure Inject_Key in the SPARK code. This loop includes an expensive "mod 9" operator, causing a pipeline stall when enclosed in a loop. Unrolling this loop "by hand" in the source code improves the performance of the SPARK code from 20.2 to 13.3 clocks per byte using GNAT 6.4.0w at –O2, for example.

**Improving GNAT Pro 6.4.1**

As a result of this analysis, we designed a number of improvements to the Ada middle-end, which translates the Ada front-end's intermediate language into that expected by the GCC back-end. In particular, these improvements generate GCC intermediate-language that is more amenable to the optimization of partial redundancies, scalarization of composite objects, and store/copy propagation.

These improvements are now included in the GNAT Pro 6.4.1 release of March 2011. With this new compiler, the results look like this:

| GNAT Pro 6.4.1, March 2011 | | |
|---|---|---|
| **Options** | **SPARK** | **C** |
| -O0 -gnato | 70.6 | N/A |
| -O0 -gnatp | 69.7 | 96.4 |
| -O1 -gnatp | 22.2 | 37.0 |
| -O1 -gnatp -gnatn | 20.5 | 37.0 |
| -O2 -gnatp -gnatn | 20.0 | 19.7 |
| -O3 -gnatp -gnatn | 12.3 | 12.3 |

**Analysis – GNAT Pro 6.4.1**
The results are essentially identical to those obtained with the 28th July 2010 wavefront, except for the -03 level where SPARK is now on par with C. The original analysis still holds. In particular, the effects of inlining and loop unrolling still dominate here.

SPARK still lags a little behind C at -O2. This comes from a couple of missed Partial Redundancy Elimination opportunities for the SPARK version, which can be traced to an application of the "Unchecked_Conversion" function on a composite object. The application of this function forces the Ada compiler to make worst-case assumptions about the resulting value, preventing a couple of subsequent optimizations from taking place. From this, we derive a simple coding rule for high-performance Ada and SPARK code: don't use Unchecked_Conversion on composite objects.

The changes made to the compiler are not specifically tuned to the Skein code or any other particular benchmarks, so they should benefit Ada programs in general.

## 6 Further Work and Challenges

This section presents a few ideas that might warrant further work or form challenges for other tools and research groups.

### 6.1 GCC

For the GNAT compiler, the results conform to the general trend observed over the years: the aggressive optimizations implemented in the GCC back-end are initially tuned to the C family of languages. A little more work is required in order to make them as effective in SPARK or Ada, and the end result is almost always generated code equally well optimized whatever the source language. Our improvements to GCC resulting from this project will appear in a future release of GCC from the FSF.

### 6.2 The SPARK Tools

For the SPARK tools, several improvements have been identified as a result of this work. Most notably, the procedure Skein_512_Update causes extremely poor performance from the Simplifier – taking nearly an hour to simplify on the Core i7 machine used for testing. We hope to identify and correct this matter in a future release of the SPARK Tools.

The proofs that require interaction with the Checker provide a rich source of examples for further improvement of the Simplifier's proof tactics, particularly in the area of modular arithmetic.

As we noted above, the potential to exploit SMT-based solvers offers a notable improvement in both performance and completeness of proof. We are currently benchmarking these provers on substantially larger programs than SPARKSkein to determine which prover (or combination of provers) offers the most benefit.

Finally, to improve the insight and feedback arising from failed proof attempts, we are actively pursuing research on counter-example finding, initially focusing on the use of answer-set programming [9] supported by SMT-based provers.

### 6.3 Comparison with Other Verification Tools

The Skein code could be used as a "Challenge Problem" for other verification tools. Given the SPARK proofs of the code, it could be used as a test-case to measure the performance and true false-alarm rate of other tools. In particular, we would like to assess the ability of other tools to rediscover the pre-conditions and invariants that proved troublesome to find by hand. Similarly, tools such as VeriFast [10] or Microsoft's VCC [11] could be used to recreate a proof of type safety for the C reference implementation.

## 7 Conclusions

Returning to our original goals, it seems the project can be judged a success. An algorithm like Skein can be written in a "formal" language like SPARK without sacrificing readability and performance. Portability can only be judged a success – a single set of sources with no macros, "ifdefs" or pre-processing gives identical results on every architecture and OS we could find. SPARK's type-safe nature allowed us to "turn up the dials" on the compiler's optimizers with confidence. The SPARK code did perform better than C at –O1, reflecting more aggressive inlining in the Ada front-end at that level. At –O2 and –O3, C pulls ahead by a small margin using compilers that were available in the middle of Summer 2010. As a result, we identified and implemented improvements in more recent GCC releases that show the SPARK code having essentially identical performance to the C.

**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 this paper.

**Obtaining SPARKSkein**
The SPARKSkein sources, test cases, and proofs are available from the Skein website [1]. The GPL editions of the SPARK Toolset and GNAT compilers are available from [5].

## References

1. Skein project homepage, http://www.skein-hash.info/
2. Carré, B., Bergeretti, F.: Data- and Information-Flow Analysis of While Programs. ACM Transactions on Programming Languages and Systems, Volume 7, Number 1, January 1985. pp 36-61.
3. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley (2003, reprinted in 2007), ISBN 978-0-321-13616-0
4. Tokeneer Discovery: A SPARK Tutorial. http://www.adacore.com/home/products/sparkpro/tokeneer/discovery/
5. SPARK GPL Edition site: http://libre.adacore.com/
6. Paul B. Jackson, Bill J. Ellis and Kathleen Sharp. Using SMT Solvers to Verify High-Integrity Programs. 2nd International Workshop on Automated Formal Methods, AFM '07, Atlanta, Georgia, USA. See also http://homepages.inf.ed.ac.uk/pbj/
7. Z3: An efficient theorem prover. Microsoft Research. http://research.microsoft.com/en-us/um/redmond/projects/z3/
8. Alt-Ergo website: http://alt-ergo.lri.fr/
9. Baral, Chitta. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003.
10. Bart Jacobs and Frank Piessens. The VeriFast program verifier**.** Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
11. VCC: A Verifier for Concurrent C. http://research.microsoft.com/en-us/projects/vcc/
12. National Institute of Standards and Technology, Computer Security and Resource Center, Cryptographic Hash Algorithm Competition: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html
13. The Skein Hash Function Family, Niels Ferguson et al, http://www.skein-hash.info/sites/default/files/skein1.2.pdf