An Evaluation of Hash Functions on a Power Analysis Resistant Processor Architecture

Simon Hoerder, Marcin Wójcik, Stefan Tillich, Daniel Page

▶ To cite this version:

Simon Hoerder, Marcin Wójcik, Stefan Tillich, Daniel Page. An Evaluation of Hash Functions on a Power Analysis Resistant Processor Architecture. 5th Workshop on Information Security Theory and Practices (WISTP), Jun 2011, Heraklion, Crete, Greece. pp.160-174, 10.1007/978-3-642-21040-2_11 . hal-01573303

HAL Id: hal-01573303

https://inria.hal.science/hal-01573303

Submitted on 9 Aug 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

An Evaluation of Hash Functions on a Power Analysis Resistant Processor Architecture

Simon Hoerder[1], Marcin Wójcik[1], Stefan Tillich[1], Daniel Page[1]
{hoerder, wojcik, tillich, page}@compsci.bristol.ac.uk

Department of Computer Science, University of Bristol

Abstract. Cryptographic hash functions are an omnipresent component in security-critical software and devices; they support digital signature and data authenticity schemes, mechanisms for key derivation, pseudo-random number generation and so on. A criterion for candidate hash functions in the SHA-3 contest is resistance against side-channel analysis which is a major concern especially for mobile devices. This paper explores the implementation of said candidates on a variant of the Power-Trust platform; our results highlight a flexible solution to power analysis attacks, implying only a modest performance overhead.

1 Introduction

Within the cryptographic community, open "contests" to evaluate, select and standardise the use of secure and efficient primitives have become de rigueur. The most high-profile example is the Advanced Encryption Standard (AES) contest run by NIST from 1997 to 2000 to find a replacement for DES, the incumbent block cipher design. This model was repeated in 2007 when, partly motivated by increasingly able attacks [1, 2] on SHA-1 [3], NIST launched the SHA-3 contest [4] to develop a new cryptographic hash function. Briefly, a hash function

$$H : \{0,1\}^* \longrightarrow \{0,1\}^n$$

maps an arbitrary-length input (or message) to a fixed-length, n-bit output (or digest). Hash functions support, for example, digital signature and data authenticity schemes, mechanisms for key derivation and pseudo-random number generation and are indispensable for security-critical devices. As such, various security requirements (e.g., the need for H to be collision resistant) are outlined in [5]. However, in common with the AES contest, other metrics are important for SHA-3; specifically, efficiency in hardware and on a variety of software-based platforms is paramount.

Within the context of embedded and mobile computing, such metrics are particularly pertinent: they represent the exact resources in short supply. The same context may imply additional requirements in the sense that physical security (e.g., against side-channel and fault attacks) is also a valid metric. Example attacks on hash functions are given by [6–11]; these are exacerbated by the wide range of use cases. Ideally one has an idea of the trade-offs different countermeasures offer so as to select the right one before deployment, but in practice this topic has not drawn much attention (for example, note the discussion triggered by Rivest's question [12] on the matter).

Keeping this difficulty in mind, one attractive approach is to provide a "generic" countermeasure. For power analysis based attacks, and focusing on hardware implementation, this can be realised by utilising a so-called secure logic style. The idea is to take a generic circuit and automatically replace CMOS cells with alternatives such as SABL [13] or WDDL [14]. To consider a similar approach for software, one must instrument a generic countermeasure so that each instruction is prevented from leaking information during execution. Several proposals exist, such as NONDET [15], but a more concrete and complete implementation is provided by Power-Trust [16, 17]. The SPARC V8-based Power-Trust platform houses a secure zone, implemented in a secure logic style, and security-critical instructions are executed only by this zone to avoid leakage; the result is a generic countermeasure, mounted in a general-purpose processor, which offers an extremely flexible solution. The question is, how does this solution fare wrt. the SHA-3 use-case? Does it, for example, imply a performance overhead low enough to allow secure deployment of the selected SHA-3 candidate in embedded and mobile applications?

Focusing on 6 of the 14 remaining (as per round two) SHA-3 candidates, this paper addresses three points all stemming from the same underlying work, namely investigation of said candidates on the Power-Trust platform:

Performance of candidates, i.e., assuming that Power-Trust provides an adequate countermeasure against power analysis attacks, what overhead does this imply and is this the best approach? Section 2.1 presents a concrete attack scenario; the criteria for performance includes throughput and instruction mix (e.g., any bias toward memory access).

Agility of SPARC V8 analogous instructions, i.e., given a set of protected instructions required for one SHA-3 candidate, can we implement another candidate with the same set?

Potential for advanced Instruction Set Extensions (ISEs), i.e., for which candidates can we find useful ISEs? For example, we suggest several generic (i.e., not Power-Trust-specific) instruction set extensions which could be used to accelerate BMW.

One can view the second and third points as evaluating the Power-Trust design itself; the novel aspect in this respect is the workload used (namely the SHA-3 candidates), which is more diverse than previously studied.

2 Background

2.1 Side-Channel Attacks on Hash Functions

There are numerous examples of successful side-channel attacks on specific hash functions, and a variety of specific countermeasures have been proposed [6–11].

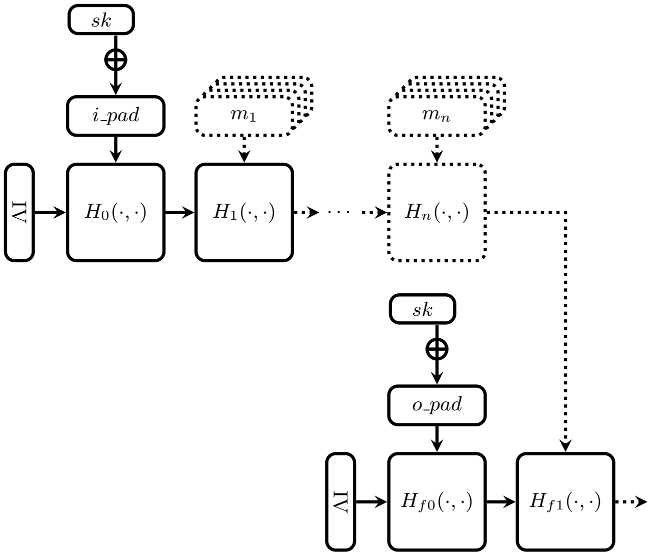

Fig. 1. Protecting HMAC from side-channel attack: only the operations and values drawn with solid lines need to be protected.

However, within the context of developing hash functions it is attractive to be more general (both for attacks and countermeasures) since this allows far easier high-level comparison. As such, we keep our model of side-channel attack as generic as possible within practical limits:

Simple Power Analysis (SPA) is possible whenever the sequence of operations performed during execution of H depends on a fixed, security-critical input.

Differential Power Analysis (DPA) is possible when the input to an invocation of H combines fixed security-critical data and variable data which can be controlled by the attacker. That is, invocation resembles $H(s, m)$ for a fixed, security-critical s and variable m.

Timing Attacks are possible whenever a security-critical input affects the time taken to execute H; examples include conditional branches or cached table look-ups based on said input.

Additionally, for all three types of attacks we allow the attacker to perform a profiling step to create templates. In the following, $H_i(s_{i-1}, m_i)$ denotes the i-th invocation of the compression function used by H with the state (or chaining variable) s and message m as input.

Neither SPA nor timing attacks matter for the hash functions that we are considering: they always use the same instruction sequence (i.e., there are no input-dependant branches), and do not use any table look-ups. Indeed, CubeHash [18, Page 3] makes this an explicit design criterion. On the other hand, DPA *does* matter. Fig. 1 illustrates this fact for H used within the popular HMAC construction. The outputs of H_0 and H_{f0} are intermediate states; they are both fixed (they depend only on constant values) and security-critical (since they are derived directly from sk, the key used to authenticate messages). Thus, H_1 and H_{f1} fall squarely into our attack scenario:

$$\begin{aligned} &H_1\big(\ \underbrace{H_0(IV, sk \oplus i\_pad)}_{\text{constant, secret}}\ ,\ \underbrace{m_1}_{\text{variable}}\ \big) \\ &H_{f1}\big(\underbrace{H_{f0}(IV, sk \oplus o\_pad)}_{\text{constant, secret}}, \underbrace{H((sk \oplus i\_pad) \parallel m)}_{\text{variable}}\big) \end{aligned}$$

To roughly outline a potential DPA attack, notice that the attacker can repeatedly invoke the HMAC construction with an m of his choice. By observing the power consumption during execution, correlation between the data-dependent interaction of m and the secret constant allows him to hypothesise about the value of the constant and ultimately to recover it, thus undermining security.

This scenario demonstrates the value of an agile solution via two points. First, an inflexible solution dictates the H to be used; this is unattractive because if H is (seriously) broken, one might hope to change it without incurring significant cost. Second, notice that the vulnerable invocations are H_1 and H_{f1} only, while $H_{2,\ldots,n}$ need no protection as long as the compression function is one-way. An inflexible hardware-oriented approach might implement a countermeasure for *all* invocations, hence incurring a performance overhead in each. A more flexible solution would apply the countermeasure only where necessary, and potentially provide a performance advantage in other cases.

2.2 The Power-Trust Platform

The so-called "Power-Trust platform" is a SPARC V8-based ASIC prototype of a side-channel resistant embedded processor implementing the security concept developed in the context of the Power-Trust project [16, 17]; its main goal is to evaluate the validity and effectiveness of the security concept as a whole. Furthermore, the prototype allows to investigate various design options and trade-offs in practice, e.g., different management instructions, exception handling features, and secure logic styles. The Power-Trust prototype has integrated support for AES and ECC, but the security concept itself is principally suited for handling a wide range of cryptographic workloads.

The basic idea of the security concept used in Power-Trust is to combat side-channel leakage directly in the processor hardware. The main concern are power and EM analysis attacks, but also timing attacks are mitigated. As a first measure, the circulation of potentially vulnerable datums is restricted to a tiny portion of the processor (essentially the functional units) by masking them whenever they are not required. This includes values which pass through various pipeline stages or which are written to caches and memories. The second measure protects all remaining vulnerable parts of the processor containing the unmasked data values themselves, the masks, and any values related to mask generation, by implementing them in a secure logic style. This part of the processor is referred to as “secure zone” and shown in Fig. 2. The secure zone offers a range of instructions which can be executed within its boundaries. From its interface, the secure zone looks very similar to a regular functional unit, which facilitates integration into the processor.

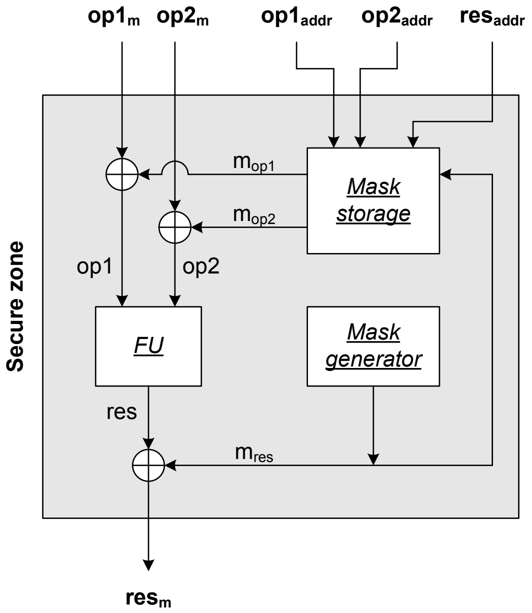

Fig. 2. The secure zone of the Power-Trust platform.

In the following, we explain the secure zone concept from a programmer’s point of view. In order to implement a cryptographic algorithm in a power-analysis resistant manner, the following steps are necessary:

- Before execution, the inputs to the cryptographic algorithm are masked explicitly by the caller.
- Any instructions which produce potentially vulnerable values must be executed within the secure zone.
- Depending on the implementation and the secure zone capabilities, it might be required to save some masks to memory and restore them later on to the secure zone.
- Once the output of the cryptographic algorithm has been calculated, the mask is removed by the caller.

Explicit masking and unmasking of inputs and outputs can be seen as transferring values between “normal” domain and “masked” domain. In the masked domain, values can only be manipulated by instructions of the secure zone (in the following denoted as *secure zone instructions*)[1], and consequently a cryptographic algorithm can only be protected if it can be implemented with secure zone instructions. The number of masked values which are readily available for processing is limited by the number of masks that the secure zone can actually store. However, masks can be swapped in and out of the secure zone in order to extend the number of masked values at the expense of some additional *mask management instructions* and storage.

In relation to the workload of typical cryptographic algorithms, masking and unmasking constitute only a minor overhead. Implementations using secure zone instructions can even see a considerable speed-up in comparison to the use of native processor instructions, since secure zone instructions can be tailor-made to fit specific algorithms or classes of algorithms. However, the management of masks might entail overheads, especially if a large number of masked values is required. Thanks to the flexibility of the mask management instruction set, this overhead can often be minimised by exploiting the structure of the protected cryptographic algorithm.

A mask may never directly leave the protection of the secure zone, as otherwise an attacker might launch a higher-order attack [19] on masked data and the corresponding mask. However, masks need to be extracted if the secure zone runs out of storage entries for masks or if there is a task switch. For this case, masks can be represented as a specific state of the mask generator unit which originally produced the mask and the number of steps the mask generator has taken till it produced the mask. The secure zone features mask management instructions for extracting the state of the mask generator and the step count. Similarly, instructions for setting the mask generator state and regenerating masks from a given step count exist for restoring masks to the secure zone.

In the Power-Trust prototype, the mask generator state consists of 128 bits and the step count (including some additional meta-information) is another 32 bits; therefore in the worst case a 32-bit mask requires five 32-bit words of storage. However, several stored masks can relate to the same state of the mask generator, thus greatly reducing the required memory. Similarly, the software can take steps to ensure the step count from a given mask generator state is low when a mask is written out to memory. In this way, when the mask is then restored, the required number of instructions is limited.

2.3 Our Variant of the Power-Trust Platform

We made two additional choices at the architectural level in order to ensure realistic and comparable results:

– We needed to decide on the number of masks that can be held within the mask storage, selecting 32 as a trade-off between 8 masks supported by the

[1] Of course, masked values could be manipulated by "normal" processor instructions, but this would mean that masked values and masks become desynchronised, leading to erroneous output from the algorithm.

current IC prototype and the upper bound of 2^{10} which could principally be supported by the architecture. We believe this to be at the upper-edge of economic possibilities, but will demonstrate that some candidates can be implemented with a much smaller mask store.
- We only add instructions to the secure zone if they can be executed in one cycle and do not affect the critical path. Designing more elaborate functional units (e.g., multipliers) for the secure zone is principally possible but would require considerable design effort and is left for further research.

Based on these choices, we are confident that any subsequent prototypes can support our candidates without significant differences in performance from the original.

A third choice had to be made on the software level; any consideration of operating system influences such as trap handling, interrupts and context switches would have biased our results towards a specific use case, e.g., toward smartcards. However, we want our results to be as generic as possible, and therefore did not consider any operating system.

3 Implementation of Hash Functions on the Power-Trust Platform

To make comparison easier, we chose the following hash functions for implementation on the Power-Trust platform:

- BLAKE-32 and its third round version BLAKE-32v3 ([20], [21])
- BlueMidnightWish−256 ([22], [23])
- CubeHash160+16/32+160-256 and CubeHash16+16/32+32-256 which was suggested for the third round ([18], [24])
- Keccak$[1088, 512, 32]$ ([25], [26])
- SHA-256 ([27])
- Shabal-256 ([28], [29])
- Skein-256-256 (this is the “low-memory” proposal of [30])

In the following we will motivate these choices, and highlight noteworthy specifics in our implementations. Where possible we follow the notation of the original submissions.

BLAKE-32 *and* SHA-256 were easy to implement for us due to their small internal states; for BLAKE-32 we followed the example of the “Optimized_32bit” implementation provided by the BLAKE team. We also give numbers for the third round version BLAKE-32v3 which increases the number of rounds from 10 to 14.

CubeHash operates on a state that is too large to fit completely into the registers, and uses a large number of round function iterations within each H_i; this is a bad combination for Power-Trust because masked data has to be swapped in and out of memory frequently. However, by choosing a suitable memory layout the number of memory accesses can be reduced. The 1024-bit CubeHash state is represented as a 5-dimensional cube `state[][][][][]` of two 32-bit words per dimension but can be split into four 3-dimensional subcubes `state[x₁][][][][x₅]` with $x_{1,5} \in \{0, 1\}$ requiring 8 masked registers each. The first 9 steps of the round function can be computed first on the two subcubes `state[x₁][][][][0]`, and then on the two subcubes `state[x₁][][][][1]` since there are no interdependencies during these 9 steps. Therefore, no more than two of these subcubes have to be kept in registers at any point of time. The 10-th (and last) step swaps the subcubes `state[1][][][][x₅]` which can be implemented simply by swapping pointers. Overall, this means that only three subcubes have to be loaded and stored per iteration of the round function.

Keccak$[1088, 512, 32]$ allows a separation in the memory layout similar to that afforded by CubeHash if one follows the example given by the unrolled "Optimized_32bit" implementation by the Keccak team. However, one has to deal with ten memory blocks of five 32-bit values, using 10 masked registers for each of the intermediate variables `D[5][2]` and `C[5][2]`.

BMW256 had to be implemented without optimisation regarding memory usage since its internal state can not be separated into bigger blocks, as it was for CubeHash or Keccak, due to its high interdependency. However, we were able to identify two possible sets of generic ISEs that implement the six $s_{0,\ldots,5}$-functions

$$\begin{aligned} s_{i\in\{0,\ldots,3\}}(x) &:= \mathtt{SHR}(x, c_{i,0}) \oplus \mathtt{SHL}(x, c_{i,1}) \oplus \mathtt{ROTL}(x, c_{i,2}) \oplus \mathtt{ROTL}(x, c_{i,3}) \\ s_{i\in\{4,5\}}(x) &:= \mathtt{SHR}(x, c_{i,0}) \oplus x \end{aligned}$$

where SHR, SHL, and ROTL denote right shift, left shift, and left rotate, respectively, and $c_{i,j}$ are constants specifying the number of bits to shift or rotate. Furthermore, they always occur in combination with a modular addition

$$z \leftarrow s_i(x) + y \bmod 2^{32}$$

where y in some cases is the output of another s-function. Thus we implemented and compared three versions of BMW256:

BMW **Plain:** This implementation of BMW256 uses no ISEs.
BMW **"generic s":** This implementation uses one ISE, namely

```
SZ_BMWS %x, %i, %z
```

to compute

$$z \leftarrow s_i(x).$$

BMW **"special s":** This implementation uses six ISEs, namely

$$\texttt{SZ\_BMWS}_i \texttt{ \%x, \%y, \%z}$$

to compute

$$z \leftarrow s_{i \in \{0,\ldots,5\}}(x) + y \bmod 2^{32}.$$

Shabal-256 , according to our results, has significantly worse performance on Power-Trust than other SHA-3 candidates. One of the main reason is the relatively large internal state comprising forty-eight 32-bit words which exceeds the number of processor registers. Additionally, it requires a relatively large amount of iterations with a high interdependency between the internal state variables. The resulting memory accesses for masked values generate considerable overhead on the Power-Trust platform.

Skein offers two replacements for SHA-256, a "primary proposal" Skein-512-256 and a "low-memory" proposal Skein-256-256 targeted at embedded devices; since our platform is intended for mobile and embedded devices we chose to implement Skein-256-256. The Skein family is optimised for 64-bit architectures, but most of the Skein-256-256 kernel can be easily implemented on our 32-bit architecture; the exception is addition of 64-bit values. To implement it, we use SZ_ADDcc and SZ_ADDX commands analogous to the SPARC V8 ADDX and ADDcc instructions which require a carry flag within the secure zone[2]. The flag has to be taken care of by the scheduling algorithm of the operating system and is not reflected in our analysis any further.

Other hash functions We did not consider the AES-based candidates (e.g., ECHO) since we expect they can be implemented using variants of the existing AES-oriented ISEs within Power-Trust. In addition, we did not consider candidates requiring multiplication (e.g., SIMD); as mentioned before, implementation of sufficiently efficient multipliers for the Power-Trust secure zone is a non-trivial task which we reserve for future work.

4 Results

4.1 Instruction Set Agility

Many SHA-3 candidates have been developed with ISEs in mind: CubeHash, for example, can capitalise on the availability of SSE-based ISEs on x86 platforms. In a similar way, certain candidates can exploit ISEs available in the Power-Trust platform. For example, it already provides AES-oriented ISEs and, in Section 3, we outlined various extensions for BMW256.

[2] One straight forward possibility to do this is to provide a command that reads the value of the flag (and all other flags if there are any) into a masked registers which can then be stored to memory and another command to restore the flags from a masked register.

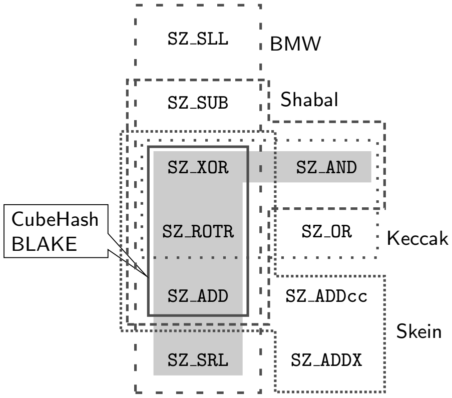

Fig. 3. Generic commands required from the secure zone to implement SHA-3 candidates. With the exception of SZ_ROTR they have an unprotected equivalent in the SPARC V8 instructions (see appendix B, [31]).

Despite the advantage this implies on platforms which support such ISEs, the approach is a potential disadvantage on platforms which do not: on many mobile and embedded platforms, for example, SIMD ISEs are missing (unless one counts packed arithmetic within word-sized values). As a result, it is useful to consider the agility of a minimal instruction set as a design metric for Power-Trust. That is, given there is an inherent cost associated with adding an ISE to Power-Trust, it is attractive to support a broad workload (i.e., many different SHA-3 candidates) using as few secure zone instructions as possible. This is especially important when considering the need to support migration from SHA-256 to SHA-3 within the device lifetime. With this in mind, we investigated which Power-Trust instructions are required by each candidates; the result is shown in Fig. 3. A major feature of the results is that a secure zone processor that provides instructions which can implement SHA-256, also provides all instructions needed for BLAKE-32 and CubeHash as well. For all the other candidates considered, additional instructions must be implemented in the secure zone and will incur additional expenses.

4.2 Performance

To optimise performance, we fully unrolled all implementations, hard-coded any constants and ran them on a cycle-accurate simulator of the platform. We did not implement message padding; instead we assumed having a padded but unmasked message block stored in memory which is loaded when appropriate and then masked. As shown in Section 2.1, the most common case for hash function implementations with countermeasures against side-channel attack will have a preformatted, unmasked message in memory which has to be hashed without leaking information on the previous states. The code can easily be adopted to hash masked messages and the costs of loading a masked message block is to some extent absorbed by the then superfluous message masking. However, this would have required a convention with the calling function how to store a masked message and the related mask information.

The performance results of our implementations are shown in Table 1; the instruction counts show the candidates separated into three distinct but constant groups. The first group is formed by BLAKE-32, BLAKE-32v3, SHA-256 and all three BMW256 implementations. The second group comprises CubeHash16+16/32+32-256, Keccak$[1088, 512, 32]$ and Skein-256-256; the ranking within this group varies depending on the performance criteria. Most notably, it shows the performance disadvantage CubeHash16+16/32+32-256 and Keccak$[1088, 512, 32]$ incur in software implementations for supporting only one state size for all security parameters. The third group comprises CubeHash160+16/32+160-256 (which has been superseded by CubeHash16+16/32+32-256) and Shabal-256. Both are not competitive on this platform. (See also Fig. 4 and Fig. 5.)

Hash Function	#Ops		#Ops/byte		#Registers		code size
	$H_i(\cdot,\cdot)$	H	$H_i(\cdot,\cdot)$	H	masked	in total	$H_i(\cdot,\cdot)$
BLAKE-32	4142	4142	64.72	64.72	18	23	17.28kB
BLAKE-32v3	5678	5678	88.72	88.72	18	23	23.53kB
BMW256	6042	15068	94.41	235.44	29	32	25.44kB
BMW256, "generic s" ISE	4686	12356	73.22	193.06			20.15kB
BMW256, "specialised s" ISEs	4622	12228	72.22	191.06			19.90kB
CubeHash160+16/32+160-256	14880	160540	465.00	5016.88	17	21	62.00kB
CubeHash16+16/32+32-256	14880	44444	465.00	1388.88			62.01kB
Keccak$[1088, 512, 32]$	107960	107960	812.35	812.35	30	32	456.68kB
SHA-256	6833	6833	106.77	106.77	22	27	27.12kB
Shabal-256	128387	513548	2006.05	8024.19	31	32	556.66kB
Skein-256-256 ("low-memory" variant)	13222	39666	413.19	1239.56	28	31	54.72kB
	see Fig. 4		see Fig. 5				

Table 1. Total number of instructions required for one iteration of the compression function $H_i(\cdot,\cdot)$, to hash a one-block message (denoted by H) and the register usage of the implementations.

Hash Function	#`load`		#`store`		#`load+store`	
	$H_i(\cdot,\cdot)$	H	$H_i(\cdot,\cdot)$	H	$H_i(\cdot,\cdot)$	H
BLAKE-32	192	192	26	26	**218**	**218**
BLAKE-32v3	256	256	26	26	**282**	**282**
BMW256	512	1634	302	757	**814**	**2391**
CubeHash160+16/32+160-256	1256	13814	1668	17544	**2924**	**31358**
CubeHash16+16/32+32-256	1256	3830	1668	4872	**2924**	**8702**
Keccak$[1088, 512, 32]$	14674	14674	9810	9810	**24484**	**24484**
SHA-256	328	328	48	48	**376**	**376**
Shabal-256	20268	81072	12111	48444	**32379**	**129516**
Skein-256-256 ("low-memory" variant)	544	1632	541	1623	**1085**	**3255**
					see Fig. 6	

Table 2. Total number of load and store instructions required for one iteration of the compression function $H_i(\cdot,\cdot)$ and to hash a one-block message (denoted by H).

Another interesting metric are the number of load and store instructions contained within the total number of instructions; these are listed in Table 2. The

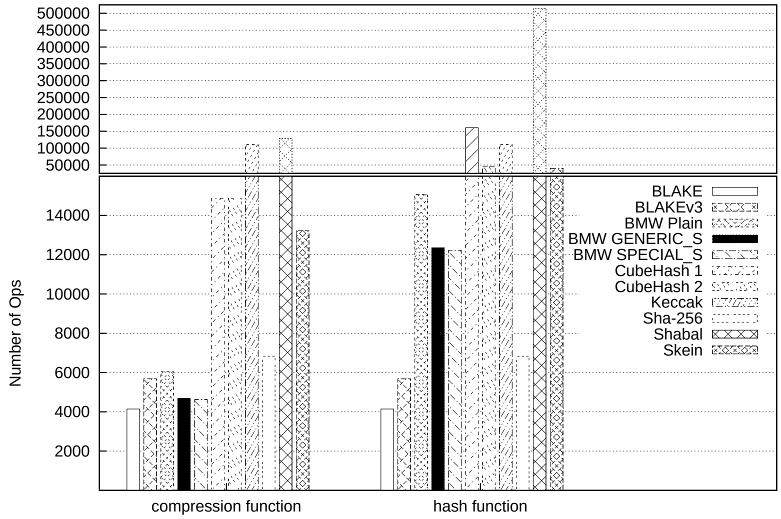

Fig. 4. Total number of instructions (including loads and stores) for one block (see Table 1).

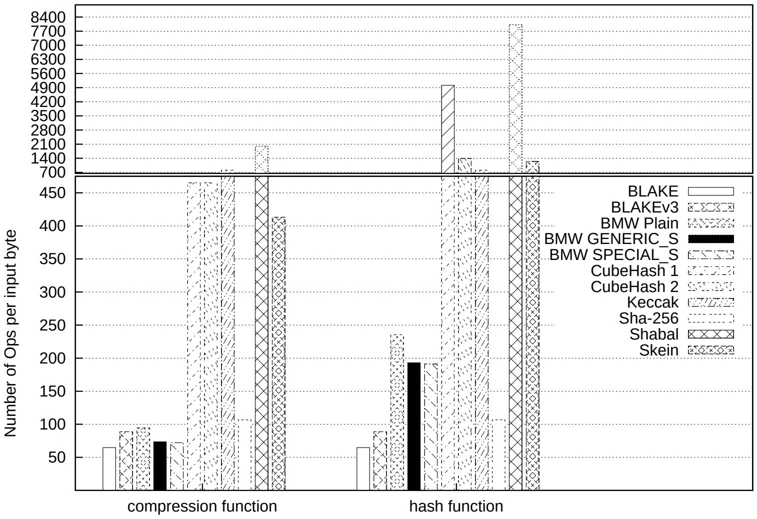

Fig. 5. Number of instructions per byte of input for one block (see Table 1).

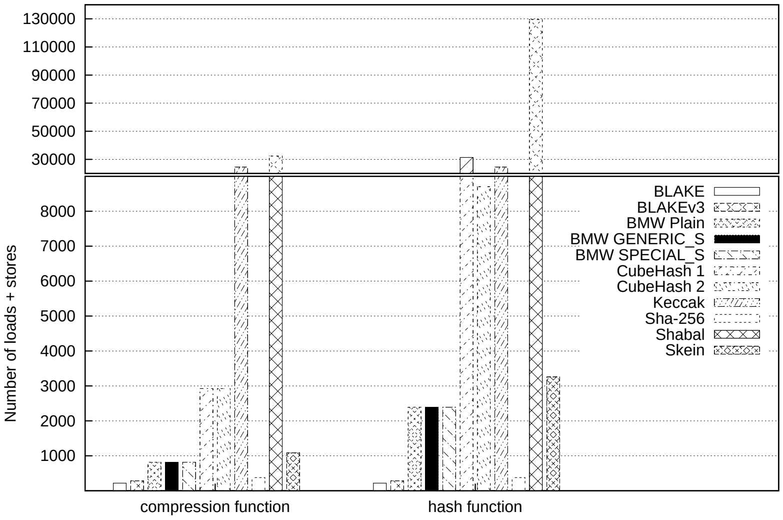

Fig. 6. Total number of load and store instructions for one block (see Table 2).

ISEs for BMW256 have, as expected, no influence on the number of memory accesses. The ranking of algorithms in respect to memory access is not very different from the ranking in respect to instructions per byte; while Keccak$[1088, 512, 32]$ ranks better on the number of instructions per byte than Skein-256-256 and CubeHash16+16/32+32-256, it is behind them in the number of memory access. The poor ranking of Shabal-256 with respect to any of the instruction counts is easily explained by the high number of memory access required to implement it; about 28% of all instructions are memory accesses.

Working in the design phase of a processor architecture, we can only present operation counts that do not represent the costs of `load` and `store` instructions properly. Therefore we decided not to compare our results with other studies such as eBASH [32] as they measure their results in processor cycles.

5 Conclusions

In this paper we demonstrated the flexibility of the Power-Trust platform wrt. to provision of generic countermeasures against side-channel attacks at reasonable costs; these metrics are paramount in the design and deployment of hash functions on secure embedded and mobile devices. Furthermore, our analysis contributes to the SHA-3 competition by highlighting, for the first time, the cost each candidate incurs from hardware protection. We additionally provided the first example of (non-AES) ISEs for BMW256, and outlined design requirements for general ISEs in this area. Furthermore, the effort to produce human-optimized code for these hash functions highlights the need to develop a compiler for this platform in the future; this work will then provide a good base to measure the compiler's efficiency.

Acknowledgements. The research described in this paper has been supported by EPSRC grant EP/H001689/1, and, in part by the European Commission through the ICT Programme under contract ICT-2007-216676 ECRYPT II. The information in this document reflects only the author's views, is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. Wang, X., Yu, H., Yin, Y.: Efficient Collision Search Attacks on SHA-0. In: Advances in Cryptology (CRYPTO), Springer-Verlag LNCS 3621 (2005) 1–16
2. Wang, X., Yin, Y., Yu, H.: Finding Collisions in the Full SHA-1. In: Advances in Cryptology (CRYPTO), Springer-Verlag LNCS 3621 (2005) 17–36
3. National Institute of Standards and Technology (NIST): Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-2 (Aug. 2002) `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.

4. National Institute of Standards and Technology (NIST): Cryptographic Hash Algorithm Competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html
5. Rogaway, P., Shrimpton, T.: Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance and Collision Resistance. In: Fast Software Encryption (FSE), Springer-Verlag LNCS 3017 (2004) 371–388
6. Dent, A., Dottax, E.: An overview of side-channel attacks on the asymmetric NESSIE encryption primitives. NESSIE Public Report NES/DOC/RHU/WP5/020/a (May 2002) https://www.cosic.esat.kuleuven.be/nessie/reports/phase2/sidechannels.pdf.
7. Lemke, K., Schramm, K., Paar, C.: DPA on n-Bit Sized Boolean and Arithmetic Operations and Its Application to IDEA, RC6, and the HMAC-Construction. In Joye, M., Quisquater, J.J., eds.: Cryptographic Hardware and Embedded Systems (CHES 2004). Volume LNCS 3156., Springer Verlag (2004) 205–219 http://dx.doi.org/10.1007/978-3-540-28632-5_15.
8. Okeya, K.: Side Channel Attacks Against HMACs Based on Block-Cipher Based Hash Functions. In L.M. Batten, R. Safavi-Naini, ed.: Information Security and Privacy (ACISP 2006). Volume LNCS 4058., Springer Verlag (2006) 432–443
9. McEvoy, R., Tunstall, M., Murphy, C., Marnane, W.: Differential Power Analysis of HMAC based on SHA-2, and Countermeasures. In Sehun, K., Yung, M., Lee, H.W., eds.: Proceedings of the 8th International Workshop on Information Security Applications (WISA 2007). Volume LNCS 4867., Springer Verlag (2007) 317–332 ISBN: 3-540-77534-X.
10. Gauravaram, P., Okeya, K.: Side Channel Analysis of Some Hash Based MACs: A Response to SHA-3 Requirements. In Chen, L., Ryan, M., Wang, G., eds.: Proceedings of the 10th International Conference Information and Communications Security (ICICS 2008). Volume LNCS 5308., Springer Verlag (2008) 111–127
11. Fouque, P.A., Leurent, G., Réal, D., Valette, F.: Practical Electromagnetic Template Attack on HMAC. In Clavier, C., Gaj, K., eds.: Cryptographic Hardware and Embedded Systems (CHES 2009). Volume LNCS 5747., Springer Verlag (2009) 66–80 http://dx.doi.org/10.1007/978-3-642-04138-9_6.
12. Rivest, R.: Side-channel-free timings ? E-Mail to the hash-forum@nist.gov mailing list (Nov 2010) http://cio.nist.gov/esd/emaildir/lists/hash-forum/msg02189.html.
13. Tiri, K., Akmal, M., Verbauwhede, I.: A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. In: European Solid-State Circuits Conference (ESSCIRC). (2002) 403–406
14. Tiri, K., Verbauwhede, I.: A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In: Design, Automation, and Test in Europe (DATE). (2004) 246–251
15. May, D., Muller, H., Smart, N.: Non-deterministic Processors. In: Australasian Conference on Information Security and Privacy (ACISP), Springer-Verlag LNCS 2119 (2001) 115–129
16. IAIK, Graz University of Technology: Power-Trust project website. http://www.iaik.tugraz.at/content/research/implementation_attacks/prj_powertrust/
17. Tillich, S., Kirschbaum, M., Szekely, A.: SCA-Resistant Embedded Processors—The Next Generation. In: 26th Annual Computer Security Applications Conference (ACSAC 2010), 6-10 December 2010, Austin, Texas, USA, ACM (2010) 211–220

18. Bernstein, D.: CubeHash specification (2.B.1). Submission to NIST (Round 2) (2009)
19. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Advances in Cryptology (CRYPTO), Springer-Verlag LNCS 1666 (1999) 388–397
20. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.W.: SHA-3 proposal BLAKE. Submission to NIST (2008)
21. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.W.: OFFICIAL COMMENT: BLAKE tweak. E-Mail to the `hash-forum@nist.gov` mailing list (Nov 2010) `http://cio.nist.gov/esd/emaildir/lists/hash-forum/msg02233.html`.
22. Gligoroski, D., Klima, V., Knapskog, S., El-Hadedy, M., Amundsen, J., Mjølsnes, S.: Cryptographic Hash Function BLUE MIDNIGHT WISH. Submission to NIST (Round 2) (2009)
23. Gligoroski, D., Klima, V., Knapskog, S., El-Hadedy, M., Amundsen, J., Mjølsnes, S.: Clarification on the rotation constant for the variable M_15. Official Comment to hash-forum@nist.gov (Round 2) (Nov 2009) `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/BMW_Comments.pdf`.
24. Bernstein, D.: CubeHash parameter tweak: 10× smaller MAC overhead. Submission to NIST (Round 2) (2010)
25. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak sponge function family main document. Submission to NIST (Round 2) (2009)
26. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak specifications. Submission to NIST (Round 2) (2009)
27. National Institute of Standards and Technology (NIST): Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-3 (Oct. 2008) `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.
28. Bresson, E., Canteaut, A., Chevallier-Mames, B., Clavier, C., Fuhr, T., Gouget, A., Icart, T., Misarsky, J.F., Naya-Plasencia, M., Paillier, P., Pornin, T., Reinhard, J.R., Thuillet, C., Videau, M.: Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition. Submission to NIST (2008)
29. Bresson, E., Canteaut, A., Chevallier-Mames, B., Clavier, C., Fuhr, T., Gouget, A., Icart, T., Misarsky, J.F., Naya-Plasencia, M., Paillier, P., Pornin, T., Reinhard, J.R., Thuillet, C., Videau, M.: Indifferentiability with Distinguishers: Why Shabal Does Not Require Ideal Ciphers. Cryptology ePrint Archive, Report 2009/199 (2009)
30. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family. Submission to NIST (Round 2) (2009)
31. SPARC International, Inc.: The SPARC Architecture Manual, Version 8, 535 Middlefield Road, Suite 210, Menlo Park, CA 94025. (1992) Revision SAV080SI9308.
32. eBACS: ECRYPT Benchmarking of Cryptographic Systems: ECRYPT Benchmarking of All Submitted Hashes. `http://bench.cr.yp.to/results-sha3.html`