XBX: eXternal Benchmarking eXtension for the SUPERCOP crypto benchmarking framework

Christian Wenzel-Benner[1] and Jens Gräf[2]

[1] ITK Engineering AG
Software Center 1, 35037 Marburg, Germany
`Christian.Wenzel-Benner@itk-engineering.de`,
WWW home page: `http://www.itk-engineering.de`
[2] LiNetCo GmbH
Hauptstrasse 17a, 35684 Dillenburg, Germany
`jgraef@linetco.com`,
WWW home page: `http://www.linetco.com`

Abstract. SUPERCOP [1] is a benchmarking framework for cryptographic algorithms like ciphers and hash functions. It automatically benchmarks algorithms across several implementations, compilers, compiler options and input data lengths. Since it is freely available for download the results are easily reproducible and benchmark results for virtually every computer that is capable of running SUPERCOP are available. However, since SUPERCOP is a collection of scripts for the Bourne Again Shell and depends on some command line tools from the POSIX standard in it's current form it can not run on any hardware that does not support POSIX. This is a significant limitation since small devices like mobile phones, PDAs and Smart Cards are important target platforms for cryptographic algorithms. The work presented in this paper extends the SUPERCOP concepts to facilitate benchmarking external targets. A combination of hard- and software allows for cross compilation with SUPERCOP and execution/timing of the generated code on virtually any kind of device large enough to hold the object code of the algorithm benchmarked plus some space for communication routines and a bootloader.

Key words: SUPERCOP, XBX, benchmarking, microcontroller, small device, 8-bit, hash function

1 Introduction

The design of a cryptographic algorithm is always a trade-off between security and performance. A 'good' algorithm either achieves stronger security than a 'bad' one at the same runtime and memory cost or the same level of security at lesser cost. Yet telling a 'good' algorithm from a 'bad' one is not always trivial.

Aside from spotting obvious design flaws, both security and performance are not easily quantified.

1.1 Judging Security

If a newly proposed algorithm were to be broken by applying a well-known mode of attack this would be an obvious design flaw. But if it is not vulnerable to any known attack that does not mean it is flawless. An algorithm that seems secure for a long time may suddenly be affected by a new type of attack that was not anticipated. If and when such a new attack is going to be discovered can not be determined in advance. There is however one universal rule: attacks only get better, they never get worse. The flip-side of this coin is that algorithms only get weaker, never stronger.

1.2 Judging Performance

Judging the performance of an algorithm seems trivial by comparison. Yet it is not uncommon to see different performance numbers claimed by different people for a well known standard algorithm like SHA-256 in the course of a four day conference[2]. Obviously, different implementations of the algorithm, different compilers and different target platforms result in a huge diversity of performance numbers. Which one is the 'true' performance number? A sophisticated benchmarking framework like SUPERCOP can answer this question for a given CPU, a given implementation and a given compiler. Across all these parameters there is a fastest combination of CPU, implementation and compiler that is arguably 'true' because the SUPERCOP framework is freely available and all parameters used to obtain the performance number are clearly stated in the result file. A freely available benchmarking framework that explicitly states all parameters used to obtain a performance number like SUPERCOP provides a universal rule for performance evaluation, too: implementations, CPUs and compilers only get better, they never get worse. Hence a given algorithm's performance only gets better over time.

1.3 Additional Criteria

On desktop computers and servers the size of program code and lookup tables is usually not an issue. When the implementation of a cryptographic algorithm uses table lookups on such machines it is a concern because of timing attacks[3], not because of the amount of memory required. Small devices are different, they impose severe size limitations. There are hard limits, such as a 64k address space on an 8-bit machine, and somewhat softer restrictions, like the price of a smart card that is to be manufactured several million times. In both cases, smaller is better. Implementors compete for the smallest implementation, designers for the smallest algorithm (at comparable security level). The fact that theoretical work[4] concerning memory consumption of cryptographic algorithms is being done indicates that this is an area a growing interest.

1.4 Motivation for the eXternal Benchmarking eXtension

Without a sophisticated framework different performance numbers obtained under different (sometimes not clearly stated) circumstances circulate and make it very hard to judge how well designed a given algorithm really is. The eXternal Benchmarking eXtension presented here brings the advantages of SUPERCOP benchmarking to small devices like 8-bit microcontrollers. The results obtained this way are useful to

- find the fastest algorithm for a given target platform
- find the smallest algorithm for a given target platform
- find the best (cross-)compiler for a given hardware design and algorithm (for either speed or size)
- find the best compiler settings for a given hardware design and algorithm (for either speed or size)
- select a microcontroller for a future hardware design
- compare different implementations, e.g. a proprietary and commercial assembler implementation vs. public domain c code
- design new algorithms to run well on targeted hardware platforms

2 Design Goals

The aim of XBX (as the name suggests) is to extend SUPERCOP to a new domain: benchmarking external devices as opposed to the CPU(s) inside the computer that runs the SUPERCOP framework. To extend means 'to stretch out' and when stretching something it is usually advisable to take care not to rip it in two. In order to keep what the authors of this work perceived as the core of SUPERCOP intact the design goals for XBX were defined as follows:

Goal 1 automatic testing of algorithms by a simple script invocation

Goal 2 precise performance numbers for different message lengths that reflect real world user experience

Goal 3 free source code, for every user to inspect and re-use

Goal 4 cheap, easily available hardware

Goal 5 compatibility to standard SUPERCOP algorithm interface

Goal 6 compatibility to standard SUPERCOP results interface

As in many engineering projects there are also restrictions that are of a more practical nature but nonetheless must be taken into account if any result is to be generated. In this case the main restrictions were limited manpower and no funding at all. The goals derived from those limitations are:

Goal 7 development using pre-owned or free development tools

Goal 8 re-use of as many existing components as possible

Goal 9 focus on SUPERCOP subset of current public interest: eBASH [3]

3 Hardware

3.1 Overview

XBX hardware consists of two main components: the eXternal Benchmarking Harness XBH and the eXternal Benchmarking Device XBD. The three hardware components PC, XBH and XBD are shown in Fig.1, together with their physical connections. The XBH connects to the PC running the eXternal Benchmarking Software (XBS), which is based on SUPERCOP, by means of Ethernet. This provides easy interfacing with any kind of computer that can run the SUPERCOP framework regardless of operating system. An RS232 port is available for low level configuration and debug output during development. Communication between the XBD and the harness is handled by means of a data connection and discrete digital I/O lines. The data connection is implemented using either I^2C or UART, depending on the type of device used as XBD. However, if UART is used for XBH-XBD communication the RS232 port of the XBH is no longer available for debugging purposes. The digital I/O lines are used for special purposes where the data connection would not perform adequately. The first purpose is device reset of the XBD. The XBD's reset pin is connected to a digital output of the XBH (in open collector configuration) and a pull-up resistor to the XBD's supply voltage. This allows the XBH to issue a hardware reset on the XBD, either because of a timeout or due to a command received from the PC. In the event that the XBD crashes, e.g. due to stack overflow, this mechanism provides a fast and reliable way to recover communication to the XBD in a situation where the data connection would be utterly useless.
Timing measurement is the second purpose that uses a dedicated digital I/O line. A digital output on the XBD is hooked up to an event capture pin on the XBH. Edges on that pin are triggered when a piece of code to be benchmarked is called and again when it returns. The event capture pin the XBH captures and timestamps these events and provides a duration from which a clock cycle count can be calculated.

3.2 Microcontroller Family

Due to goals 4, 7 and 8 the Atmel AVR 8-bit microcontroller family was selected to be the project's workhorse.

[3] SUPERCOP can benchmark many types of cryptographic algorithms, eBASH is the type for hash functions which are currently in focus due to the NIST SHA-3 competition.

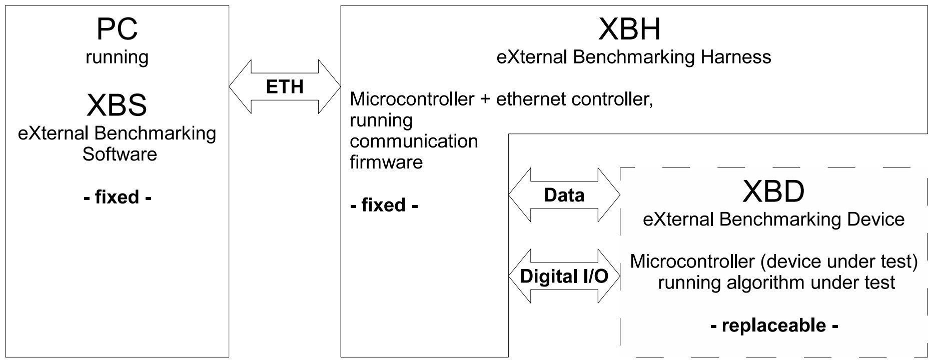
PC
running
XBS
eXternal Benchmarking Software
- fixed -
ETH
XBH
eXternal Benchmarking Harness
Microcontroller + ethernet controller, running communication firmware
- fixed -
Data
Digital I/O
XBD
eXternal Benchmarking Device
Microcontroller (device under test) running algorithm under test
- replaceable -

Fig. 1. High level overview of the XBX hardware setup. The PC on the left can be any computer that runs XBS, the XBH is a fixed interface component that does not need any adjustments while the XBD is the device under test and can be replaced at will.

This family has many beneficial features, starting with a low unit price and good availability in a variety of stores that sell to end customers. The performance delivered by the AVR family is high for an 8-bit design and many parts come in dual-in-line packaging, which is one of the few IC packages easily soldered by hand onto a perfboard[4] or similar carrier. On the software side, a port of the GNU Compiler Collection (GCC) exists for years now and is well tested. The GCC is supplemented by a standard C library implementation specifically tailored to the AVR and available for free. These features make the AVR family very popular with hobbyists around the world, which results in the added benefit of a huge user community providing ideas and code snippets for most questions that arise during development of an AVR based application. Also, the fact that one of the authors had a compatible JTAG adapter and some AVR chips in the closet accelerated the decision.

3.3 eXternal Benchmarking Harness XBH

The XBH setup comprises an Atmel ATmega644 microcontroller running at 16 MHz, a Microchip ENC28J60 Ethernet controller and a MAX232 TTL/RS232 voltage level shifter. For simplicity it is based on a commercially available module for home automation[7] which requires only minor modifications to act as XBH. An in system programmer connector is available to update the firmware as well as several signal connectors. The module runs on 9V AC power and can supply XBDs with up to 100mA of operating current.

[4] A perfboard is a prototyping board that component are soldered to, as opposed to a (solderless) breadboard which works with little wires that are just plugged in to form connections.

3.4 eXternal Benchmarking Device XBD

The XBD is the device on which the actual benchmarking takes place. The XBD module consists of a microcontroller and whatever clock source and voltage regulation is necessary for the controller. Due to the design of the XBX setup the XBD can be easily replaced, requiring only the data connection and a digital output pin for timing measurements. Connecting the reset pin is highly recommended, although not strictly required. The timing measurement does not use any timer resources on the XBD, so theoretically even a microcontroller without any timers could be used. However, in order to be able to calibrate the timing measurements before a benchmark run it is advisable to have at least one timer unit available on the XBD.
The reference implementation uses an ATmega644 with an 8 MHz crystal oscillator circuit that runs on the 5V supply of the XBH. The data connection to the XBH is I^2C in this implementation. Running the XBD at half the frequency of the XBH gives the reference implementation the best possible timing accuracy.
A different XBD implementation uses a Luminary Micro LM3S811 evaluation board. The LM3S811 is an ARM Cortex-M3 based 32-bit microcontroller and much more powerful that the AVR. It runs on 3.3V and the data connection uses a UART.

4 Software

The software is laid out as a chain of components with different tasks. Most components take the form of either shell or Perl scripts. Keeping several small components makes testing individual functions easier and reduces the likelihood of bugs compared to one big monolithic tool. The components are:

– Object file creation
– Download of binary code and parameters
– Execution framework
– Timing and result collection
– Benchmark control
– Post-processing

4.1 XBS: Benchmark Control

In a top-down view of the software architecture, benchmark control is highest layer. It talks to the user, it controls all actions. The benchmark control functionality is derived from SUPERCOP's control scripts, and called eXternal Benchmarking Software (XBS). SUPERCOP has a very simple user interface script called 'do'. Since one computer running XBS can control many different external targets the XBS scripts have some options that SUPERCOP's 'do' does not require.

XBS needs to be told which target platform to use since every platform potentially has it's own compilers, linkers and compiler options. The platform is selected by a line in a config file.

Once a build process for the selected platform is started the XBS copies the application code that is to be benchmarked into a temporary directory, where it is combined with supporting code called 'application framework' (AF). The AF provides all the communication services that the XBD will provide, except for bootloader functionality. The application code and AF combined for the binary of the application that is later downloaded into the XBD.

The binary just created is subjected to static analysis concerning the size of it's sections. If it is determined that this binary will not even fit into the memory of the XBD, download will not be tried.

If the binary passed the static analysis and looks as if it will fit into the XBD benchmark control calls a helper script that will talk to the XBH, which will talk to the bootloader on the XBD in order to download the newly created and checked application binary into the XBD. After successful download, the helper script is called again to trigger a short benchmark run executed and the performance of the triple [Algorithm, Compiler, Options] is stored for later reference, provided the binary produces the correct result for the known answer test. At this time, stack consumption is measured if the XBD AF for the selected platform supports this feature[5]. This process is repeated until all triples have been built, statically checked and if applicable, downloaded and benchmarked.

From the stored performance numbers for the short benchmark run the fastest tuple [Compiler, Options] per algorithm is selected and the corresponding binary application is downloaded into the XBD and subjected to a detailed benchmark at different message lengths.

The results of the short and detailed benchmark runs are written to a text based output file, with XBS specific information like stack use embedded as comments in SUPERCOPs results format. Thus the result files should be readable by the same tools that process SUPERCOP results [6] although without making use of the additional information like stack usage.

4.2 Algorithms to Benchmark

The major part of the algorithms benchmarked using XBX are taken from the SUPERCOP suite. The XBS scripts are closely modeled on SUPERCOP in that regard, using the same directory structure to hold algorithms and their implementations. Some code that was not submitted to SUPERCOP was adapted by the authors to fit the same interface and subsequently benchmarked. Most of this code came from 'Das Labor'[5], a small device working group related to Ruhr Uni Bochum who wrote a collection of cryptographic primitives implemented specifically for the Atmel AVR family.

[5] This is currently available for Atmel AVR targets only.

[6] A quick test by Dan Bernstein showed that they are indeed, although no written documentation of that test exists.

4.3 Hardware Abstraction

To separate the benchmarking logic from platform specific code such as communication, execution of binaries and debugging output a simple hardware abstraction layer (HAL) was employed. This HAL hides the device dependent aspects of the XBD like special function registers required to use the UART, the exact method used to erase and program flash pages in the bootloader and the fact that some microcontroller families, e.g. the Atmel AVR, are based on Harvard architecture and do not hide this fact from the programmer. The special treatment of constant *data* in what *per definitionem* is *program memory* on such devices is also hidden in the HAL. Without this functionality all constant data would end up in the RAM, rendering most SUPERCOP submitted implementations useless due to the fact that not even the initial stack would fit into RAM anymore.
One function in the HAL is especially important for goal 2: precise performance numbers. Since the actual timing measurement takes place on the XBH but the reported value is the amount of CPU cycles on the XBD it is important that the relation between the time bases on XBH and XBD is known as exactly as possible. To this end, a timing calibration service has been implemented. When the XBS requests a timing calibration, the XBH triggers the timing calibration routine on the XBD. This routine busy loops for a device dependent amount of time, toggles the timing output digital I/O line as it does when benchmarking algorithms and additionally counts the number of CPU cycles it spent between the two digital I/O toggles using an internal timer. This number of cycles is reported to the XBH, which reports it along with it's own timing measurement of the same event to the XBS. A correction factor for clock drift between XBH and XBD and/or a sanity check on the reported values can then be performed on the PC by the XBS. This timing calibration sequence looks as depicted in Fig.2 . Measuring stack usage is another challenge that can only be solved in a device dependent manner, yet should be available to the application via a standard interface. The HAL contains two functions that allow for stack measurement: paintStack and countStack. PaintStack 'paints' the free stack area with a known pattern, called a canary bird. Then the function to be benchmarked is called and after it returns, countStack counts the number of canary birds that did not 'survive'. This gives the maximum amount of stack used by the benchmarked function. Combined with the static RAM requirement obtained from the application binary and the known RAM requirement of the AF, the total RAM consumption of a triple [Algorithm, Compiler, Options] can be measured.

4.4 Application Framework

The application framework provides hardware independent basic management functions like processing requests, parameter handling and so on. It is combined with the algorithm to benchmark and the hardware abstraction layer for the device under test to form the XBX application binary.

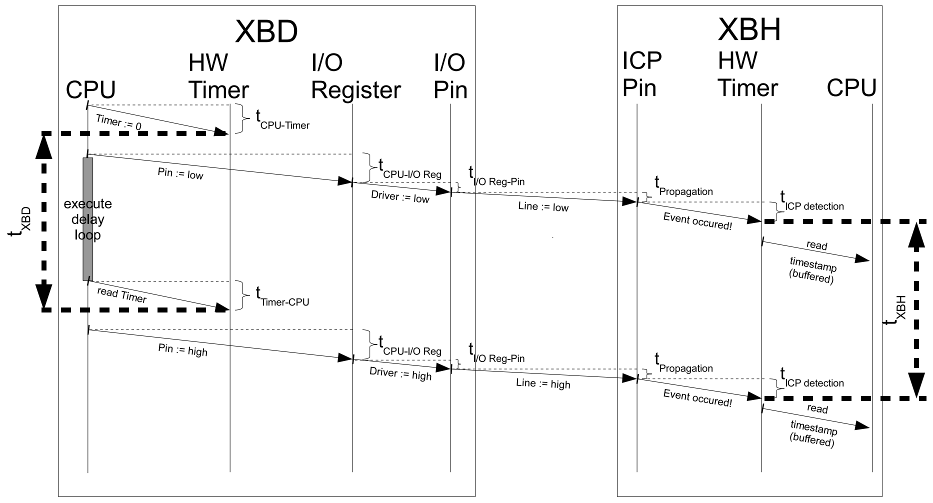

Fig. 2. UML sequence diagram for the XBX timing calibration routine. If it holds that a) reading and writing the timer requires the same time and b) switching the timing indicator pin high takes as long as switching it low then all delays from measurement and indication are symmetrical and cancel each other out. Thus if XBD and XBH have perfectly identical clock sources $t_{XBD} = t_{XBH}$ holds, otherwise $\frac{t_{XBD}}{t_{XBH}}$ can be used for sanity checks and is also logged by the XBS so it can later manually be applied as a correction ratio by the user.

4.5 Bootloader

The bootloader is used to download and execute application binaries. It is formed by combining the generic boot loader logic code with the hardware abstraction layer. Before benchmarking a target device, the bootloader must once be manually uploaded, e.g. via JTAG. After that, application download and execution is handled over a communication channel established between the bootloader and the benchmarking harness.

4.6 Benchmarking Harness

The firmware for the XBH receives commands from the XBS on the PC by means of UDP packets. The commands form a protocol that is simply called the XBH protocol, which features ASCII based command words with ASCII-encoded hex digits as parameters. Although the upper 4 bit of every byte are wasted, this is no major concern. The bandwidth on the Ethernet is orders of magnitude higher than on the following I^2C or UART link to the XBD. The ASCII encoding allowed for simple testing using the *netcat* command in the early development phases and is easily processed by Perl scripts. Monitoring XBH protocol in a serial terminal also benefits from this choice. The XBH software generates requests to the XBD as necessary. The protocol for these requests is called XBD protocol and uses ASCII commands but binary encoding for the parameters. Answers from the XBD are processed and reported back to the XBS over Ethernet using XBH protocol. Since the XBS never uses the XBD protocol, changes in the XBS do not affect existing XBDs and vice versa. Keeping in line with goal 8: re-use, the XBH firmware consists of an embedded web server software[6] by Ulrich Radig with the XBH functionality added as another UDP service and some modifications to the timer handling code in order to facilitate XBD timing measurements.

5 Benchmarking Results

This sections gives a few examples of benchmarking results obtained using the XBX.

5.1 Different Implementations of Skein512 on Atmel ATmega1281

The ATmega1281 comes from the same family as the ATmega644 in the reference implementation of XBX, it has the same CPU core and performance per MHz. The RAM however is at 8kiB twice as big which helps enormously in running implementations that are not size optimized. The performance numbers for Skein512[8] listed in table 1 are certainly not the best the algorithm can do, they just reflect the best implementations available to the authors at the time this work was written. The C implementations were compiled with the AVR port of GCC, the original SUPERCOP submission by the Skein team had to be modified to fit into the XBD's flash memory. Even though the ATmega1281 comes with 128kiB of flash memory, this implementation was so aggressively loop unrolled that the second author of this work has to re-roll the loops manually to make it fit. This is a typical issue with speed optimized SUPERCOP submissions intended to run on a PC and by no means specific to the Skein team's work. The huge advantage of the assembly implementation in both execution time and

Table 1. Skein512, different implementations on Atmel ATmega1281 in cycles per byte

Property	Skein Team C	Das Labor C	Das Labor ASM
cpb @ 1536 bytes message length	7842.6	8602.1	1571.1
RAM usage(Stack+global/static)	2684	1580	1391

space is evident, yet in this case a size optimized C implementation also gives a 40% advantage in RAM usage at only a small performance loss.

5.2 SHA-3 candidates on an ARM Cortex-M3 32-bit CPU using two compilers

One of the most interesting features of SUPERCOP for many users is the ability to benchmark the same implementations using different compilers and compiler options. XBX preserves this property for the target platforms where several compilers are available. One such platform is the ARM Cortex family. Using a free trial license of the ARM C compiler (ARMCC) we obtained performance numbers of SHA-3 candidate implementations using both GCC and ARMCC on a Cortex-M3 based Luminary Micro LM3S811 microcontroller. ARMCC is generally considered to be expensive but also the most sophisticated compiler available for ARM CPUs. Expectation was that it would perform better than GCC. The fastest SHA-3 candidates on this platform from the subset available for XBX benchmarking at the time met this expectation, as can be seen in table 2.

Table 2. ARMCC vs. GCC on ARM Cortex-M3 in cycles per byte for a 1536 byte message, fastest algorithms on platform

Compiler	BMW256[9]	Shabal512[10]	BMW512[9]
ARMCC	24.2	33.6	48.2
GCC	26.2	43.5	70.3

Surprisingly, some algorithms suite the optimization strategies of GCC better, resulting in an overall better performance. See table 3 for details.

Table 3. ARMCC vs. GCC on ARM Cortex-M3 in cycles per byte for a 1536 byte message, unexpected behavior

Compiler	Blake32[11]	Keccak1024c576[12]	CubeHash1632[13]
ARMCC	96.2	125.1	835.8
GCC	72.1	109.7	323.2

In general, the Cortex-M3 performs very well when compared to a roughly 50 [14] [15] [16] times larger Intel Pentium 3 desktop processor as benchmarked in [17]. Table 4 shows a performance gap of barely factor 2 between the two CPUs with respect to the three SHA-3 candidates.

Table 4. Cortex-M3 vs. Pentium 3 (683) in cycles per byte for a 1536 byte message

Compiler	BMW256	Shabal512	BMW512
Cortex-M3	24.2	33.6	48.2
Pentium 3	13.82	14.24	30.04

6 Conclusion

In this paper we introduced an extension to the SUPERCOP benchmarking suite and described the main design decisions and compromises we made in order to get it running in time for the second round of the SHA-3 competition. We believe that the overall design is sound and that using SUPERCOP-XBX meaningful results both for speed and memory requirements of cryptographic hash functions can be obtained.

Acknowledgments. First of all many thanks to Daniel J. Bernstein and Tanja Lange for their support and encouragement. Their approval really means a lot to us. The Cortex-M3 target was provided by ARM, Ltd. and we have Richard York and Alex Nancekievill to thank for that. Since the XBX project has no funding their support is most appreciated.

References

1. Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, accessed 5 November 2009.
2. NIST: First SHA-3 Candidate Conference `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/Feb2009/program.html`, accessed 27 February 2010.
3. Daniel J. Bernstein: Cache-timing attacks on AES `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, accessed 27 February 2010.
4. Kota Ideguchi, Toru Owada, Hirotaka Yoshida: A Study on RAM Requirements of Various SHA-3 Candidates on Low-cost 8-bit CPUs `http://www.sdl.hitachi.co.jp/crypto/lesamnta/A_Study_on_RAM_Requirements.pdf`, accessed 27 February 2010.
5. Daniel Otte et al.: AVR Crypto Lib `http://www.das-labor.org/wiki/AVR-Crypto-Lib/en`, accessed 27 February 2010.
6. Ulrich Radig: AVR Webserver Software `http://www.ulrichradig.de/`, accessed 27 February 2010.
7. Pollin: AVR-Net-IO Board `http://www.pollin.de/shop/downloads/D810058B.PDF`, accessed 28 February 2010.
8. Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, Jesse Walker - The Skein Hash Function Family Submission to NIST (Round 2), 2009
9. Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jørn Amundsen, Stig Frode Mjølsnes - Cryptographic Hash Function BLUE MIDNIGHT WISH Submission to NIST (Round 2), 2009

10. Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-François Misarsky, Marìa Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, Marion Videau - Shabal, a Submission to NISTs Cryptographic Hash Algorithm Competition Submission to NIST, 2008
11. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan - SHA-3 proposal BLAKE Submission to NIST, 2008
12. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche - Keccak specifications Submission to NIST (Round 2), 2009
13. Daniel J. Bernstein - CubeHash specification (2.B.1) Submission to NIST (Round 2), 2009
14. ARM: Whitepaper about the Cortex-M3 `http://www.arm.com/files/pdf/IntroToCortex-M3.pdf`, accessed 28 February 2010.
15. Intel: Presskit on Moore's law `http://www.intel.com/pressroom/kits/events/moores_law_40th/`, accessed 28 February 2010.
16. Intel: Pentium 3 datasheet `http://developer.intel.com/design/pentiumiii/datashts/245264.htm`, accessed 28 February 2010.
17. Daniel J. Bernstein and Tanja Lange (editors): SUPERCOP benchmarking results. See results for computer 'manneke'. `http://bench.cr.yp.to/results-hash.html`, accessed 27 February 2010.