NTNU
Norwegian University of
Science and Technology

Performance analysis and enhancement of the iSHAKE algorithm

Jaime Perez Crespo

Master of Telematics - Communication Networks and Networked Services
Submission date: June 2017
Supervisor: Danilo Gligoroski, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Measuring and enhancing the performance of iSHAKE
Student:	Jaime Pérez Crespo

Problem description:

After successfully implementing the iSHAKE algorithm, a proper performance analysis and comparison against other similar algorithms is needed to evaluate its efficiency. Such an analysis will help identify possible improvements that can be done to boost performance and find optimal default settings for the algorithm, as well as to provide a better picture of its strengths and benefits.

The goal of this work is then to perform such comparison and optimise the implementation of iSHAKE, making it available in a public repository and automatically tested for conformance (i.e. host it in github with continuous integration offered by travis-ci). In addition, possible use cases where iSHAKE could be implemented successfully will be studied, such as its application to the git version control system.

Responsible professor:	
Supervisor:	Danilo Gligoroski, IIK

Abstract

In the era of the *Big Data*, immense amounts of data are being generated and processed every second. Research facilities like the European *Large Hadron Collider* (LHC) in Geneva generate massive amounts of information every second[1]. When data in such large volumes needs to be processed by certain cryptographic algorithms to add integrity to it or even digitally sign it, traditional sequential hash functions are unable to cope with it.

Incremental hash functions have been proposed [BGG94] to alleviate some of the issues. However, they imply significant downsides and are still not enough to process the amounts of data that we generate nowadays. An innovative algorithm called iSHAKE has been suggested to allow a more efficient use of parallelism [MGS15] and in our previous work [PC16], we demonstrated the concept with a naive implementation. Now, we present an improved library that allows us to take the most out of modern CPUs, outperforming even the fastest algorithms available, and evaluate the advantages of using iSHAKE for several use cases.

[1]CERN has an illustrative blog post describing the volume of data generated in their experiments, and how to rebuild and store it for its later use by scientific researchers. It can be found here: http://home.cern/about/computing/processing-what-record.

Acknowledgements

I would like to thank my family and friends for their support, and especially Maria for being so patient and understanding.

My gratitude goes to Danilo Gligoroski too for his help and enthusiasm around this project, as well as to my friends and colleagues at UNINETT for making it easier for me to work on this thesis.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface.
BEP	BitTorrent Enhancement Proposal.
blob	Binary Large Object.
CAVP	Cryptographic Algorithm Validation Program.
CAVS	Cryptographic Algorithm Validation System.
CERN	Conseil Européen pour la Recherche Nucléaire (European Organization for Nuclear Research).
CI	Continuous Integration.
CPU	Central Processing Unit.
DDR3	Double Data Rate type 3.
ECRYPT	European Network of Excellence in Cryptology.
FIPS	Federal Information Processing Standard.
LHC	Large Hadron Collider.
MD5	Message Digest 5.
NIST	National Institute of Standards and Technology.
NSA	National Security Agency.
P2P	Peer-to-Peer.
SHA-1	Secure Hash Algorithm 1.

SHA-3 Secure Hash Algorithm 3.

SIMD Single Instruction, Multiple Data.

SSE Streaming SIMD Extensions.

SUPERCOP System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives.

VAMPIRE Virtual Applications and Implementation Research Lab.

VCS Version Control System.

XOF Extendable-Output Function.

Chapter 1

Introduction and related work

A hash function or algorithm can be defined as a mathematical, deterministic, irreversible function that can be applied to data of any length in order to obtain a fixed-size output that corresponds univocally to the input [FS03, Chapter 6].

Functions like that provide certain interesting properties that can be used in computer science. For example, they are useful to reduce big inputs to a fixed-size string that univocally identifies the originating input. Since they are deterministic, the same input produces always the same output, and as such, they can be used to verify the integrity of the input (that is, verify that the input has not been modified in transit or by some malicious third party).

Sometimes, we just don't want to store the original data, but still be able to verify it. Passwords are a good example of this. Keeping a hash of a password instead of the password itself is more secure for the end user (as no system administrator needs to be trusted to hold the user's password and in case of a security breach or a data leak, the original passwords won't be in the public domain), and it still allows us to authenticate the user by hashing the password provided and comparing the resulting hash with the one we have stored. In this use case, though, we need a particularly strong hash function that meets some additional properties.

A cryptographic hash function is a hash function that needs to fulfill certain extra requirements, just because the features provided by the function are paramount to the security of the system relying on them. In addition to being deterministic and irreversible, cryptographic hash functions must adhere to the following principles:

– They need to be *optimised* so that their performance is suitable for most use cases.

– The slightest change in the input must trigger a change of most –if not all– the output, such as no correlation between two almost identical inputs can be established by observing their resulting hashes. This is known as the *avalanche effect.*

– It is infeasible to find (or build) two different inputs that produce the same output. This is called a *collision* in the hash algorithm.

Why are those properties so important? A non-deterministic function cannot be used because it doesn't generate always the same output. Imagine how could we verify a user's password in the previous example if the hash function used does not produce the same output every time we process the same input. Similarly, if the hash function is reversible, we can obtain the input given the output. Going back to the same example again, this would mean the ability to recover the user's password from the stored hash, running the whole system pointless. This is known in cryptography as a *preimage attack*, and the ability of a hash function to resist it is called *preimage resistance* [RS09]. Preimage resistance depends also on the *avalanche effect* because a relation between the output and the input can be used to obtain an unknown input from its output. Imagine an external observer that notes the relationship between both in a certain hash algorithm. Now, in order to recover the input corresponding to a given output (a preimage attack), the observer can compute the hashes of random inputs, keeping the results to analyze the relationships and modify the input fed to the algorithm every time such as its result keeps getting closer to the hash we want to reverse. When both the computed hash and the given one match, we have found the original input.

Finally, when we talk about *collisions* in hash algorithms, we need to make a distinction between two different cases. A *collision attack* consists in finding any two different inputs producing the same hash. Of course, given the fixed-size of the output, the *pigeonhole principle* guarantees that collisions will occur[1]. However, when it is computationally difficult to find those collisions, we say that the hash function exhibits *collision resistance.*

On the other hand, when we are given a certain input and we want to find another one that produces the same hash (as opposed to finding any two inputs), we call that a *second preimage attack.* For a cryptographic hash function, it is essential to offer *second preimage resistance.* Let us present another example for this. Imagine a digital signature scheme where we have both the original input and the corresponding signature, the latter being the hash of the input encrypted with the private key of the signer. If we are able to find any other input that produces the same hash that was encrypted by the signer, we can say that the alternative message was the

[1]The mathematical *pigeonhole principle* states that when a number n of items are to be stored in a set m of containers, with $n > m > 0$, then at least one of the m containers must contain more than one item.

subject of the signature and that won't be deniable (known as *non-repudation*), as the signature will match [Sch96, Chapters 2, 20].

1.1 The scale problem

So far, we have discussed all the security-related properties expected from a cryptographic hash function. However, we left aside one very important aspect: *performance*.

Let's go back to the digital signature example we just used to illustrate the importance of performance in context. A sharp reader could be thinking at this point why using a hash function at all, instead of just encrypting the entire data with the private key of the signer. As it turns out, public key or asymmetric cryptography is incredibly expensive in computational terms, so such an approach would significantly slow down the process and ruin the user experience. This is analogous to public key encryption: instead of encrypting some input with someone else's public key, we use a much faster symmetric encryption algorithm to encrypt with a randomly generated, fixed-size key and then encrypt that key using asymmetric encryption. So, similarly, digital signatures use hash functions to reduce the size of the input to something affordable for the asymmetric encryption algorithm.

When the size of the input to process (whether *process* means here to encrypt it or sign it) grows, the algorithm that does the initial processing needs to be extremely fast as the demand for computational resources has been shifted from the public key algorithm to the symmetric encryption or hashing ones.

Even though algorithms evolve and with the years we change from using one to the other, their core principles remain the same. On the other hand, we have increasing amounts of data sources generating much more information than we can process. In some cases, that information needs to be processed in a way that requires the use of a hash function, and if that function is not able to cope with the size of the data it is being fed, we can then say that the system does not scale.

Let us present one last example to illustrate this problem. Assume we have a log of events that grows when each new event is appended to the end. Now, imagine that events are produced by hundreds of devices connected to the network, and that, on average, the log has a new event appended every second. If we want to use that log as a *papertrail*, we need to make sure to add integrity to it, so that we can demonstrate in court that it has not been altered. Maybe we even want to digitally sign it. Let's pretend that the hash algorithm we use for that takes 1.5 seconds to process the entire log at its current state, which in turn keeps growing and growing (leading to longer processing times) while more devices keep sending their events (eventually leading to more frequent events). If the rate in which we receive events is higher than the rate in which we process them, the system simply chokes and stops working.

This example is a good depiction of the problem we attempt to solve here. We not only need a hash function that performs better than any other, but also a model that allows us to scale in use cases like that, completely independent of the size of the input.

In general, a typical approach to resolve any scalability or performance issues consists in allowing the use of parallelism to distribute the work among different workers and collect the results when they are ready to combine them in one final, unique outcome. However, most hash functions tend to process the input data sequentially, in a way tha does not allow parallel processing to improve the performance.

Our proposal, called *iSHAKE*, attempts to resolve this problem not only dramatically enhancing the performance of traditional, sequential hash algorithms, but also changing the paradigm to allow the modification of the input with the recomputation of the resulting hash in constant time ($\mathcal{O}(1)$) without compromising any of the fundamental properties of a cryptographic hash algorithm.

1.1.1 Merkle trees

An intuitive way to resolve the problem at hand would be to build a tree whose leaves are equal-size blocks containing the input data split in order, then apply the hash function to those leaves and use the resulting hashes as input for another iteration of the algorithm, until only one hash operation is performed. In computer science, this is called a *hash tree* or a *Merkle tree* [Mer82], and the main advantage is that blocks can then be hashed in parallel. An additional feature provided by this model is that a change in a particular block does not require to process the entire input again, but only the block that has been modified, plus all the intermediate hashes that connect that block to the top of the tree. Figure 1.1 shows an example of such a structure.

Unfortunately, Merkle trees have several disadvantages that make them less suitable as a solution in this context:

- If being able to recompute the resulting hash without processing the entire input is a requirement, all the intermediate hashes must be kept for every given input, constituting a significant data overhead.
- Sequential processing of the input is even slower than when using a regular hash algorithm (if no parallelism is used), as more hash operations are needed (one for each node of the tree).
- Recomputing the resulting hash after a block has changed in a Merkle tree needs logarithmic time ($\mathcal{O}(log\ n)$), with a total number of $log_2(n) + 1$ hash

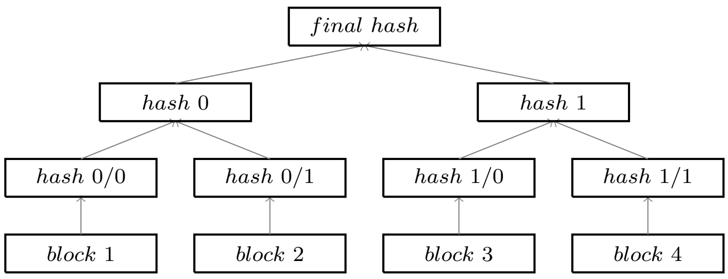

Figure 1.1: Example of a Merkle tree with four blocks. The input is divided into equal-size blocks, which are then processed using the hash function. A binary tree is built out of the resulting hashes, subsequently applying the hash function to them in a loop, until one single hash operation is performed to obtain the final hash.

operations. While this is a significant improvement over the linear time needed by sequential algorithms ($\mathcal{O}(n)$), it could still be much better.

– Hash operations can only be executed in parallel in the same level of the tree, as for each level the hash operations require the hashes of the next level to be available. This means, for instance, that the hash operations needed to recompute a hash upon a change in a block cannot be performed in parallel.

This set of issues makes it less convenient to use regular Merkle trees to solve our problem, and we need therefore a different solution.

1.1.2 Incremental hashes

A more generic way to approach the problem is usually known as incremental hash algorithms [BGG94]. While a Merkle tree can be considered an incremental hash algorithm itself, multiple proposals exist suggesting the reduction of the depth of the tree (effectively improving the performance) or even the use of different functions to process the input depending on the depth of a particular node in the tree [BDPVA14]. Figure 1.2 gives an overview of their behaviour.

Some incremental hash algorithms can provide a significant improvement in performance respect to Merkle trees. However, they still imply a big data overload as we need to keep all the intermediate hashes computed in the tree if we want to be able to modify the final hash upon a change in the input. This limitation is always present unless we process the intermediate hashes in a different way.

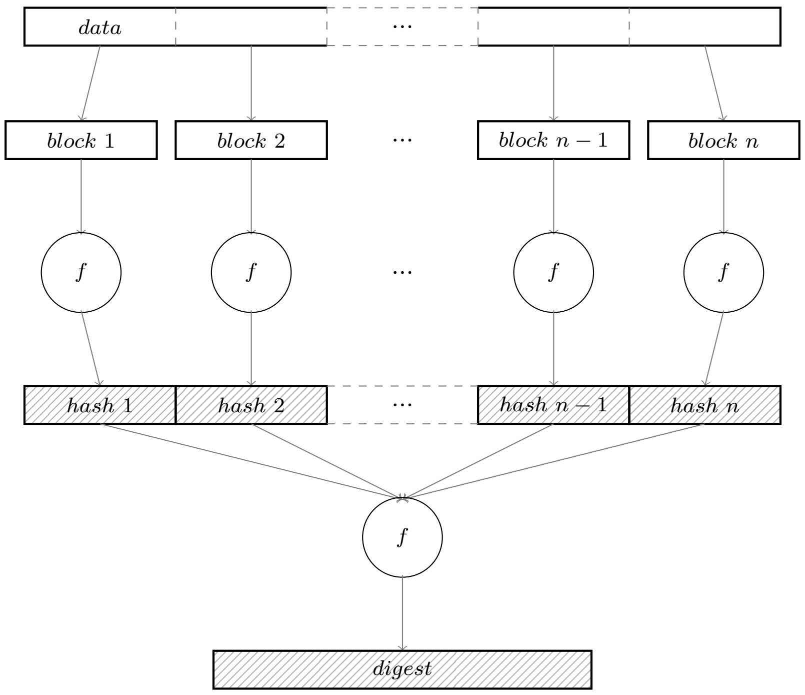

Figure 1.2: Example of an incremental hash algorithm. In this case, the incremental hash algorithm divides the input data into blocks, and applies the hash function over each block, concatenating the results in the same order as the blocks. The resulting concatenation of hashes is passed again to the input of the hash function, obtaining the final digest. Different incremental hash algorithms may have more levels of indirection, increasing the amount of intermediate hashes used.

1.2 iSHAKE

As we have seen, incremental hash algorithms come at a price in terms of data overhead that we need to keep to get the most out of them. An alternative to a tree-based incremental hash algorithm is *iSHAKE* [MGS15], which proposes to stop building hash trees and combine the intermediate hashes with an efficient commutative operation that can be reversed. If we can reverse this *combine* operation, we can then remove, update and add new blocks without the need to keep all intermediate hashes, just by recomputing the hash corresponding to the block or blocks affected. Figure 1.3 illustrates this proposal.

Our previous work presented an initial implementation of iSHAKE [PC16], featuring the ability to insert, update and delete blocks from any given hash and its corresponding original input. Having the ability to perform those operations is key

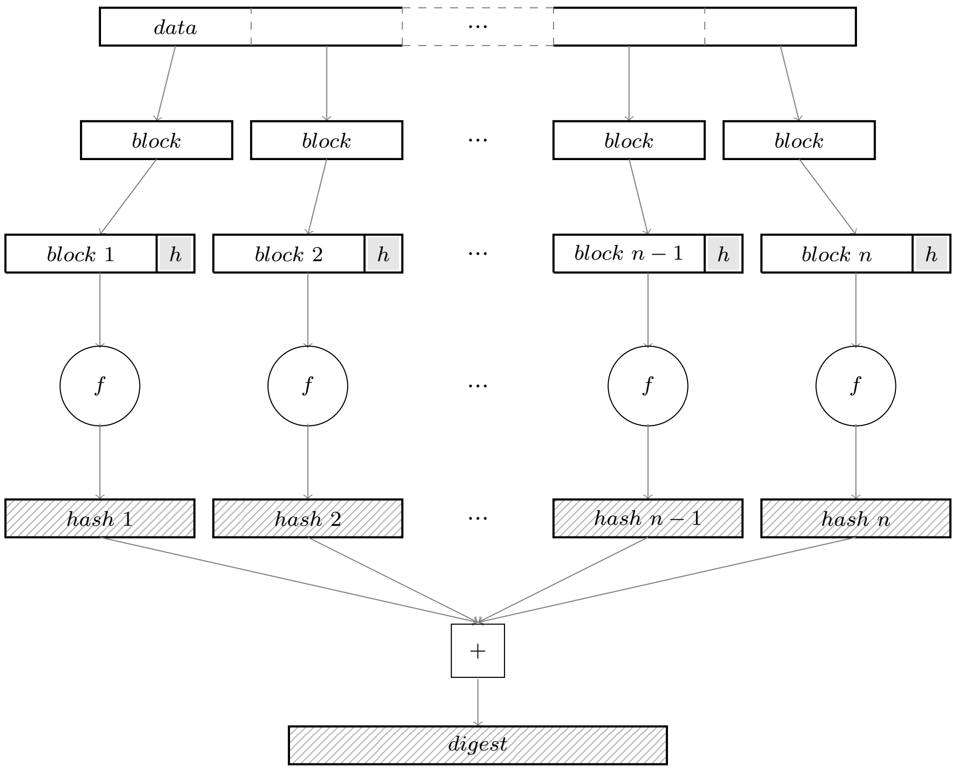

Figure 1.3: General description of the iSHAKE algorithm. The input is split into blocks of size `block size`. Each block is appended with a header of variable size depending on the mode of operation. The hash function f is then applied to each block, and the resulting hashes are combined together with a commutative, reversible operation to form the final digest.

to guarantee a constant time for recomputations of the resulting hash upon a change in the input, while significantly reducing (or even removing) the data overhead introduced by most incremental hash algorithms. We made this implementation publicly available[2] for anyone willing to test it and used a system similar to NIST's Cryptographic Algorithm Validation System to test our code automatically thanks to a Continuous Integration platform.

Now, we present an improved implementation providing complete parallelization of all the operations and improving the performance dramatically, up to the point that it can compete with the fastest hash algorithms in the market.

[2]See http://github.com/jaimeperez/ishake.

1.3 Outline of this report

This project demonstrates in practice the main advantages of the aforementioned iSHAKE incremental hash algorithm. We start by describing all the fixes and improvements made to the original implementation in Chapter 2. Next, we present a set of experiments as well as their results, comparing iSHAKE with other sequential and incremental hash algorithms in Chapter 3. We will also study the possible application of our solution to existing, real-world use cases, in Chapter 4, and finalise presenting our conclusions in Chapter 5.

Chapter 2 Improving the implementation

The first goal of the work presented here was to improve the existing implementation as much as possible, making it suitable for use in production environments, and enabling us to measure its performance in fair conditions against other existing algorithms. As such, a full code review was carried out to identify possible bugs and weaknesses, as well as possible improvements that could emerge from changes in the design decissions originally taken.

This chapter presents that work in a comprehensive and exhaustive manner. We will go through the different issues found during the code review and explain their fixes in detail. We will also review the improvement areas outlined in our previous work [PC16] and discuss the actions taken to tackle them.

2.1 Bug fixes

Software has bugs. We could even argue that bugs are an inherent property of software, based on the fact that software is written by humans who will inevitably make mistakes. Therefore, it is our responsibility to acknowledge this as a fact and assume our software will have bugs that we need to fix. This is the only way we can actually focus on tracking them down, understanding why they happened, and providing appropriate fixes. During our code review, we identified and fixed a set of bugs that we describe in this section.

2.1.1 Memory management

Programming in C [ISO99] is difficult. The amount of freedom the language provides the programmer with comes at a price, that being the complexity and the myriad of details that need to be taken into account even in the simplest algorithms. Memory management is one of the most typical examples of this, especially for those programmers used to languages that take care of memory automatically, so that its management is completely transparent to them and they are not used to manually allocating and freeing memory to store their data.

We identified an issue with memory management in our iSHAKE implementation, related to the way we passed iSHAKE blocks around to the API functions. In an attempt to make it easier to handle variables, *pass-by-reference* arguments were used only when the block was supposed to be modified by the function call (e.g. during a block insertion or deletion, the block previous to the one we are inserting or deleting needs to be modified to point to the appropriate next block in the chain), and *pass-by-value* arguments were used the rest of the time.

This has proven to be a mistake, as in C passing an argument as a reference does not only imply the capability to modify the contents of the referenced variable, but also different areas of memory being used. In this particular example, an iSHAKE block contains, in turn, more references to memory addresses that contain the data of the block, and that data is not copied over to the stack when using *pass-by-value* arguments, but the reference to the data itself.

In a scenario with sequential processing, this would not show up as an issue because blocks are processed one at a time, and that means we won't release the memory associated with a block before we are done processing it. However, when parallelism is introduced, multiple branches of execution would be competing to access the same data (remember we are using references instead of copying data over from one function to another), and race conditions occur. The most typical example was some code freeing the memory where we had read the data of a block, while another code was waiting its turn to process that data, leading to a later access to unallocated memory.

This is obviously a serious issue that could lead to all kinds of errors, from segmentation faults due to code accessing memory that was previously freed in the worst case, to incorrect results of the algorithm due to the processing of data in memory that doesn't correspond to the real input.

The way to resolve a problem like this is using *pass-by-reference* arguments regardless of whether we intend to modify the contents of an argument or not. Defining a new `ishake_block` type as a reference to a `ishake_block_t` struct makes it possible to harmonize all function signatures and implementations. Additionally, all dynamically-allocated memory is now freed by the code that processes the contents stored in that memory, instead of by the code that allocated that memory in the first place. This might be more complicated since it is then easier to lose track of allocated memory and give room to memory leaks, but it is the only way to avoid the issue described here without using complicated, additional synchronization mechanisms.

2.1.2 Modulo operations

Another typical source of bugs in C is the so-called *off-by-one*. Those issues are usually due to the programmer counting one less (or one extra) item in a sequence, ignoring the fact that counts start at `0` in the language. It can also happen when performing computations that depend on the number of bits used (e.g. confusing the power of two with the number of bits in a word). The latter was the case when computing the modulo operations that combine together the hashes of every block.

In iSHAKE, the results of applying the hashing algorithm to each block are combined together using a commutative operation per each 64-bit word. If, for example, we have an input divided into two blocks, the two digests resulting from applying the hash algorithm to each block are merged by iterating over each 64-bit word of the digests and adding both together, modulo 2^{64}. This modulo operation is needed to ignore overflows when the addition of both input integers cannot be expressed in 64 bits. It also makes it easy and fast to compute the operation in 64-bit architectures.

The issue at hand was due to the modulo operation using the `UINT64_MAX` constant defined in `stdint.h`. Since a 64-bit computer cannot handle numbers greater or equal to 2^{64} directly (as that value corresponds to the 65th bit), `UINT64_MAX` represents the maximum value an unsigned, 64-bit integer can take, which is $2^{64} - 1$ (64 bits set to 1). This means we were applying a modulo operation that did not allow the highest number possibly contained in a 64-bit word ($2^{64} - 1$), with fatal consequences under certain circumstances.

For example, if we add `0x00` to the maximum unsigned 64-bit integer (`0xFF FF FF FF FF FF FF FF`) we should return the same number, as we are adding zero. The result of this operation was zero, though, as `0xFF FF FF FF FF FF FF FF` is $2^{64} - 1$, and every number modulo itself yields zero as the result. In the case of an overflow, the result of the operation was short by one. For example, adding `0x01` to `0xFF FF FF FF FF FF FF FF` should yield `0x00` as a result, but `0x01` was returned instead, due to the way the modulo operation was implemented (remember that the number 2^{64} cannot be represented in a 64-bit computer directly).

The issue was resolved by letting the processor handle the addition (or subtraction) itself, which basically means ignoring the overflow and performing the modulo operation automatically, instead of implementing that modulo operation ourselves. This way, the bug due to the upper bound being short by one is no longer an issue, as that bound is set by the architecture of the processor, not by the programmer.

2.1.3 Empty inputs

Every hash algorithm must be able to process an empty input and produce a deterministic output for it. While this held true for iSHAKE in *append-only* or *fixed-size* mode, it did not for the alternative *full* or *variable-size* mode. The issue, located in the `ishake_append()` function of the API provided, was due to a bug when no input had been processed at the time of calling. The condition evaluating whether we need or not to process any remaining data did not take into account the mode in use, but the number of bytes awaiting for processing or the number of bytes already processed.

This condition alone worked fine when running in *append-only* mode. This is because, in this mode, the header of the blocks contains a self-incrementing index. When no data was provided at all, the condition would hold true as no bytes were processed so far. The code would then add a new block with the remaining data (in this case, no data at all) and the current index plus one, that being 1 as the index was initialized to 0. So in case no data is passed to iSHAKE at all, an empty block indexed with number 1 would be the input for the underlying hash function.

However, this behaviour does not work in *full* mode, because the block header consists of a *nonce* identifying the block, and the *nonce* of the next block in the chain. Since no data was provided, an empty block could be used, but the algorithm doesn't have a *nonce* to identify that block. While the required *nonce* could be randomly generated for this case, that would break the mandatory requirement to produce deterministic output, as the result would vary depending on the aforementioned *nonce*.

The fix for the issue consists in a check for the mode of operation, skipping the processing of any remaining data or an additional empty block when running in *full* mode. We keep that way the expected behaviour in *append-only* mode while leaving the responsibility to add an empty block to the user. This means iSHAKE can generate a total of 2^{64} outputs from an empty input, depending on the *nonce* associated with the empty block.

2.2 Improvements

During our previous work [PC16] we identified several areas where iSHAKE could be improved significantly in terms of performance or usage simplicity. Additionally, our tests during the present work showed additional room for improvement. Here, we outline those improvements and make some usage suggestions based on our experience.

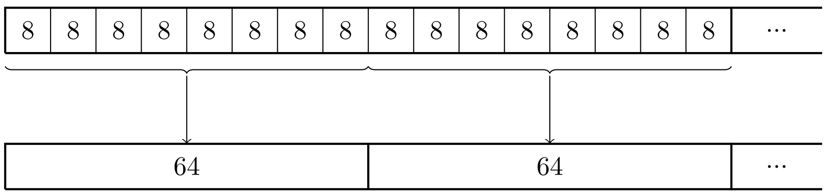

Figure 2.1: Conversion of an array of bytes to an array of unsigned 64-bit integers.

2.2.1 Frequent operations

One of the areas of improvement we previously identified was the operations that are executed most frequently. Apart from the underlying hash algorithm, we also perform several operations for each processed block:

- The conversion of the resulting hash from an array of bytes to an array of unsigned 64-bit integers, as depicted in figure 2.1. This is needed in order to perform the commutative operation from the output generated by the underlying hash function.
- A commutative operation to combine the computed digest of the block with the global digest computed so far. This is done by adding or subtracting together unsigned 64-bit integers, as depicted in figure 2.2.

Both operations take a non-negligible amount of resources since they essentially need to iterate over the resulting digest with linear complexity ($\mathcal{O}(n)$) depending on the length of the digest. After testing different alternatives with regard to the implementation, we observed no significant improvement in any of them, concluding that compiler optimisations are good enough to optimise both operations without the need for further optimisation in the code itself.

2.2.2 Keccak implementation

In our initial implementation of iSHAKE, described in [PC16], we decided to use a tiny implementation of the Keccak sponge function[1]. As we explained then, the decision to use that implementation was based on the assumption that it was nearly

[1]The *keccak-tiny* library is the work of David León Gil, and it is available under a *Creative Commons 0* (CC0) open source license here: https://github.com/coruus/keccak-tiny.

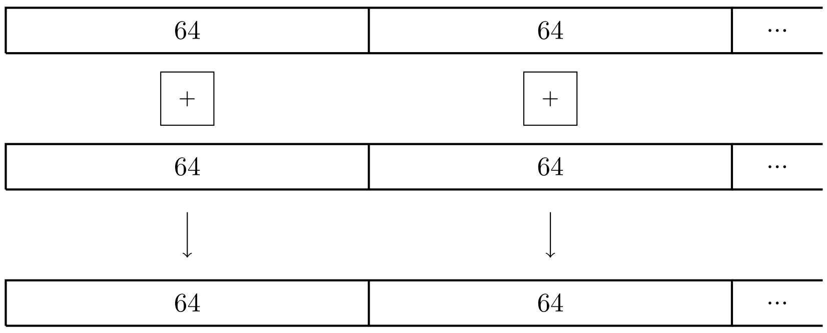

Figure 2.2: Commutative addition operation performed over two arrays of unsigned 64-bit integers.

as fast as the original implementation by the KECCAK team[2] known as the KECCAK *Code Package*[3], and that it was simple to use and build.

Our initial evaluation about the simplicity of this implementation was too optimistic, as we had to adapt it to our needs in order to be able to process large sets of data in chunks. With regard to its performance, our tests showed that our initial assumption was, unfortunately, wrong too. The official implementation can be up to a couple orders of magnitude faster than the *keccak-tiny* library, and that had a serious impact on the performance of our iSHAKE library itself. As an example, while we measured approximately 103 seconds to process 1 GiB of data with the iSHAKE 256 algorithm using the *keccak-tiny* implementation, our new development using the KECCAK *Code Package* takes less than 4 seconds to process the same data in one of our test machines.

The difference in performance is really noticeable and critical for our purpose to demonstrate the efficiency of iSHAKE when competing with other hash algorithms. Therefore, the iSHAKE implementation has been updated to use the KECCAK *Code Package*, allowing for architecture specific optimisations. This official KECCAK implementation provides multiple builds optimised for different architectures, with the most meaningful for our interests being the generic optimisation for 64-bit platforms, and the *lane complementing transform* optimisation, described in [BDP+12].

As we will demonstrate later in Chapter 3, the change of the underlying KECCAK implementation allows us to dramatically improve the performance of iSHAKE to

[2]The KECCAK team is the group of authors of the KECCAK sponge function that was finally chosen by NIST to become the SHA-3 standard [Nat15]. This group of authors are also the proponents of the SAKURA tree-based incremental hash algorithm [BDPVA14]

[3]This is the official implementation published under a *CC0* open source license by the KECCAK team here: https://github.com/gvanas/KeccakCodePackage.

be able to compete with other extremely fast hash algorithms and even outreach their performance under certain circumstances, unleashing all the advantages of our model.

2.2.3 Parallel computation

The iSHAKE algorithm has two main advantages over regular, sequential hash algorithms. First, it allows for very efficient hash recomputing when the input changes, without the need to process the entire input again. Second, parallel processing of the input is possible as it is divided into self-contained blocks that can be passed directly to the underlying hash function.

During our previous work, we demonstrated its efficiency recomputing a hash upon a change in the input, and our new implementation based on the KECCAK *Code Package* will help us compare it to other similar algorithms. However, that original implementation lacked the ability to exploit processor parallelism and was, therefore, equivalent to any other sequential processing algorithm.

A new implementation has been made to allow the programmer to specify the number of threads to use for processing. We used a *worker-dispatcher* model, where the main thread of execution (the *dispatcher*) reads the input data and divides it into blocks. Those blocks are appended their corresponding header and added to a *task* structure that indicates what operation to perform on the given block. Tasks are pushed to a heap, from where a set of worker threads pull the tasks for processing. In this case, the completion of a task involves both the computation of the digest corresponding to the block and the combination of that resulting digest into the final digest. Figure 2.3 describes the model as implemented. This particular approach allows us to take the most out of the available resources to compute the hash corresponding each block and combining it with the current hash with the commutative operation. The downside to our model is the complex synchronization that is required between the dispatcher and the worker threads, and also among the latter.

Note that our implementation makes use of *POSIX threads* (commonly known too as *pthreads*), meaning the library can be used only in systems with support for them. Most *UNIX-like* operating systems typically include versions of *POSIX threads*, like *Linux*, *macOS*, *Solaris*, *Android* or the *BSD* family of operating systems. *Microsoft Windows* provides a subsystem implementing parts of the *pthread* API, but no attempts to build the library on that platform have been made, and therefore we can't offer any guarantee that it would work on it. In any case, and while proper support for native threading in Microsoft platforms is developed, the *pthread* library remains as a new dependency when building iSHAKE.

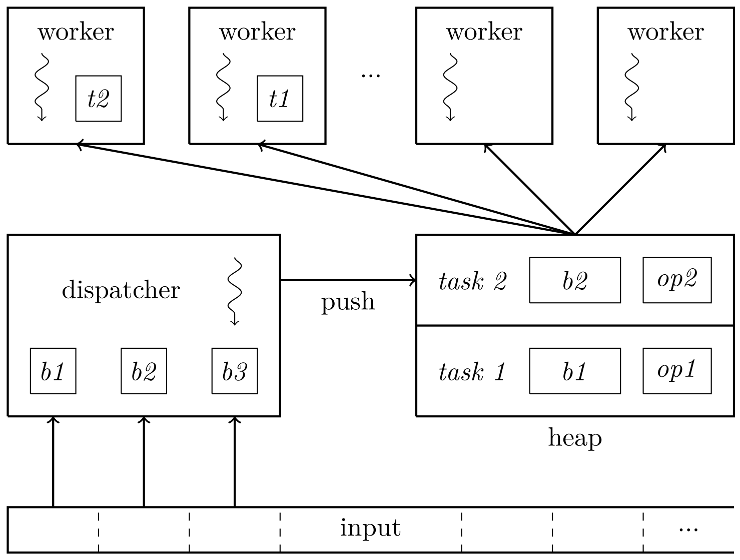

Figure 2.3: Parallel processing of iSHAKE blocks by means of a *worker-dispatcher* model.

This improvement is key to demonstrate iSHAKE's abilities compared to other hash algorithms. While most functions are optimised to be executed taking the most out of a single processor, using, for example, SIMD instruction sets (*Single instruction, multiple data* [HP11]) like Intel's *Streaming SIMD Extension* (SSE* [Int17]), iSHAKE is designed to take the most out of parallel architectures where multiple cores are available, a situation typical nowadays in most CPUs. Taking this even further, and given the similarity of our model with the *split-apply-combine* strategy for data analysis [Wic11], we can easily adapt the algorithm to work with similar or derived models like Hadoop's *MapReduce* [DG04], in order to take advantage of high-performance computing clusters. On the other hand, iSHAKE can of course benefit from any optimisations provided by the underlying hash function too, maximizing the performance both per block and per input.

2.2.4 Appending blocks efficiently in *full* mode

During our previous work [PC16], we noticed a design optimisation that can be performed to iSHAKE in *full* mode. Appending blocks by the end of any given input is extremely efficient in *append-only* mode. This is because, in that mode, indices are used to identify each block, instead of random *nonces*. This means we can avoid the need to build a linked list of blocks to keep track of the relative order between them, and therefore we don't need to keep any additional data other than the resulting digest. However, an interesting consequence of this simplification is that we don't need to apply the underlying hash algorithm to any existing block in order to append a new one.

In *full* mode, though, we need to keep such a linked list of blocks to keep their relative order, as the order cannot be established using random identifiers instead of an index. When the linked list is built forward, that is, each block has a pointer in its header to the next block in the list, appending a block to any existing input means that we need to hash the new block and modify the last existing one. Block updates in iSHAKE require two hash operations, since the old version of the block needs to be removed from the resulting hash first, and then the new version of the block (with the same identifier, but now also with a pointer to the newly appended block) needs to be added. Therefore, appending a new block implies three hash operations in *full* mode, compared to the single hash operation needed in *append-only* mode. This can have a significant impact on the performance of the algorithm in *full* mode, as it can take up to three times the time needed to process any given input compared to *append-only* mode.

Given that any input data would initially be read sequentially (from the beginning to the end), when using iSHAKE in *full* mode, every block processed by the algorithm implies three hash operations instead of just one. The obvious solution to this issue is to invert the direction of the linked list of blocks, such as blocks have a pointer to the *previous* block instead of the *next* one. That way, when appending blocks by the end of the chain, the last block does not need to be modified at all, and only one hash operation is needed in turn.

The drawback of using a reversed list of blocks is that prepending blocks to any given input is now much more inefficient, as it then needs three hash operations instead of the single one needed before. However, we consider this case where blocks are prepended much less frequent than its counterpart, and therefore this change in the design of iSHAKE has been implemented in the reference library, changing also the API provided in the signatures of the `ishake_insert()` and `ishake_delete()` functions.

2.2.5 Efficient use of memory

The iSHAKE API provides the programmer with a significant freedom over the parameters of the algorithm. One of the most critical parameters with regard to performance is the block size to use. It affects not only the performance but also the result of the algorithm. While this freedom is an advantage from the user's point of view, as they are able to fine tune the algorithm for their needs, some guidance is needed to make sure the best choices are made.

When we are concerned about optimising the performance of an algorithm, we need to make sure that the computer running it will use its resources in the most efficient way. This includes maximizing the use of memory so that both the operating system and the machine do not waste it by dividing it into fixed-size chunks.

Modern 32 and 64-bit architectures use a minimum page size of 4 KiB [Int17]. Some other architectures like the *UltraSPARC Architecture 2007* require a minimum page size of 8 KiB [Ora10]. Considering this, it is a sound idea to use block sizes such as they are multiple of a full page. Choosing the best block size is a complicated task depending on multiple factors, such as the estimated size of the data to process (whether it's a small, medium or large input), any previous restrictions, et cetera.

We recommend in any case using block sizes that are multiple of the page size on all architectures where the algorithm is going to be used. This gives us the flexibility to adapt the block size to whatever fits best for the average size of input that we are going to process, but at the same time makes sure that blocks will fit perfectly in a certain number of pages. As a general recommendation, block sizes should be a multiple of 8 KiB to cover most common architectures.

An additional consideration needs to be made regarding the difference of block size and the amount of data included in a block. In our previous work, no such difference was made [PC16], leading to blocks of *block size* + *header size* length, as described by picture 2.4. This, however, makes it difficult to use optimal block sizes, as they would be affected by the mode the algorithm is running on (due to the difference in the length of the headers used).

We suggest changing the paradigm so that the header is part of the block size, as depicted in figure 2.5. That way, the block size can be fixed regardless of the execution mode and the header size, and it is then the amount of data that fits into the block that varies depending on each case.

The API provided by iSHAKE did not make any assumptions, so it was the responsibility of the programmer to pass a block size to the `ishake_init()` function that takes into account the size of the header. The library has been modified accordingly to make it simpler to use so that the block size specified during initialization includes the size of the corresponding header. This change affects the `ishake_append()` function, which is the only function in the API that builds blocks itself. For the rest of the functions, the programmer is responsible for providing blocks that fit (including their headers) completely within the block size specified during initialization.

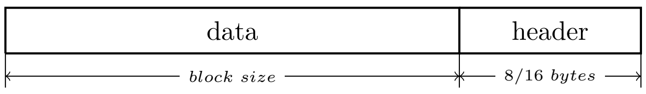

Figure 2.4: Original iSHAKE blocks. The data included in the block is appended with a header that will vary its size depending on the mode of operation of the algorithm.

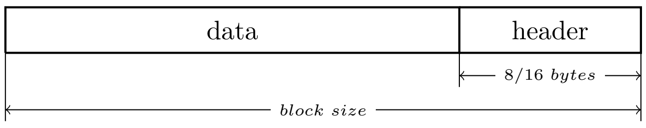

Figure 2.5: Optimised iSHAKE blocks. The amount of data in bytes per block equals the block size minus the number of bytes reserved for the header.

2.2.6 Python interfaces

The iSHAKE library comes with a couple of utilities that allow us to use the library through a command-line interface, the `ishakesum` and `ishakesumd` binaries. While the former allows us to hash completely any input using iSHAKE, the latter uses its own file name conventions to allow us to recompute the hash for any given input, inserting, deleting or updating blocks from files in a directory structure. Both utilities are however difficult to use and require significant infrastructure in order to use the iSHAKE commands.

We have implemented a wrapper library in *Python* to allow programmers of this language to use iSHAKE easily while retaining the performance optimisations of the implementation written in C. In conjunction with the *Python Fire* library provided by Google[4], an interactive command-line utility was also implemented, allowing users to hash any input and perform iSHAKE commands interactively with very simple code written in Python. This way, users can split the input into blocks and handle those programmatically to pass them to iSHAKE, and use its commands to recompute any existing hash. This is an important step forward in making iSHAKE more flexible and easier to use.

[4]*Python Fire* is a Python library to create interactive command-line interfaces from any Python object. See https://github.com/google/python-fire for more information.

Chapter 3

Performance evaluation

In our previous work [PC16] we focused on an implementation of the iSHAKE algorithm that would allow us to demonstrate its intuitive superiority over traditional sequential hash algorithms when recomputing a hash for an input that has been changed. Our experiments proved us right, showing a dramatic improvement when we need to recompute a hash for large inputs. However, the underlying implementation of the SHAKE extendable output function that we used at the time was far from optimal and caused our results to be significantly slower than they should.

Additionally, the lack of a parallelized implementation of the algorithm made it also difficult to test the performance of iSHAKE in a different scenario: hashing a large input from the beginning to the end, leveraging parallelism to speed up the process.

Both issues have been resolved in the library, and our current iSHAKE reference implementation is now optimised for most architectures thanks to the use of the *Keccak Code Package* and built-in support for threads, as described in sections 2.2.2 and 2.2.3, respectively. This allows us to focus on testing the performance of the algorithm when processing entire inputs instead of recomputing an existing hash and comparing that performance to other existing algorithms, both new and well established. Additionally, we test iSHAKE with different settings in terms of block size and the number of *worker* threads, in order to assess the best configuration with regard to the performance of the algorithm when processing large inputs.

In this chapter we present the results of our tests, as well as the methodology and infrastructure used to take the measurements we needed (see Appendix C for the full list of measurements used through this text), and introduce the algorithms we compare against.

3.1 Methodology

In order to carry out our tests in the most reliable way to get significant, trustworthy results, we have used different platforms and software. The main rule is to execute the test in isolation, with no other software running at the same time, except for the basic services provided by the operating system. We diversified the types of platforms and operating systems used too, in the hope to find meaningful differences that could lead us to further optimise the algorithm.

In some cases, we reused existing benchmarking tools to test other algorithms. When testing iSHAKE, though, a custom benchmarking tool was built in C in order to minimize any external factors that could add a bias to the results or affect them in any way. The source code of this tool (called *testPerformance*) is also distributed with iSHAKE and can be adapted to perform the tests we need.

In general, we follow the same approach as the SUPERCOP toolkit[1] developed by the VAMPIRE laboratory[2], the third laboratory run by ECRYPT II[3]. Inputs of different sizes are passed through the tested algorithms while counting the number of CPU cycles used in the processing. That number is then divided by the total amount of bytes in the input in order to obtain the number of cycles used per byte by the algorithm, giving a more precise measurement of its efficiency that can be easily compared.

Where measuring the CPU cycles taken by the algorithm is not possible (for example, when measuring algorithms for which we do not have a benchmarking tool), the standard command-line utilities are used, and we measure the *wall time*[4] taken to process the input. Even though *CPU time*[5] could be considered more accurate to test the time taken by an algorithm to complete, we decided not to use it for two different reasons:

[1]SUPERCOP stands for *System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives* and it measures the performance of hash functions and ciphers. See https://bench.cr.yp.to/supercop.html

[2]The *Virtual Applications and Implementations Research Lab* focuses their work on the implementation of cryptographic algorithms, with a special focus on efficiency and security. For more information, see http://hyperelliptic.org/ECRYPTII/vampire/.

[3]The *European Network of Excellence in Cryptology II*, or ECRYPT II, is a European initiative funded by the European Commission, under contract number ICT-2007-216676. It aims at the improvement and strengthening of the state of the art and adoption of cryptology, as well as to facilitate cryptographic research on a pan-European level. See http://www.ecrypt.eu.org/ for more information about this initiative.

[4] *Wall time* is the time elapsed according to the computer's internal clock, and should match the time measured by an external observer.

[5] *CPU time* is the time the CPU spends executing *user code*, that is, the code of the algorithm being measured without any system calls or any other code run in the kernel of the operating system.

- As we are interested in measuring the performance of iSHAKE for real-world use cases, it is useful to know the time taken by the algorithm as experienced by the end user, rather than the time taken by the CPU to execute it, which could be misleading depending on external factors.

- Given that user code and system calls are interleaved and it is therefore very difficult to measure the time taken only in *user land*, the operating system will use internal mechanisms to measure that time instead of just sampling the start and end time for the entire execution. This has a side effect that punishes particularly the measurements in iSHAKE, as CPU time will include the time taken by all threads of execution to complete. This means that even if the algorithm completes very fast by leveraging parallelism, CPU time will be much higher than in other algorithms, as it will measure the execution time of each thread and sum all of them together.

We present here the results for two kinds of experiments. First, we will change the configuration options of iSHAKE and measure the performance when processing a variable-size input in order to determine the best setup, and establish some basic guidelines to optimise performance. Secondly, we will run the same input through different hash algorithms while measuring the time taken to process it, so that we can compare them in terms of speed.

Every measurement presented here has been taken as either the average or the best figure observed for a set of repeated executions of the tested algorithm with the same input and configuration unless explicitly stated otherwise. Please refer to Appendix C for the complete list of measurements used to compile the results presented in this chapter.

Finally, input data is generated dynamically with the `dd` command, using the `/dev/zero` virtual device as input, and producing blocks of a size equal to iSHAKE's block size minus 8 bytes for its header (we use *append-only* mode for simplicity). This data is passed then via a *pipe* mechanism to the appropriate command so that it resides in memory at all times and we avoid any external bottlenecks such as disk storage.

3.2 Test platforms

Before we go on and present the result of our experiments, we need to describe the two different platforms used to carry the tests out. Two different machines were used, a laptop computer and a server machine, called *nepenthe* and *bigmem*, respectively. Here we present all their main characteristics in terms of both software and hardware.

nepenthe

This is a laptop computer manufactured by Apple Inc. during 2015, with the following characteristics:

– *Architecture*: AMD64
– *Microarchitecture*: Haswell/Crystalwell (0306C3h)
– *CPU manufacturer*: Intel Corporation
– *CPU model*: Core i7 4980HQ
– *CPU year*: 2014
– *CPU speed*: 2.8 GHz
– *Number of cores*: 4
– *Number of threads*: 8
– *Number of CPUs*: 1
– *Total number of threads*: 8
– *Level 1 cache*: 64 KiB (per core)
– *Level 2 cache*: 256 KiB (per core)
– *Level 3 cache*: 6 MiB (shared)
– *Memory size*: 16 GiB
– *Memory speed*: 1600 Mhz
– *Memory type*: DDR3
– *Operating system*: macOS Sierra 10.12.5

bigmem

This is a server computer hosted in the Department of Telematics at NTNU, used mainly to benchmark software and test use cases that require large amounts of memory (hence its name). It has been used indeed to run SUPERCOP benchmarks in the past. Here are the relevant characteristics:

– *Architecture*: AMD64
– *Microarhitecture*: Nehalem (206e6)

- *CPU manufacturer*: Intel Corporation
- *CPU model*: Xeon X7560
- *CPU year*: 2010
- *CPU speed*: 2.26 GHz
- *Number of cores*: 8
- *Number of threads*: 16
- *Number of CPUs*: 4
- *Total number of threads*: 64
- *Level 1 cache*: 64 KiB (per core)
- *Level 2 cache*: 256 KiB (per core)
- *Level 3 cache*: 24 MiB (shared, per CPU)
- *Memory size*: 1 TiB
- *Operating system*: Ubuntu Linux 16.04.2 LTS (GNU/Linux 4.4.0-78-generic x86_64)

3.3 Optimal configuration

For this experiment, we change the configuration of iSHAKE 128 and evaluate its performance for variable-size and fixed-size inputs. In this case, we would like to establish a method for determining optimal configuration parameters under given circumstances, and therefore the tests were carried out only in the machine called *nepenthe*. The results presented here are not to be taken as default values for the algorithm (although they could present a valid guideline to choose those defaults), but just as a way to find those defaults depending on the use case at hand. Furthermore, we feel the need to emphasize that changing certain settings of the algorithm would result in a change of the output itself, as it is the case with the block size.

First, we pick different block sizes (500 bytes, 1 KiB, 10 KiB, 100 KiB and 1 MiB, respectively), and measure the cycles per byte employed by iSHAKE to process variable-size inputs using different amounts of worker threads (from zero up to twice the number of CPU threads available minus one). For each configuration (block size and the number of threads), we process the input twenty times after calibrating our instrumentation tools, and we pick the best value in the series (that being the lowest amount of cycles per byte). Input consists always of 1024 blocks of the size given, hence resulting in different overall lengths.

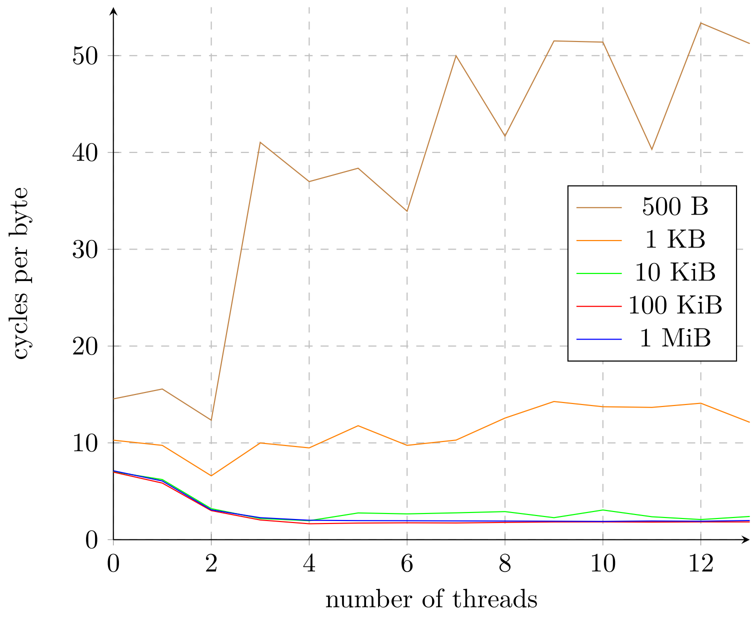

Figure 3.1: CPU cycles taken by iSHAKE 128 to process each input byte for 1024 blocks of variable size and a variable number of threads.

Figure 3.1 shows the data collected with each line representing a different block size. As we can observe, very small block sizes lead to much worse performance rates, as the algorithm needs to call the underlying hash function many more times, with the subsequent increase in resources needed to process the input. As we can observe, a configuration with small block sizes performs better with a reduced amount of threads (or even no threads at all), from what we can guess that the overhead introduced by the use of multiple threads is higher than the resources needed to process such small blocks.

As we increase the block size, the performance in cycles per byte tends to be similar, and therefore we provide figure 3.2 to have a better glimpse on the differences. There we can observe that 10 KiB blocks are still too small to provide the best performance. If we increase the block size too much (up to 1 MiB in this case), we see also a degradation of the performance, while the 100 KiB block size performs best in this test. Subsequent tests showed that the optimal block size in this platform was around **52 KiB**, which is coherent with our assessment to have a block size multiple of the page size used by the system (in this case, 13 pages).

An interesting observation that we can make out of figure 3.2 is that the best performance overall happens when using only 4 worker threads, matching the number

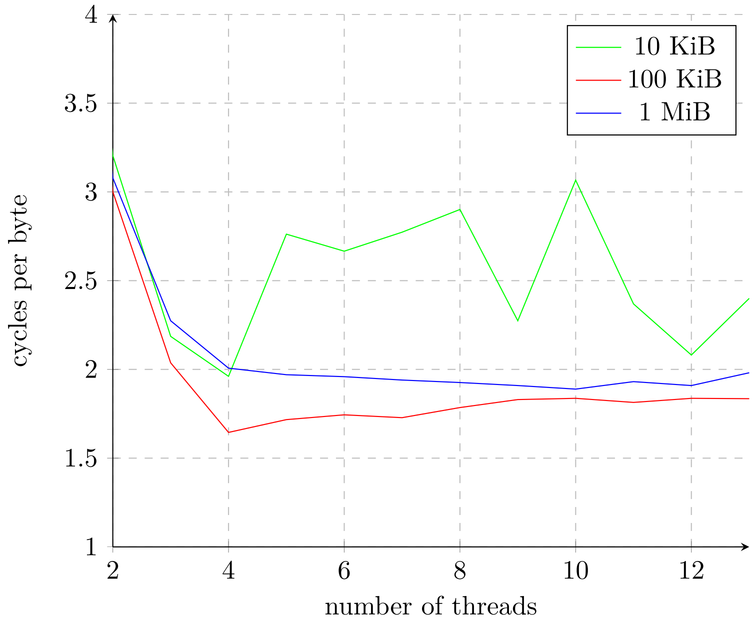

Figure 3.2: CPU cycles taken by iSHAKE 128 to process each input byte for 1024 blocks of variable size and a variable number of threads. Detail of the best performing block sizes.

of cores available in the machine. While it could make sense to think that the best performance would happen when using as many threads as CPU threads are available (8 in this case), further tests performed in *bigmem* confirmed this observation too. Our intuitive explanation for this behaviour is that even if we have more CPU threads available, cache misses impose a degradation in performance as every two CPU threads share the level 2 cache (which is only 256 KiB per core in this machine, meaning it is relatively simple to fill it up with data so that 100 KiB blocks are evicted before being processed, and the CPU needs to fetch them from the level 3 cache or even from system memory). In that case, a number of worker threads that equals the number of cores instead of the number of CPU threads, means that we can optimise the level 2 cache so that no worker threads compete for it during the hash operation, obtaining the best performance observed in our measurements.

For the second part of this experiment, we fix the size of the input data to 1 GiB approximately (by adjusting the total number of blocks depending on the block size), and vary the block size and number of threads. For each combination of both, we gather the corresponding measurements and keep the best figures obtained, as we did previously.

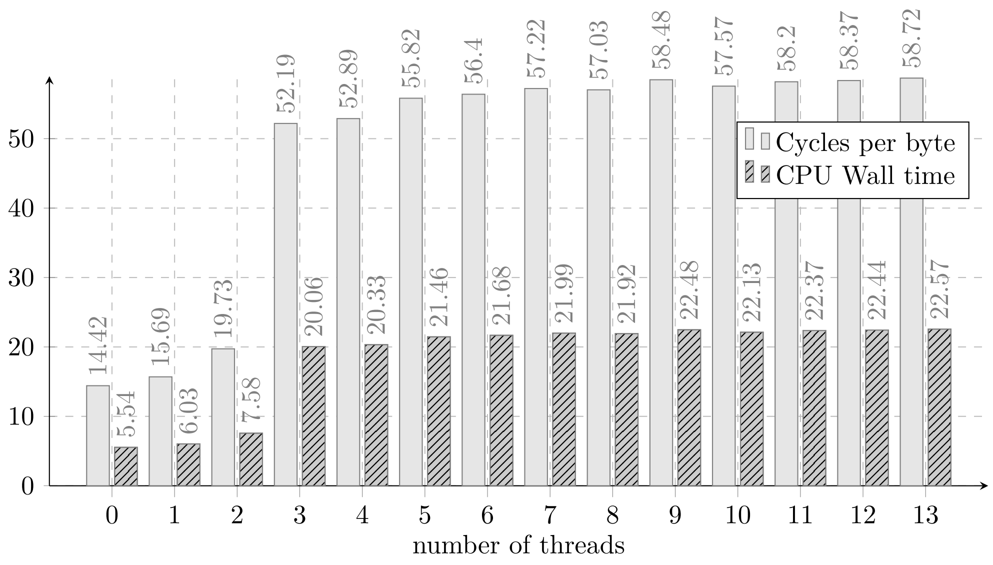

Figure 3.3: Cycles per byte and CPU wall time when hashing 1GiB of data split in blocks of 500 bytes.

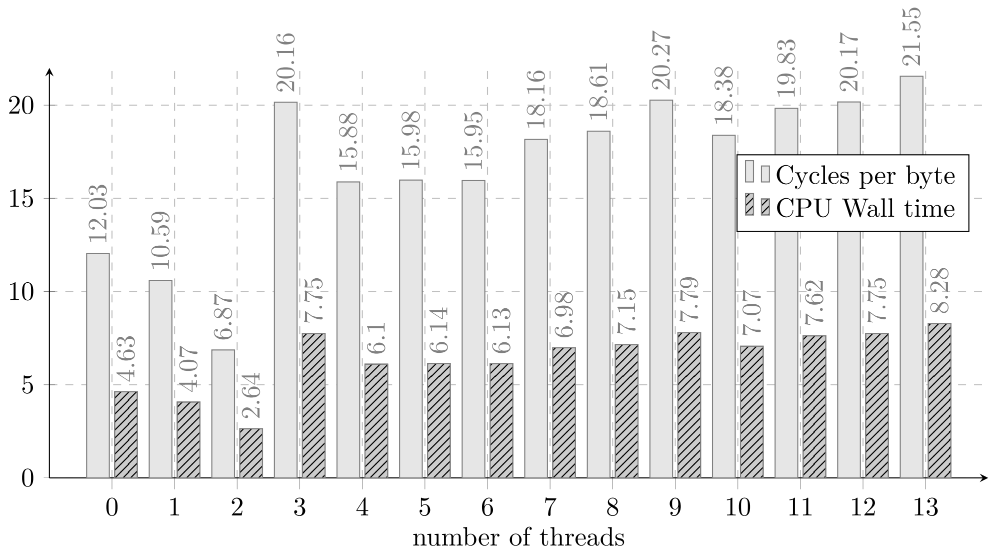

Figure 3.4: Cycles per byte and CPU wall time when hashing 1GiB of data split in blocks of 1 KiB.

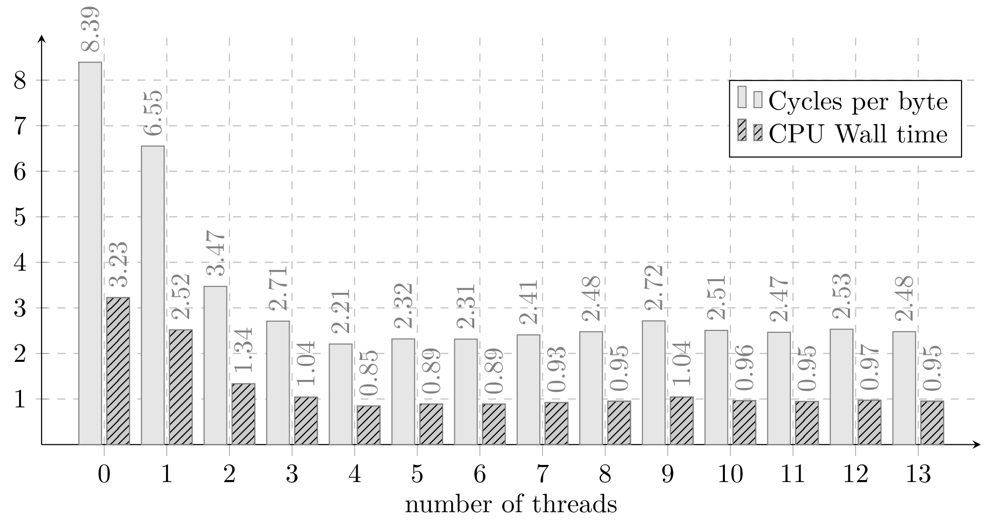

Figure 3.5: Cycles per byte and CPU wall time when hashing 1GiB of data split in blocks of 10 KiB.

When using small block sizes such as 500 bytes and 1 KiB, we observe again that both the wall time and the cycles per byte increase with the number of threads used, while the optimal configuration seems to be between 0 and 2 threads, depending on the case. Both are depicted in figures 3.3 and 3.4, respectively. In any case, the test confirms that no matter the size of the input, using very small block sizes is counter-productive and degrades the performance of the algorithm significantly.

Moving on to the next block sizes (10 KiB, 100 KiB and 1 MiB), figures 3.5, 3.6 and 3.7 show the performance of iSHAKE 128 with each of them. The tests yield very similar results, and surprisingly, this time the number of threads that offer the best performance is always around 8, matching the number of CPU threads available. However, the results are very similar for every number of threads above or equal to the number of cores, which could suggest that, in fact, cache misses are a problem important enough to minimize the performance gain that we could get by using more threads, while the increase in the size of the input makes the overhead of using more threads less important, obtaining an overall improvement. In any case, we can observe a clear correspondence between the number of cycles per byte used by the algorithm to process the input and the wall time elapsed, and this second test confirms our initial results about medium-sized blocks offering the best performance.

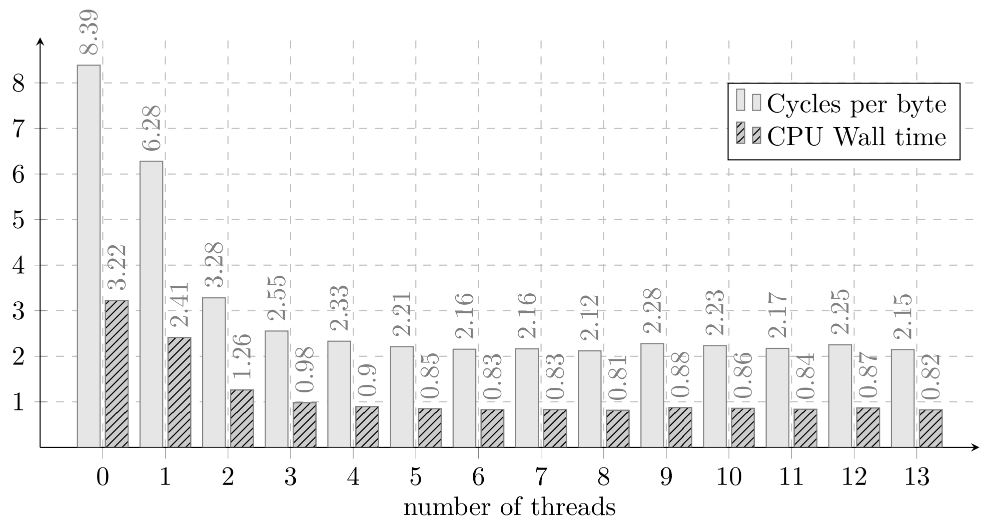

Figure 3.6: Cycles per byte and CPU wall time when hashing 1GiB of data split in blocks of 100 KiB.

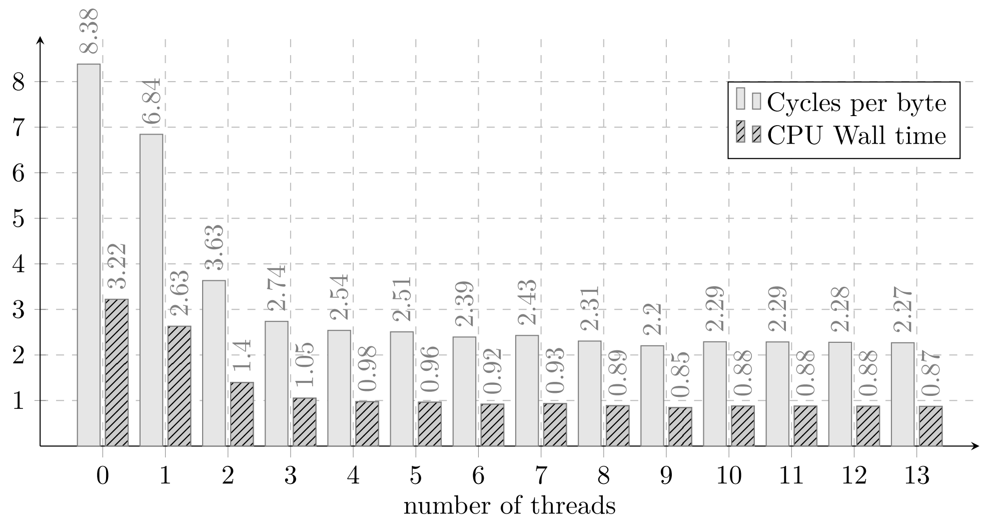

Figure 3.7: Cycles per byte and CPU wall time when hashing 1GiB of data split in blocks of 1 MiB.

3.4 Performance comparison

In our second experiment, we compare iSHAKE with other algorithms, both regular sequential algorithms, as well as others designed to exploit parallelism to speed up the processing. In this case, we measure the time taken by the algorithms compared to hash 1 GiB of data generated with the `dd` utility as in the previous experiment, in order to minimize the bottlenecks.

For each algorithm tested we take 50 samples and compute the main statistical properties of the data set obtained, allowing us to compare them in a meaningful way. But before presenting our results, let us enumerate the algorithms chosen and explain why they were picked for this comparison.

3.4.1 Algorithms

One of the goals of the work presented here is to demonstrate the feasibility of iSHAKE as a widely used hashing algorithm, as well as its superiority in terms of performance when we need to process large inputs. For this reason, we have chosen several algorithms that are either heavily widespread or that incorporate the idea of parallel processing on their design. First we present the list of algorithms compared.

MD5

Message Digest 5 (MD5) [Riv92] was designed by Ronald Rivest in 1992 to be used as a cryptographic hash function. Although multiple issues have been published and the algorithm is considered broken for any security-related uses[6], it is still widely used for its speed [Sig15]. Even though its use is in decline nowadays, MD5 is still taken as a reference in terms of performance, and many modern algorithms set its speed as the goal to reach. For this reason, we consider MD5 is an algorithm relevant enough in our comparison, and we include it in our experiments here.

[6]MD5 has a long trail of security issues and has been subject to extensive cryptanalysis. Den Boer and Bosselaers were able to find two different *initialization vectors* in 1993 that produced an identical digest, causing a pseudo-collision in the compression function. Later, in 1996, Dobbertin was able to confirm a collision in the compression function.

The main problem with MD5 is the length of the resulting digest, being only 128 bits, making it affordable in computational terms to perform a *birthday attack*. In order to evaluate this weakness, the *MD5CRK* project was started in March 2004, and it took barely five months to succeed [BCH06] [HPR04]). Further cryptanalysis was published a year later, in March 2005, demonstrating the possibility to create two X.509 certificates with different public keys and the same MD5 digest [LWdW05]. Just a few days later, Vlastimil Klima improved the algorithm enough to be able to find MD5 collisions in just a few hours with a simple laptop [Kli05].

Years later, in 2010, the first single-block collision ever published was announced [XF10] and later enhanced [Ste12]. Eventually, research demonstrated that it is possible to find collisions in MD5 in less than a second using any modern computer [XLF13]. All these issues forced the approval of an informational RFC to add security considerations to the use of MD5 [CT11].

SHA-1

Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash algorithm designed by the National Security Agency of the United States of America (NSA) and standardised by NIST in a Federal Information Processing Standard [Nat95]. SHA-1 produces 160 bits of output, usually encoded in 40 hexadecimal digits. Due to the *birthday paradox*[7], brute force attacks should be able to find collisions in 2^{80} evaluations, a number that has been unfeasible so far for current technology. However, cryptanalysts have found several weaknesses in SHA-1, downgrading its security to around 2^{63} operations, and eventually demonstrating a full collision of the algorithm[8].

SHA-1 is heavily used worldwide, and plenty of other algorithms and protocols still rely on it not only as a regular hash function but also with security purposes. Since the first real-world collision was published in early 2017 [SBK+17] after its deprecation by NIST in 2011 and later disallowance after 2013 [BR11], many of those still using it have finally started to migrate to other hash algorithms for their use cases, where iSHAKE could demonstrate valuable. It is, therefore, our intention by including SHA-1 in our comparison to show the suitability of iSHAKE in those cases. We will get back to this later on and assess the possibility of using iSHAKE instead of SHA-1 for several use cases in Chapter 4.

BLAKE2

The BLAKE2 cryptographic hash function [AS15] is an improved version of BLAKE, the original function designed by Aumasson et al. for the NIST SHA-3 hash function competition. BLAKE made it to the final round with other four algorithms, but the competition ended up with KECCAK being selected as the new SHA-3 algorithm and standardised in [Nat15].

Based on Dan Bernstein's ChaCha stream cipher [Ber08], it provides two flavours: *BLAKE2b* optimised for 64-bit platforms, and *BLAKE2s* optimised for 8 to 32-bit platforms. It also includes parallel variants for both, called *BLAKE2bp* and

[7]The birthday paradox, in probability theory, refers to the probability to find two persons with the same birthday in a set of n randomly chosen people. In cryptography, this probabilistic model is used to downgrade the security of a hash function with n bits output, so that a collision can be found in $2^{n/2}$ evaluations instead of the 2^n required by a standard brute force attack.

[8]Initial cryptanalysis on SHA-1 focused on versions of the algorithm with reduced rounds [RO05]. Later studies were able to reduce the time required to find a collision in the algorithm from 2^{80} to 2^{69} [WYY05] and 2^{63} [Coc07] attempts. In 2015, an attack named *the SHAppening* was published, allowing a freestart collision attack on SHA-1's compression function requiring only 2^{51} operations [SKP15]. However, it wasn't until February 2017 that Google Inc. announced the first public collision ever found in SHA-1 [SBK+17].

Even though collisions are still expensive with current hardware, experts have been warning about the use of SHA-1 as a cryptographic hash function for years. The National Institute of Standards and Technology issued a special publication [BR11] back in 2011 deprecating the algorithm until the end of 2013, and disallowing its use starting in 2014.

BLAKE2sp, respectively, that exploit multiple cores or Single Instruction, Multiple Data CPUs to improve the performance. Finally, the *BLAKE2x* variants can produce digests of arbitrary length, equivalent to the SHAKE Extendable-Output Function (XOF) defined by the SHA-3 standard [Nat15].

For all these reasons, and given the claim of the authors that BLAKE2 can be as fast as MD5, it feels natural to compare the performance of iSHAKE against it, especially given the attention that BLAKE2 has been receiving since its conception.

KangarooTwelve

KangarooTwelve [BDP+16] is an Extendable-Output Function based on the KECCAK-f[1600] permutation reduced to 12 rounds, as opposed to the original 24 rounds featured by the SHA-3 cryptographic hash function and the SHAKE XOF. This reduction has been deemed secure by the authors given that the best published collision attacks have been able to break KECCAK only up to 6 rounds [DDS13, DDS14, QSLG17, SLG17].

The algorithm is a direct competitor of BLAKE2, featuring arbitrary output length, high performance (thanks to the use of multiple cores and SIMD instruction sets of modern CPUs to provide a high degree of parallelism), tree hashing, final node growing and SAKURA encoding [BDPVA14]. As such, it is also an interesting competitor of iSHAKE, especially given that both use the same underlying KECCAK-p permutation.

ParallelHash

ParallelHash is one of the four SHA-3 derived functions standardised by NIST in [KCP16]. As the previously mentioned BLAKE2 and KangarooTwelve, it can be used as an XOF to produce arbitrary-length outputs, and it was designed to exploit the parallelism available in modern processors to enhance the performance when hashing very large inputs. We consider it, therefore, an interesting competitor of iSHAKE and include it in our experiments to be able to compare its performance.

3.4.2 Results

In this experiment, we measured the wall time and cycles per byte taken by different algorithms, including iSHAKE with different configurations and optimisations, to process 1 GiB of data. From all the samples taken, we took the best values observed for each of them in order to compare their performance.

Figure 3.8 shows the wall time taken by some of the algorithms compared. In this case, we test iSHAKE in the best configuration observed so far, with 52 KiB

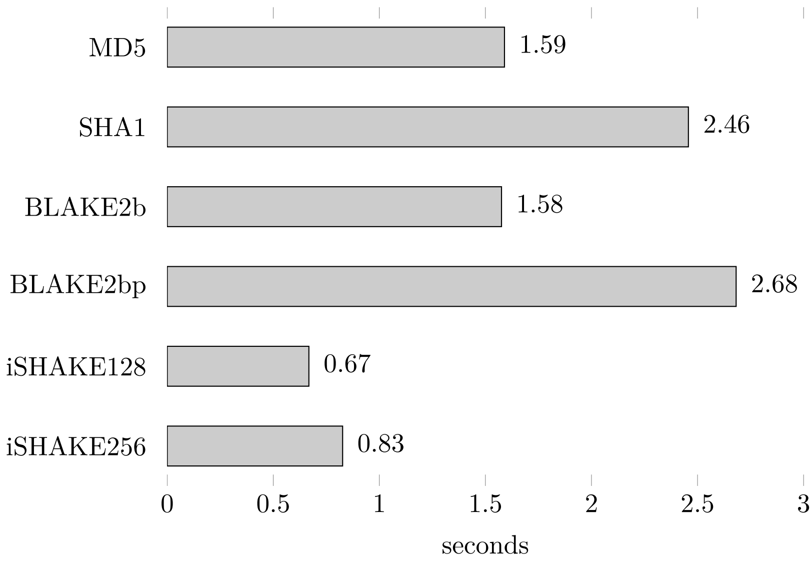

Figure 3.8: Minimum wall time taken by different algorithms to process 1 GiB of data.

blocks and 8 worker threads in the *nepenthe* machine. As we can see, both flavours of iSHAKE, offering 128 and 256-bit security equivalence, outperform all the other algorithms tested by a big margin. It uses around 42% of the time taken by the best competitor, BLAKE2b which is the best performer among the rest of the algorithms, just ahead of MD5, confirming the claim of the authors that it can be as fast as the latter. SHA-1 is slightly worse than these two, taking almost an additional second to process the data. The worst algorithm in this comparison is BLAKE2bp, which surprisingly takes even more time than SHA-1 to process the input. This is particularly puzzling given that this variant of BLAKE2 is supposed to take advantage of parallelism in order to speed up the computations, but it takes more than a second more than the regular version of the algorithm.

In the second figure, 3.9, we show the best *cycles per byte* measurements obtained in the *nepenthe* machine for a set of algorithms that take advantage of parallelism, including iSHAKE 128. In this case, we measured iSHAKE with two different configurations: one without any parallelism at all, just using sequential processing; the other, using iSHAKE with seven worker threads. The latter is by far the best performing algorithm, with as low as 1.65 cycles per byte of input, ahead of KangarooTwelve, which is the second best performing algorithm in this test. This is particularly relevant, as both iSHAKE and KangarooTwelve use the same underlying Keccak-*p* permutation. However, while iSHAKE uses the 24 rounds of the permutation defined in SHA-3, KangarooTwelve reduced this number to 12 rounds

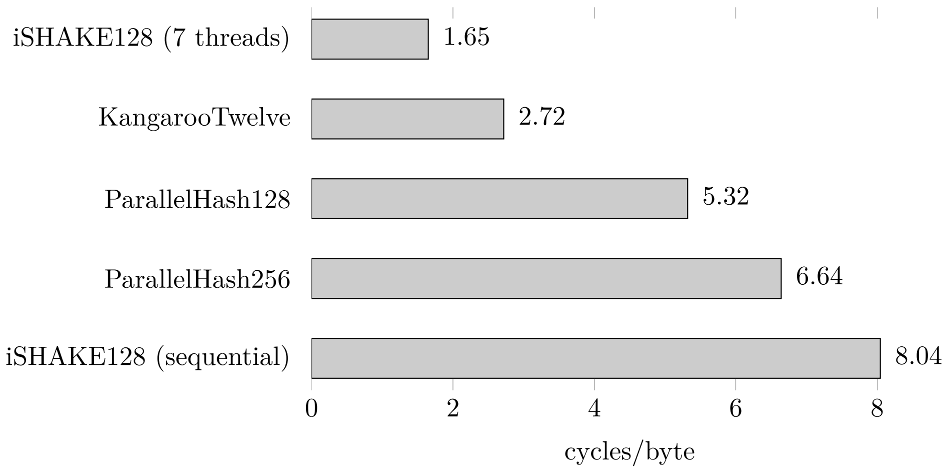

Figure 3.9: Minimum CPU cycles per byte observed for different algorithms and configurations.

instead, and should, therefore, be much more efficient. This result demonstrates the power of iSHAKE leveraging the parallel processing of blocks in multiple cores, as it is able to outperform an algorithm using half of the permutation rounds and one single, sequential hash execution, as opposed to one hash function execution with 24 rounds per each block in our case.

ParallelHash, on the other hand, offers surprisingly low performance in this test, in both the 128 and 256-bit security equivalent versions. We believe that an implementation of this algorithm that uses multiple cores at the same time, instead of just SIMD instructions to achieve parallelism, could offer much better figures than the ones shown here. To the best of our knowledge, there are no such implementations at the moment of this writing, though.

Finally, we compile a set of measurements for all the algorithms compared in figure 3.8 (see Appendix C for the full tables with all samples) and compute the most relevant statistical properties for each set, like the average, standard deviation, quartiles and 90th percentile. Table 3.1 shows the statistics for the values measured in the *nepenthe* machine, while table 3.2 presents the same information for the values obtained in *bigmem*.

	MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (8 threads, generic optimisations)	**iSHAKE 256** (8 threads, generic optimisations)	**iSHAKE 128** (8 threads, lane complementing)	**iSHAKE 256** (8 threads, lane complementing)
average	$1,6101$	$2,5079$	$1,5956$	$2,8454$	$0,7027$	$0,9299$	$0,9012$	$0,9938$
std dev	$0,0143$	$0,0422$	$0,0108$	$0,0543$	$0,0243$	$0,0541$	$0,1003$	$0,0785$
min	$1,59$	$2,457$	$1,576$	$2,682$	$0,668$	$0,827$	$0,694$	$0,854$
max	$1,657$	$2,612$	$1,633$	$3,136$	$0,776$	$1,026$	$1,144$	$1,184$
median	$1,608$	$2,4885$	$1,595$	$2,8355$	$0,6985$	$0,937$	$0,9075$	$0,995$
90-perc.	$1,627$	$2,5674$	$1,6081$	$2,8688$	$0,7305$	$0,9991$	$1,0126$	$1,1065$
1-quart.	$1,5992$	$2,4725$	$1,5892$	$2,83$	$0,6857$	$0,9007$	$0,8515$	$0,9322$
3-quart.	$1,616$	$2,5397$	$1,601$	$2,8467$	$0,716$	$0,9652$	$0,947$	$1,0387$

Table 3.1: Statistics for some algorithms hashing 1 GiB of data, *nepenthe*.

	MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (8 threads, generic optimisations)	**iSHAKE 256** (8 threads, generic optimisations)	**iSHAKE 128** (8 threads, lane complementing)	**iSHAKE 256** (8 threads, lane complementing)
average	$2,777$	$4,2162$	$2,5098$	$3,6034$	$1,1052$	$1,3394$	$1,0378$	$1,2652$
std dev	$0,1681$	$0,2901$	$0,1654$	$0,4543$	$0,2108$	$0,1848$	$0,2078$	$0,2088$
min	$2,45$	$3,79$	$2,22$	$2,29$	$0,79$	$1,02$	$0,72$	$0,88$
max	$2,99$	$5,47$	$2,78$	$4,4$	$1,53$	$1,78$	$1,41$	$1,65$
median	$2,785$	$4,185$	$2,55$	$3,725$	$1,035$	$1,295$	$0,98$	$1,29$
90-perc.	$2,98$	$4,361$	$2,68$	$4,142$	$1,401$	$1,563$	$1,321$	$1,523$
1-quart.	$2,75$	$4,105$	$2,52$	$3,2025$	$0,93$	$1,1925$	$0,8825$	$1,055$
3-quart.	$2,91$	$4,34$	$2,64$	$3,905$	$1,29$	$1,4875$	$1,21$	$1,4275$

Table 3.2: Statistics for some algorithms hashing 1 GiB of data, *bigmem*.

As previously observed, *nepenthe* yields better performance measurements than *bigmem*. However, there is a significant difference between the two of them in terms of the optimisations used. The Keccak Code Package provides two possible optimisations to build the library on 64-bit platforms: one generic set of optimisations, and another using those optimisations plus a feature called *lane complementing* described in the Keccak implementation overview [BDP+12]. If we observe the results in table 3.2, iSHAKE offers better performance when using the lane complementing transform. However, this feature seems to be counter-productive under certain circumstances, as we can see in table 3.1 with the measurements in *nepenthe*. Further investigation is needed to properly assess the origin of this issue, although the lack of some SIMD instructions in the mobile processor used by the latter is likely to be the root of this performance penalty.

Chapter 4 Use cases

When a new system or algorithm is designed, it is done with the intention to resolve a problem or to obtain a benefit respect to the starting point. The process starts with the general description of a use case where the new algorithm or system could help, but that needs to be translated into practical, real-world examples that can actually be implemented for people to benefit from our work.

In this chapter, we evaluate some of those possible real-world scenarios where iSHAKE could be used to improve the current situation and assess how it could technically fit in the existing systems and software.

4.1 The Git Version Control System

Git is an open source Version Control System (VCS) used to keep track of changes in documents, and its most typical use is to manage software projects. Git was created in 2005 by Linus Torvalds (author of the Linux kernel) together with other kernel developers, precisely for the development of the Linux project itself. Therefore, *speed* and *integrity* of the data were some of the key design principles in mind when Git was conceived.

One of the main innovations made by Git was the introduction of digests as identifiers for commits, instead of using traditional, sequential revision numbers [Sin11]. This way of working introduces some challenges, but mostly adds the flexibility needed to manage very complex and big projects. In general, everything in Git is an object:

– *Blobs* (Binary Large Objects) hold the contents of files, although they include no metadata about them (like file names or timestamps of any kind). Figure 4.1 represents the structure and process to build a blob object.

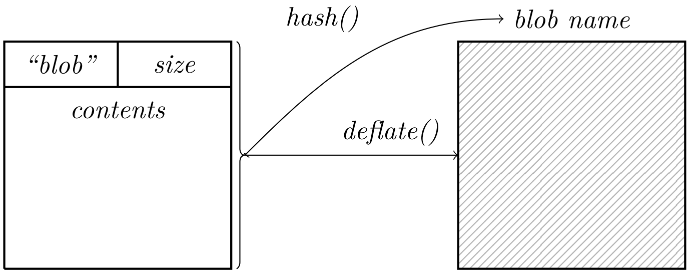

Figure 4.1: Git blob object. The contents of a file are prepended the literal word "*blob*" followed by the size of the contents. The resulting object is passed through a hash function to obtain the name of the blob, and fed afterwards to a *deflate* function to compress it and generate the final object.

- *Trees* contain lists of one or more file names, with each of them being either blobs or other trees. They act as directory entries on a file system, and in general constitute a Merkle tree containing all trees and blobs that are part of the repository being tracked. The top tree object in the structure reflects the current status of all files and directories at a given point in time.

- *Commits* are the way Git builds a history of changes in a repository. They contain a pointer to the top level tree object, additional metadata describing the change (for example who did it, a message explaining the commit or a timestamp), and a link to any parent commit or commits.

- *Tags* are a way to attach additional metadata to any object. They are typically used to specify version numbers or to digitally sign a specific commit.

In Git, all objects have names, and their names are the hexadecimal-encoded 40-byte hash result of applying the SHA-1 algorithm to the contents of the object. Therefore, any given change in a repository (by means of a commit) is then identified by a SHA-1 hash, the same way objects are also identified by their corresponding hashes.

4.1.1 Challenges

Given the use of hashes made by Git to identify objects, collisions are a critical problem in the system. Hashes need to be unique, no matter how many objects there are in the system, and if two objects generate the same hash, a collision occurs and the object database can get corrupted. Even worse, a third-party capable of mounting a second preimage attack on SHA-1 would be able to rewrite the history kept by Git without nobody noticing.

Due to the recent developments made to break SHA-1 [SBK+17], the Git community has been discussing a plan to change the underlying hash algorithm used to identify objects[1]. While SHA3 with 256-bit security was the first algorithm chosen in the discussion, the algorithm which will be finally used is still undecided and arguments have been made in favour of SHAKE, KangarooTwelve of BLAKE2[2]. A document has been collaboratively written to draft a transition plan to support other hash functions and phase out SHA-1, describing the requirements and needs for the new algorithm (or algorithms)[3].

4.1.2 Using iSHAKE

There are two different scenarios in Git where iSHAKE could be proven useful:

– Repositories with very large amounts of objects (either representing files, meaning lots of *blob* and *tree* objects, or a very long history, meaning many *commit* objects.).

– Repositories with very large files.

Both cases are not as uncommon as one might think. A good example is the Git repository where Microsoft keeps the source code of its Windows operating system, which has been recently in the news due to its very large size (300 GiB)[4]. In general, well-established projects that have been around for many years and are well maintained are prone to a huge history, originating enormous amounts of objects. In some cases, when Git is used not only to manage source code but also binary contents[5], the size of the files managed can also be a challenge, degrading the performance.

Switching Git to use iSHAKE might bring interesting advantages to tackle the particular issues that arise in such extreme scenarios while keeping it simple enough so that common users of the system do not experience any performance degradation or side effects.

[1]See this email thread for a complete discussion on the topic: https://public-inbox.org/git/20170304011251.GA26789@aiede.mtv.corp.google.com/.

[2]See https://public-inbox.org/git/91a34c5b-7844-3db2-cf29-411df5bcf886@noekeon.org/ for an interesting discussion initiated by the KECCAK team, suggesting alternatives to SHA3 also based on KECCAK.

[3]See https://goo.gl/gh2Mzc.

[4]See https://arstechnica.com/information-technology/2017/02/microsoft-hosts-the-windows-source-in-a-monstrous-300gb-git-repository/ for more details on how Microsoft handles this repository and how they had to extend Git to support such an extreme use case.

[5]Backup tools have been developed on top of Git, such as *bup* (https://bup.github.io/). In a use case like this, an entire disk might end up being stored in a Git repository, significantly increasing its size and posing a challenge for Git itself.

Efficient processing of large files

Given the performance advantage of iSHAKE over other hash algorithms that we have seen in Chapter 3, iSHAKE could contribute significantly to improve the performance of Git when handling very big files. There are two different ways Git could benefit from the use of iSHAKE in this particular regard:

- When adding **new** large files. In that case, the entire file needs to be processed (plus a header prepended by Git for every blob object), and the parallelism offered by iSHAKE can significantly reduce the processing time for the file as we have seen, provided that the machine where the file is processed has multiple cores allowing parallel computation.
- When modifying **existing** large files. In this case, using iSHAKE allows Git to modify an existing hash (if the file already existed, it had a *name* corresponding to its hash) in constant time, depending only on the number of blocks involved in the modification. This is possible due to the order in which Git processes blobs, computing their corresponding hash first and then deflating the contents using *zlib*[6] [7]. This way, the hash operation is performed before the changes are propagated to the entire blob due to the *avalanche effect* in the compression function. Had it been the other way around (compress first, then compute the hash) as it was in the beginning of Git, such optimisation would not have been possible.

Efficient handling of changes

In Git, both *trees* and *commits* include in their contents pointers to other Git objects, those being either blob objects or other trees. These pointers consist in the name of the blobs, with those names being the result of passing the contents through a hash algorithm.

When a change happens in a blob or a tree object, the tree objects involved need to be hashed again. This can also be costly in terms of performance if the objects are big enough. However, we can propose a change to make changes much more efficient by avoiding passing the affected objects through the hash function.

Consider a tree where a single file has been changed, and therefore so has its corresponding blob name. This means we need to hash the tree object again to reflect this change and repeat the operation if the tree object has any parents. Let

[6] Zlib is a popular compression library. See https://zlib.net/.

[7] Git's user manual specifies this order of operations for recent versions of Git, while the original version compressed the blobs before computing the SHA-1 hash. See http://schacon.github.io/git/user-manual.html#object-details for more information.

us imagine the contents of a tree object as iSHAKE blocks instead, where the first block contains the basic header identifying the object as a tree, the *size* field counting the number of other objects included, and the ordered list of objects by type and name in the file system (the names of the files or the directories). The content of the tree object, right after this initial header and list, is the list of iSHAKE hashes corresponding, in the same order, to each referenced object.

With this structure, Git can efficiently handle changes by leveraging iSHAKE's ability to update blocks. When a blob object changes its contents and therefore its corresponding hash (its name), the tree object that references it only needs to update the corresponding iSHAKE block to recompute its own hash:

1. Hash the changed object using iSHAKE .
2. Inflate (decompress) the tree object that references the changed object.
3. Take the name of the tree object (its hash) and use the commutative operation to subtract the old name of the changed object from it.
4. Finally, use the commutative operation again to add the hash of the updated object (its new name).

Note that, in this case, we are not appending headers to the iSHAKE blocks. This is like that to allow us directly reuse final iSHAKE hashes corresponding to other objects without recomputing them, and it is possible because order between these iSHAKE "blocks" (Git objects) is irrelevant. The order is kept by listing the file names and the object names in the same order inside of the tree object, but the relative order between objects has no meaning whatsoever. The structure of these new tree objects and the process to recompute their hashes is illustrated by figure 4.2.

Better resistance to collisions

One of the downsides of iSHAKE, as we have previously seen, is the length of the resulting hash (a minimum of 2688 bits for the 128-bit security equivalent) [MGS15, Wag02]. This would have an impact in Git in the sense that the space needed to store the hashes of the objects would significantly increase (almost seventeen times in the best-case scenario). However, this space can be shortened by using the binary hashes instead of their hexadecimal representation.

While this can be a serious pitfall, the use of iSHAKE could deliver some advantages in Git regarding collision resistance. Since iSHAKE uses the KECCAK sponge function, which has proven very resistant to collisions and cryptanalysis so

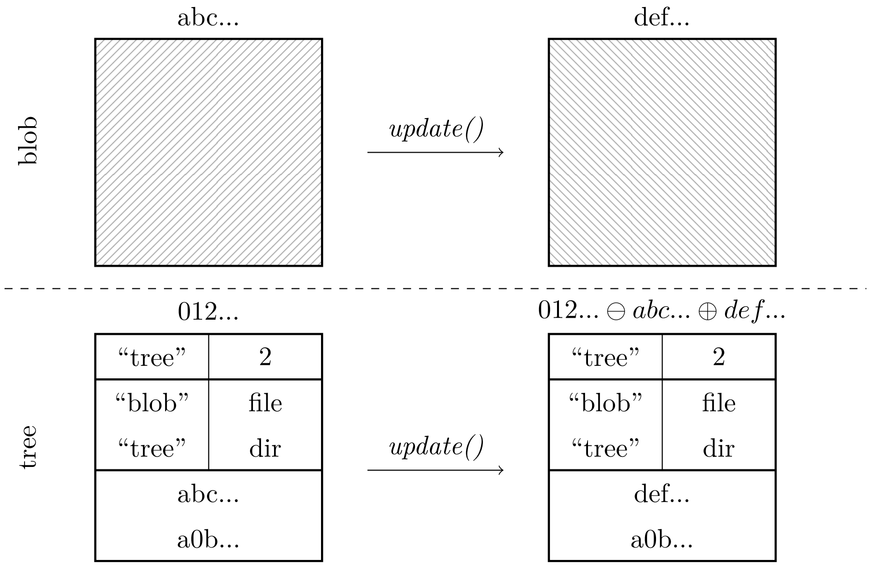

Figure 4.2: Recomputing Git tree objects. First, we update the affected blob object by computing the iSHAKE hash of the new version. Secondly, we recompute the iSHAKE hash of the tree pointing to the updated object by subtracting the old hash of the blob and adding the new one just obtained.

far, Git could benefit from that to minimize the risk of a collision even in the biggest repositories conceivable. Additionally, given that iSHAKE uses itself an extendable output function, Git can benefit from that in case that the minimum hash length proves insufficient in the future to increase the length of the hashes without breaking any existing repositories.

4.2 BitTorrent

BitTorrent is a *peer-to-peer* (P2P) protocol created by Bram Cohen in 2001, intended to allow the efficient exchange of large amounts of data through a network where all nodes (*peers*) collaborate together in the distribution. The BitTorrent protocol is standardised and extended by means of documents called BEPs (BitTorrent Enhancement Proposals), and the main document defining the protocol is BEP 3 [Coh08].

The file or files distributed with the BitTorrent protocol are split into blocks called *pieces*. In order to bootstrap the distribution, a *torrent* file or *metainfo* file is created containing (among other things) the concatenated SHA-1 hashes of every piece in order. That way, when a BitTorrent client completes the download of a piece from another peer, it can verify it by computing its SHA-1 hash and comparing that to the corresponding hash stored in the metainfo file.

4.2.1 Challenges

The BitTorrent protocol faces several challenges emerging from its design and the hash function chosen to provide integrity, SHA-1, especially after a collision was found on it [SBK+17]. This has an important consequence for the protocol in terms of integrity, as an attacker able to mount a second preimage attack could distribute a tampered block to other peers without them noticing. While this sounds still as a remote possibility, advanced cryptanalysis may be able to exploit the weaknesses of SHA-1 further, up to the point where such an attack is feasible. For this reason, the BitTorrent community has already started a discussion to transition away from SHA-1 to another cryptographic hash function[8].

Performance is perhaps the most recurrent issue for hash functions, and BitTorrent is not an exception, given that the protocol is intended to enable the exchange of very large quantities of data. That data needs to be split into pieces and the SHA-1 hash needs to be computed for each of them. This can be quite resource-consuming in older hardware or when the input data is indeed very large. It is important to note that all peers in the distributed network suffer from this problem, as the pieces must be hashed both when the torrent file is created and when they are received from another peer (called a *seeder* in BitTorrent's jargon).

Additionally, the size of the torrent or *metainfo* file can also be an issue. Since it stores the SHA-1 hashes for each piece (20 bytes per raw, binary hash), its size depends directly on the number of pieces. Bigger torrent files are more difficult to distribute, and they can suppose an overload for the machines that store and serve them. It is therefore common practice to increase the size of the pieces considerably to reduce the size of the torrent file. However, that leads to fewer pieces, and clients of the protocol take longer to obtain them and to be able to serve them themselves, reducing the efficiency of the distributed architecture. A BEP document is available with a proposal to use Merkle trees to compute the hashes and store only the root of the tree in the torrent file [Bak09]. Unfortunately, without a change in the hash algorithm used, the suggested model does not solve the integrity issue as a malicious peer could serve crafted *uncle* or *sibling* hashes in order to ensure that the modified piece can be verified against the root of the tree. Refer to [Bak09] for more information on how such extension to BitTorrent would work in practice.

[8]An interesting discussion has been taking place during the last few months, both on what hash algorithm to use, and even how to change the design to optimise the protocol using Merkle trees. See https://github.com/bittorrent/bittorrent.org/issues/58.

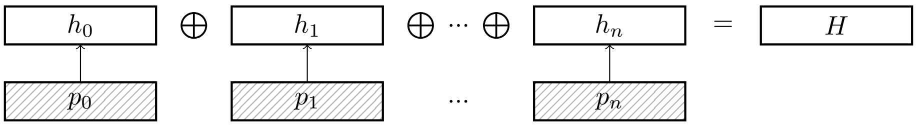

Figure 4.3: Using iSHAKE in BitTorrent. Pieces are hashed using iSHAKE (meaning they can be divided into smaller blocks for parallel processing) and the resulting hashes joined with the commutative operation. Note that the indices used in the headers of the iSHAKE blocks must continue increasing between pieces to keep their relative order.

4.2.2 Using iSHAKE

The main advantage of using iSHAKE in BitTorrent could be, unsurprisingly, a performance enhancement. Since the protocol already splits the input into equal-sized blocks called *pieces*, it is intuitive to apply iSHAKE to compute the hashes for them as if they were blocks of the function. There is even a further optimisation that we can carry out, in the case where the pieces are so big that the computation of their respective hashes can also be optimised.

We propose therefore a *two-tier* architecture where the pieces are split into iSHAKE blocks themselves to allow the optimisation of larger piece sizes. The resulting iSHAKE hashes are subsequently joined with the commutative, reversible operation to obtain a final digest H that is included in the torrent file (instead of the *pieces* key containing the hashes of every single piece [Coh08]), as depicted in figure 4.3.

With this approach, we can obtain better performance when hashing individual pieces, as well as the entire input, thanks to the parallelism offered by iSHAKE. Additionally, we reduce the size of the torrent file as it now contains a fixed-size hash (with a variable length depending on the iSHAKE variant and chosen hash length), facilitating its distribution over the web. Both features can offer a dramatic improvement to the protocol.

There is one missing aspect, though. Peers need to be able to verify the integrity of a piece that has just been downloaded. However, its corresponding hash is no longer part of the torrent file. We need a mechanism similar to the one described in [Bak09], so that a *seeder* (the BitTorrent peer that servers a given piece) provides not only the contents of the piece but also the *delta* δ between that piece and the iSHAKE hash H stored in the torrent file. This *delta* is the iSHAKE hash resulting of *removing* the piece from the final hash H ($\delta = H \ominus h$), so that the peer receiving the piece can compute the hash of the piece h and add it to δ ($x = \delta \oplus h$). If the result equals to H, then the piece is verified.

While this allows us to verify the integrity of a piece, it offers no security at all against tampered pieces distributed by a malicious peer, since that rogue peer can modify the piece at will, compute its corresponding hash h', and give other peers another delta δ' result of removing the modified piece from H: $\delta' = H \ominus h'$. This δ' will, of course, allow the verification of the tampered piece, and even though the modification will be detected upon completion of all pieces (as the result of joining all hashes together will not give H as a result), it is impossible to determine which piece was the offending one due to iSHAKE's very own nature.

In order to add proper integrity to pieces when using iSHAKE, we suggest one final tweak to the protocol that also allows us to simplify it. Let us call the peer that creates a torrent the *initial seeder* or S_0. When creating the torrent file, this peer creates a pair of cryptographic keys, one public and one private, keeping the private key for itself and including the public key in the torrent file. The peer starts then the distribution of the pieces, sending their contents to other peers, as well as their corresponding hashes h_i. Imagine S_0 sends a piece p_i to S_1. Additionally to sending the contents of the piece, S_0 computes the corresponding hash h_i and signs it with its private key, creating the signature sig_i. By sending to S_1 both the h_i hash and its signature $(h_i \| sig_i)$, S_1 is now able to verify the signature of h_i and, if it matches, continue to verify that the the h_i hash matches with the one computed for the piece p_i it just received.

If at any later point in time, another peer S_2 receives the same piece p_i from S_1, it can follow the same procedure to verify the integrity of the file. Since h_i was signed with S_0's private key, which is not available to any other peer S_n, a malicious seeder cannot alter the contents of a piece and distribute it without other peers noticing. This means, of course, that all peers in the network must keep both the hashes for each piece and their corresponding signatures, given that the signatures cannot be recomputed by any peer other than S_0. Eventually, when a peer S_n completes the download of all pieces in a torrent, their hashes can be combined together with the commutative operation and compared to the global hash H stored in the torrent file for an additional integrity verification. This solution allows us to provide strong integrity verifiability as well as authentication, while keeping anonymity[9] and the size of the torrent files constant and relatively small.

We realise that the model proposed here implies a relatively significant change to the BitTorrent protocol, and introduces additional cryptographic operations:

[9]Due to the cryptographic key pair being generated for a given torrent, it is not possible to track it to a specific person, or even track different torrent files created by the same individual. On the other hand, if full authentication is required –including the identity of the creator of a torrent file as well as non-repudiation–, the model suggested here easily allows it by fixing the key pair used by the user and tying it to a known identity.

- The initial seeder needs to compute the digital signature for each piece hash, rather than just the hashes themselves. This is a one-time operation per piece, though.
- All other peers need to perform a digital signature verification for each piece received, although these operations happen scattered in time.

Although these extra operations have their own burden in terms of performance, we believe that our proposal here can provide a noticeable improvement in terms of security and resistance to collisions of the protocol (meaning better protection against malicious peers trying to alter pieces), as well as the overall performance and scalability when piece sizes are chosen carefully, given that the iSHAKE variant and hash length can be adjusted to cope better with any possible issues derived from future cryptanalysis of the algorithm.

Chapter 5

Conclusions and future work

We have presented here our work to enhance the performance of iSHAKE, demonstrating the theoretical advantages laid out in the original design [MGS15]. In our previous, naive development [PC16], we demonstrated the core concepts of iSHAKE and why it is superior when a hash needs to be updated after a change in the input, thanks to the ability that it provides to insert, delete and update blocks. We demonstrated that all those operations can be performed in constant time, as oposed to the linear time taken by traditional sequential algorithms. Even other incremental algorithms based on Merkle trees cannot offer such a high performance, as the amount of hash operations still depends on the size of the input data.

Now we have introduced full parallelism in our implementation, not only at the lowest levels possible, taking advantage of the optimisations provided by the underlying hash function, but also at higher levels where blocks can be processed in parallel by different cores or even machines.

Using this enhanced implementation, we laid out a set of experiments to measure its performance and compare it to other hash algorithms, both sequential and incremental, such as MD5, SHA-1, BLAKE2 or ParallelHash. The results we obtained helped us show the advantages of our model and propose it as a serious contestant. Furthermore, we elaborated several examples where an incremental hash algorithm like this could prove useful, and even assessed how to use it to improve widely-used protocols and tools, like *BitTorrent* and *Git*, respectively.

Our results show that iSHAKE can provide a solid enhancement in terms of performance compared to sequential hash algorithms, and it provides also the foundations for further high-performance processing, taking our model to greater levels of parallelism. However, as any other system, the advantages are usually balanced by drawbacks, and informed decisions need to be taken to adapt the use of the algorithm to each individual use case.

In Chapter 2 we made some suggestions to fine tune iSHAKE to optimise its performance while being able to make a sound judgment on other desired properties like compatibility or portability. Later on, in Chapter 4, we presented a couple of use cases where iSHAKE could be introduced instead of other hash functions, in this case SHA-1. Our proposals there should serve as an illustration of the benefits of using iSHAKE while giving also context enough for the reader to understand the intricacies of applying such an advanced algorithm to existing use cases and considering all possible implications and side effects.

All in all, we believe the present work demonstrates the feasibility of the iSHAKE algorithm as a high-performance hash function when the amount of data to process is significantly large and real-time requirements exist.

5.1 Future work

There is still a lot of work and research that we can do to further improve iSHAKE and take advantage of its model to obtain even better performance. In this section, we discuss two possible approaches that should be considered.

5.1.1 Portability

Our current iSHAKE implementation depends on the availability of the *POSIX threads* API, which is available in most *UNIX-like* operating systems. This makes it more difficult to build and use the library in other operating systems where the API is not present, and therefore we assume the need to provide an implementation that can be built in those operating systems lacking *pthread* support without compromising the parallel-processing capability at the core of iSHAKE.

Work should be carried out to test building the library in multiple operating systems and modifying what's needed to allow its use in the most commonly used platforms.

5.1.2 High-Performance Distributed Computing

One of the two major developments that we can perform to take the most out of iSHAKE's parallel computation model is the use of *High-Performance Distributed Computing* infrastructures to run the algorithm. We have demonstrated the advantage of parallel computation by leveraging the threading capabilities offered by the operating system to significantly reduce the processing time when more than one CPU is available. Such a scenario is relatively common nowadays with the advances in hardware, but the concept itself can be easily extrapolated to other high-performance architectures where a cluster of machines is used to distribute computing tasks.

We have already mentioned before Hadoop's *MapReduce* model [DG04], based on a strategy for data analysis commonly known as *split-apply-combine* [Wic11]. The MapReduce paradigm enables high scalability across an unlimited number of servers in a Hadoop cluster. The process is as follows:

1. First, the *Map* task is executed, taking an input and converting it into smaller pieces with some sort of characterization, order, or any other properties that are interesting for our purposes.

2. Secondly, the *Reduce* task is performed by the worker nodes, taking the output of the *Map* task and processing it according to some kind of logic.

This general model can be applied to boost the performance of iSHAKE by distributing the hash operations across a set of worker nodes. In this case, the *Map* stage consists in the original input being split into blocks, which are then appended their own headers. These blocks are then sent to the cluster for the *Reduce* stage, where nodes apply the underlying hash algorithm to them to obtain a digest for each block. Finally, all digests are joined together using the commutative operation with no particular restriction in terms of their relative order.

Such a paradigm would allow us to improve even more the performance of iSHAKE for those use cases where real-time and high-performance are paramount. Furthermore, the model is generic enough so that it can be applied to a multitude of use cases and specific infrastructures, allowing for all kinds of customised implementations.

5.1.3 Underlying hash function

Due to the very own nature of iSHAKE, the hash function used to process each block of input is interchangeable. The SHAKE extendable output function was chosen to take advantage of its variable-length output and its superior speed compared to regular SHA-3. The fact is that one of the main criticisms that SHA-3 has faced since it was standardised was that it was slow when implemented in software (hardware implementations, on the other hand, are overtly faster than any other competitors) compared to other contestants of the SHA-3 competition[1]. In general, it is believed that the issue here is the extremely conservative security parameters used for the different versions of SHA-3, affecting also the SHAKE variants[2]. The latter, though, is slightly faster thanks to some implementation advantages, but still slower than

[1]See https://bench.cr.yp.to/results-sha3.html for an extensive performance comparison of the SHA-3 competition contestants.

[2]David Wong wrote an interesting article explaining the issue in his blog, available here: https://cryptologie.net/article/393/kangarootwelve/.

other contestants and much slower than the BLAKE2 function, which was not part of the competition.

These allegations against SHA-3 led many people to use the aforementioned BLAKE2 function instead of SHA-3, like *Argon2* [BDK16] (the winner of the *Password Hashing Competition*[3]) or the *libsodium* library[4], and even the KECCAK team issued an statement commenting on the issue[5]. For this reason, they also decided to introduce a new algorithm called *KangarooTwelve* [BDP+16], which is essentially a variant of the KECCAK sponge function with a reduced number of rounds from the original 24 to 12, considering this as more than enough to guarantee the security of the function (given that after all the cryptanalysis and scrutiny that KECCAK has been put through, researchers have been only able to break it up to six rounds) while providing an important boost in speed.

Provided that we trust the claims of the KECCAK team about the security of KangarooTwelve, it is natural for us to consider an implementation of iSHAKE using it instead of the SHAKE extendable output functions. As we have observed in our tests, KangarooTwelve is indeed closer to the speed that iSHAKE can provide with optimal configuration, and therefore a good candidate to speed up our performance even more, combining the fastest KECCAK-derived design with our highly-parallelisable model.

Looking further into the future, our choice to stick to KECCAK or derived functions can provide us an interesting benefit if hardware starts to incorporate logic into their instruction sets to implement the sponge function, as it is expected to happen. At that point, the benefit of parallel computation would be added on top of an underlying hash function that will be extremely fast already thanks to hardware optimisations.

[3]See https://password-hashing.net/.

[4]See https://libsodium.org.

[5]The statement about the speed of SHA-3 in all its variants is available here: http://keccak.noekeon.org/is_sha3_slow.html.

References

[AS15] Jean-Philippe Aumasson and Markku-Juhani Olavi Saarinen. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). Internet Requests for Comments, November 2015.

[Bak09] Arno Bakker. BEP 30: Merkle hash torrent extension, August 2009. Available at http://bittorrent.org/beps/bep_0030.html.

[BCH06] John Black, Martin Cochran, and Trevor Highland. A Study of the MD5 Attacks: Insights and Improvements. In *Fast Software Encryption, 13th International Workshop, FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2006. Available at http://www.iacr.org/cryptodb/archive/2006/FSE/3237/3237.pdf.

[BDK16] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 292–302, March 2016.

[BDP$^+$12] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview, version 3.2. Technical report, May 2012. Available at http://keccak.noekeon.org/Keccak-implementation-3.2.pdf.

[BDP$^+$16] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. KangarooTwelve: fast hashing based on Keccak-p. Cryptology ePrint Archive, Report 2016/770, August 2016. Available at http://eprint.iacr.org/2016/770.

[BDPVA14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Applied Cryptography and Network Security: 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, chapter Sakura: A Flexible Coding for Tree Hashing, pages 217–234. Springer International Publishing, 2014.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers., January 2008. Available at http://cr.yp.to/papers.html#chacha.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental Cryptography: The Case of Hashing and Signing. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 1994.

[BR11] Elaine B. Barker and Allen L. Roginsky. SP 800-131A. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. Technical report, Gaithersburg, MD, United States, January 2011.

[Coc07] Martin Cochran. Notes on the Wang et al. 2^{63} SHA-1 Differential Path. Cryptology ePrint Archive, Report 2007/474, 2007. Available at http://eprint.iacr.org/2007/474.

[Coh08] Bram Cohen. BEP 3: The Bittorrent Protocol Specification, January 2008. Available at http://bittorrent.org/beps/bep_0003.html.

[CT11] Lily Chen and Sean Turner. Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. RFC 6151, March 2011.

[DDS13] Itai Dinur, Orr Dunkelman, and Adi Shamir. Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials. In *FSE*, pages 219–240. Springer, 2013.

[DDS14] Itai Dinur, Orr Dunkelman, and Adi Shamir. Improved Practical Attacks on Round-Reduced KECCAK. *J. Cryptology*, 27:183–209, 2014.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, 1st edition, March 2003.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[HPR04] Philip Hawkes, Michael Paddon, and Gregory G. Rose. Musings on the wang et al. md5 collision. Cryptology ePrint Archive, Report 2004/264, 2004. Available at http://eprint.iacr.org/2004/264.

[Int17] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, March 2017.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner. *SHA-3 Derived Functions: CSHAKE, KMAC, TupleHash and ParallelHash.* NIST special publication; NIST special pub; NIST SP. National Institute of Standards and Technology, December 2016. Available at https://doi.org/10.6028/NIST.SP.800-185.

[Kli05] Vlastimil Klima. Finding MD5 Collisions – a Toy For a Notebook. Cryptology ePrint Archive, Report 2005/075, March 2005. Available at http://eprint.iacr.org/2005/075.

[LWdW05] Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding X.509 Certificates. Cryptology ePrint Archive, Report 2005/067, 2005. Available at http://eprint.iacr.org/2005/067.

[Mer82] R.C. Merkle. Method of providing digital signatures, January 1982. US Patent 4,309,569.

[MGS15] Hristina Mihajloska, Danilo Gligoroski, and Simona Samardjiska. Reviving the idea of incremental cryptography for the zettabyte era use case: Incremental hash functions based on sha-3. In Jan Camenisch and Dogan Kesdogan, editors, *iNetSeC*, volume 9591 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2015.

[Nat95] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard.* National Institute of Standards and Technology, April 1995.

[Nat15] National Institute of Standards and Technology. *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* National Institute of Standards and Technology, August 2015. Available at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[Ora10] Oracle Corporation. *UltraSPARC Architecture 2007*, September 2010.

[PC16] Jaime Pérez Crespo. Implementation of the iSHAKE incremental hash algorithm. Technical report, June 2016.

[QSLG17] Kexin Qiao, Ling Song, Meicheng Liu, and Jian Guo. New Collision Attacks on Round-Reduced Keccak. Cryptology ePrint Archive, Report 2017/128, February 2017. Available at http://eprint.iacr.org/2017/128.

[Riv92] R. Rivest. The MD5 Message-Digest Algorithm. Internet Requests for Comments, April 1992.

[RO05] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. Cryptology ePrint Archive, Report 2005/010, 2005. Available at http://eprint.iacr.org/2005/010.

[RS09] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. Cryptology ePrint Archive, Report 2004/035, July 2009. Available at http://eprint.iacr.org/2004/035.

[SBK$^+$17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190, 2017. Available at http://eprint.iacr.org/2017/190.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition.* Wiley, 2nd edition, October 1996.

[Sig15] Silent Signal. Poisonous MD5 – Wolves Among the Sheep. https://blog.silentsignal.eu/2015/06/10/poisonous-md5-wolves-among-the-sheep/, 2015.

[Sin11] E. Sink. *Version Control by Example*, chapter 12, pages 171–173. Pyrenean Gold Press, 2011. Available at http://ericsink.com/vcbe/html/cryptographic_hashes.html.

[SKP15] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full SHA-1. Cryptology ePrint Archive, Report 2015/967, 2015. Available at http://eprint.iacr.org/2015/967.

[SLG17] Ling Song, Guohong Liao, and Jian Guo. Non-Full Sbox Linearization: Applications to Collision Attacks on Round-Reduced Keccak. Crypto 2017 Proceedings, August 2017. To appear.

[Ste12] Marc Stevens. Single-block collision attack on MD5. Cryptology ePrint Archive, Report 2012/040, 2012. Available at http://eprint.iacr.org/2012/040.

[Wag02] David Wagner. A Generalized Birthday Problem. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

[Wic11] Hadley Wickham. The Split-Apply-Combine Strategy for Data Analysis. *Journal of Statistical Software, Articles*, 40(1):1–29, 2011.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005. Available at http://www.iacr.org/cryptodb/archive/2005/CRYPTO/1826/1826.pdf.

[XF10] Tao Xie and Dengguo Feng. Construct MD5 Collisions Using Just A Single Block Of Message. Cryptology ePrint Archive, Report 2010/643, 2010. Available at http://eprint.iacr.org/2010/643.

[XLF13] Tao Xie, Fanbao Liu, and Dengguo Feng. Fast Collision Attack on MD5. Cryptology ePrint Archive, Report 2013/170, 2013. Available at http://eprint.iacr.org/2013/170.

Appendix A
Header file of the iSHAKE library

ishake.h

```
#include <stdint.h>
#include <math.h>
#include <pthread.h>

#include "modulo_arithmetics.h"

#ifndef _ISHAKE_H
#define _ISHAKE_H

/*
 * Define a default block size of 100KiB
 */
#ifndef ISHAKE_BLOCK_SIZE
#define ISHAKE_BLOCK_SIZE 100*1024
#endif

/*
 * Constants for the two different modes of operation.
 */
#define ISHAKE_APPEND_ONLY_MODE 0 // fixed-size data
#define ISHAKE_FULL_MODE 1        // variable-size data

/*
 * Type definition for a function that obtains the hash of some given
 * data. This function is used internally and will be mapped to a
 * function in the set of shake*)() or sha3*() functions.
 */
typedef int (*hash_function)(
    uint8_t*,
    size_t,
    const uint8_t*,
    size_t
);

/* data types */

```

```
/**
 * Type used internally to represent the ID header appended to blocks
 * in the variable size mode of iSHAKE.
 */
typedef struct {
    uint64_t nonce;
    uint64_t prev;
} ishake_nonce;

/**
 * The header appended to a block.
 */
typedef struct {
    unsigned char length;
    union {
        ishake_nonce nonce;
        uint64_t idx;
    } value;
} ishake_header;

/**
 * A block to be used in the iSHAKE algorithm.
 */
typedef struct {
    unsigned char *data;
    uint32_t data_lenp;
    ishake_header header;
} ishake_block_t;

/**
 * A task for a thread to run the iSHAKE algorithm on a block.
 */
typedef struct _task_t {
    group_op op;
    ishake_block_t *block;
    struct _task_t *prev;
} ishake_task_t;
typedef ishake_task_t* ishake_stack_t;

/*
 * Type definition of the ishake main structure. It keeps the status of
 * the algorithm at any given point in time.
 */
typedef struct {
    uint8_t mode;            // mode of operation
    uint64_t block_no;       // # of blocks processed
    uint32_t block_size;     // size of blocks
    uint16_t output_len;     // size of output (in bytes)
    uint64_t proc_bytes;     // # of bytes processed
    uint32_t remaining;      // # of bytes remaining
    uint64_t *hash;          // intermediate hash
    unsigned char *buf;      // buffer for data remaining

    // threading related properties
    uint16_t thrd_no;         // number of threads to use
    pthread_t **threads;      // list of threads
    uint8_t done;             // flag to signal we are done

    pthread_mutex_t stack_lck;       // a mutex to control the stack
    pthread_mutex_t combine_lck;     // a mutex for combine ops
    pthread_cond_t data_available;  // condition for data availability
    ishake_stack_t stack;            // the stack containing tasks
} ishake_t;

/* ishake_* API functions */

/**
 * Initialize a hash. This function must be called in order to obtain a
 * valid ishake state to use through the rest of the API.
 */
int ishake_init(
    ishake_t *is,         // the iSHAKE struct
    uint32_t blk_size,    // the block size to use
    uint16_t hashbitlen, // the length in bits of the output
    uint8_t mode,         // the mode of operation
    uint16_t threads      // the number of worker threads to use
);


/**
 * Append data to be hashed. Its size doesn't need to be multiple of
 * the block size.
 */
int ishake_append(
    ishake_t *is,         // the iSHAKE struct
    unsigned char *data, // the data to append
    uint64_t len          // the length in bytes of the data given
);


/**
 * Insert a new block right before another. Its size MUST be exactly
 * the same as the block size, and the prev pointer in the block
 * must point to the previous block in the list.
 *
 * Pass NULL as next when inserting the last block.
 */
int ishake_insert(
    ishake_t *is,         // the iSHAKE struct
    ishake_block_t *new, // the new block
    ishake_block_t *next // the next block
);


```

```c
/**
 * Delete a block, updating the next block in the list to point to the
 * block previous to the one we are deleting.
 *
 * Pass NULL as next when deleting the last block.
 */
int ishake_delete(
    ishake_t *is,             // the iSHAKE struct
    ishake_block_t *deleted, // the block to delete
    ishake_block_t *next     // the next block
);


/**
 * Update a block with new data. Old data must be provided too.
 */
int ishake_update(
    ishake_t *is,         // the iSHAKE struct
    ishake_block_t *old, // the old block
    ishake_block_t *new  // the new, modified block
);


/**
 * Finalise the process and get the resulting digest.
 */
int ishake_final(
    ishake_t *is,    // the iSHAKE struct
    uint8_t *output // a buffer to store the digest
);


/**
 * Obtain the hash corresponding to some chunk of data.
 */
int ishake_hash(
    unsigned char *data, // input data
    uint64_t len,         // the input length in bytes
    uint8_t *hash,        // a buffer to store the digest
    uint16_t hashbitlen  // the length of the digest
);


/**
 * Obtain the hash corresponding to some piece of data, performing
 * the computation in parallel by threadno threads.
 *
 * Define ISHAKE_BLOCK_SIZE if you wish to modify the block size.
 */
int ishake_hash_p(
    unsigned char *data, // input data
    uint64_t len,         // the input length in bytes
    uint8_t *hash,        // a buffer to store the digest
    uint16_t hashbitlen, // the length of the digest
    uint16_t threadno    // the number of worker threads to use
);


/**
 * Cleanup the resources associated to a given iSHAKE struct.
 */
void ishake_cleanup(
    ishake_t *is // the iSHAKE struct
);

#endif // _ISHAKE_H
```

Appendix B
Usage of the tools provided

sha3sum

```
sha3sum [--shake128|--shake256|--sha3-224|--sha3-256|--sha3-384|
         --sha3-512] [--bytes N] [--hex] [--quiet] [--help] [file]

        --shake128      Use 128 bit SHAKE.
        --shake256      Use 256 bit SHAKE.
        --sha3-224      Use 224 bit SHA3.
        --sha3-256      Use 256 bit SHA3. Default
        --sha3-384      Use 384 bit SHA3.
        --sha3-512      Use 512 bit SHA3.
        --bytes         The number of bytes desired in the output. Only
                        for the SHAKE algorithm.
        --hex           Input is hex-encoded. Defaults to binary input.
        --quiet         Output only the resulting hash string.
        --help          Print this help.
        file            The file to hash. Data can also be piped into
                        the program.
```

ishakesum

```
ishakesum [--128|--256] [--hex] [--bits N] [--block-size N] [--quiet]
          [--threads] [--profile] [--help] [file]

        --128           Use 128 bit equivalent iSHAKE. Default.
        --256           Use 256 bit equivalent iSHAKE.
        --hex           Input is hex-encoded. Defaults to binary input.
        --bits          The number of bits desired in the output. Must
                        be a multiple of 64. Between 2688 and 4160
                        for iSHAKE 128, and between 6528 and 16512
                        for iSHAKE 256. The lowest number for each
                        version is the default.
        --threads       The number of threads to use. No threads are
                        used by default.
        --profile       Measure the performance of the operation(s) to
                        run.
        --block-size    The size in bytes of the iSHAKE internal blocks.
        --quiet         Output only the resulting hash string.
        --help          Print this help.
        file            The file to hash. Data can also be piped into
                        the program.
```

sha3sumd

```
sha3sumd [--128|--256] [--bits N] [--block-size N] [--quiet] [--help]
         directory

        --shake128      Use 128 bit SHAKE.
        --shake256      Use 256 bit SHAKE.
        --sha3-224      Use 224 bit SHA3.
        --sha3-256      Use 256 bit SHA3. Default
        --sha3-384      Use 384 bit SHA3.
        --sha3-512      Use 512 bit SHA3.
        --bytes         The number of bytes desired in the output. Only
                        for the SHAKE algorithm.
        --quiet         Output only the resulting hash string.
        --help          Print this help.
        directory       The path to a directory whose contents will be
                        hashed in order. Every file in the directory
                        will be read and incorporated into the input,
                        in the same order as they appear in the
                        directory.
```

ishakesumd

```
ishakesumd [--128|--256] [--bits N] [--block-size N] [--quiet] [--help]
           directory

        --128           Use 128 bit equivalent iSHAKE. Default.
        --256           Use 256 bit equivalent iSHAKE.
        --bits          The number of bits desired in the output. Must
                        be a multiple of 64. Between 2688 and 4160
                        for iSHAKE 128, and between 6528 and 16512
                        for iSHAKE 256. The lowest number for each
                        version is the default.
        --block-size    The size in bytes of the iSHAKE internal blocks.
        --mode          The mode of operation, one of FULL or
                        APPEND_ONLY. Defaults to APPEND_ONLY.
        --rehash        The hash to use as a base, computing only those
                        blocks that have changed.
        --threads       The number of threads to use. No threads are
                        used by default.
        --profile       Measure the performance of the operation(s) to
                        run.
        --quiet         Output only the resulting hash string.
        --help          Print this help.
        directory       The path to a directory whose contents will be
                        hashed in order. Every file in the directory
                        will be read and incorporated into the input,
                        in the same order as they appear in the
```

```
                                directory.
```

combine

```
combine [--add|--sub] hash1 hash2

          --add               Apply the addition operation. Default.
          --sub               Apply the subtraction operation.
          --help              Print this help.
          hash1               The first operand to the operation requested.
          hash2               The second operand to the operation requested.
```

Appendix C
Performance measurements

C.1 Hash algorithm comparison

nepenthe

`AMD64; Haswell/Crystalwell (0306C3h); 2014 Intel Core i7 4980HQ; 4x2800MHz`

Table C.1: Time taken (in seconds) by different algorithms to process 1 GiB of data, sliced in blocks of 52 KiB (MD5, SHA1, BLAKE2b, BLAKE2bp, iSHAKE 128 and iSHAKE 256).

MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (7 threads, generic optimisations)	**iSHAKE 256** (7 threads, generic optimisations)	**iSHAKE 128** (7 threads, lane complementing)	**iSHAKE 256** (7 threads, lane complementing)
1,614	2,516	1,595	2,839	0,668	0,922	0,694	0,871
1,613	2,487	1,585	2,853	0,697	0,915	0,701	0,869
1,62	2,478	1,589	2,851	0,776	0,936	0,754	0,904
1,637	2,465	1,576	2,85	0,689	0,916	0,725	0,9
1,657	2,469	1,608	2,831	0,676	0,93	0,751	0,913
1,633	2,482	1,579	2,832	0,688	0,924	0,719	0,914
1,615	2,472	1,581	2,829	0,681	0,942	0,845	0,921
1,626	2,502	1,584	2,867	0,692	0,952	0,815	0,941
1,627	2,483	1,593	2,841	0,67	0,957	0,768	0,949
1,619	2,49	1,597	2,829	0,675	0,97	0,799	0,889
1,647	2,537	1,594	2,834	0,701	0,975	0,811	0,886

Continued on next page

Table C.1 – continued from previous page

MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (7 threads, generic optimisations)	**iSHAKE 256** (7 threads, generic optimisations)	**iSHAKE 128** (7 threads, lane complementing)	**iSHAKE 256** (7 threads, lane complementing)
$1,627$	$2,523$	$1,592$	$2,817$	$0,694$	$0,989$	$0,847$	$0,854$
$1,613$	$2,524$	$1,597$	$2,844$	$0,685$	$0,963$	$0,87$	1
$1,615$	$2,603$	$1,588$	$2,853$	$0,706$	$0,895$	$0,84$	$0,947$
$1,601$	$2,48$	$1,581$	$2,844$	$0,727$	$0,927$	$0,894$	$0,967$
$1,609$	$2,485$	$1,591$	$2,83$	$0,769$	$0,938$	$0,875$	$0,939$
$1,624$	$2,54$	$1,582$	$2,837$	$0,679$	$0,898$	$0,947$	$1,054$
$1,623$	$2,469$	$1,585$	$2,836$	$0,677$	$0,925$	$1,083$	$1,008$
$1,615$	$2,477$	$1,593$	$2,828$	$0,672$	$0,897$	$0,915$	$1,007$
$1,612$	$2,477$	$1,581$	$2,835$	$0,68$	$0,909$	$0,906$	$0,908$
$1,598$	$2,509$	$1,633$	$2,835$	$0,689$	$0,829$	$0,917$	$0,925$
$1,613$	$2,477$	$1,611$	$2,831$	$0,706$	$0,841$	$0,878$	$0,963$
$1,598$	$2,47$	$1,601$	$2,832$	$0,704$	$0,835$	$1,008$	$0,98$
$1,616$	$2,457$	$1,609$	$2,864$	$0,694$	$0,827$	$1,144$	$1,008$
$1,602$	$2,478$	$1,606$	$2,83$	$0,7$	$0,85$	$1,115$	$1,034$
$1,613$	$2,524$	$1,608$	$2,843$	$0,739$	$0,969$	$0,942$	$1,038$
$1,607$	$2,467$	$1,615$	$2,796$	$0,708$	$0,877$	$0,879$	$1,003$
$1,596$	$2,464$	$1,607$	$2,682$	$0,684$	$0,915$	$0,969$	$0,967$
$1,594$	$2,466$	$1,607$	$2,969$	$0,681$	$0,92$	$0,876$	$1,02$
$1,599$	$2,471$	$1,602$	$3,136$	$0,688$	$1,026$	$0,914$	$0,994$
$1,617$	$2,541$	$1,601$	$2,838$	$0,691$	$0,945$	$0,906$	$1,017$
$1,6$	$2,539$	$1,598$	$2,826$	$0,695$	$0,925$	$0,891$	$1,08$
$1,603$	$2,539$	$1,609$	$2,828$	$0,706$	$0,966$	$0,909$	$1,134$
$1,604$	$2,527$	$1,597$	$2,847$	$0,726$	$0,962$	$0,964$	$1,073$
$1,59$	$2,543$	$1,591$	$2,852$	$0,723$	$0,961$	$0,897$	$1,127$
$1,59$	$2,554$	$1,601$	$2,846$	$0,72$	$0,955$	$0,91$	$1,184$
$1,597$	$2,56$	$1,59$	$2,813$	$0,703$	$0,955$	$0,919$	$1,12$
$1,602$	$2,612$	$1,6$	$2,832$	$0,7$	$0,95$	$0,865$	$1,07$

Continued on next page

Table C.1 – continued from previous page

MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (7 threads, generic optimisations)	**iSHAKE 256** (7 threads, generic optimisations)	**iSHAKE 128** (7 threads, lane complementing)	**iSHAKE 256** (7 threads, lane complementing)
1, 601	2, 571	1, 589	2, 83	0, 675	1, 025	0, 898	1, 017
1, 602	2, 572	1, 598	2, 835	0, 693	0, 951	1, 07	1, 039
1, 608	2, 589	1, 597	2, 835	0, 716	0, 972	0, 953	1, 045
1, 616	2, 567	1, 592	2, 844	0, 69	1	0, 924	0, 967
1, 599	2, 547	1, 592	2, 832	0, 707	0, 999	0, 932	0, 996
1, 592	2, 496	1, 591	2, 885	0, 728	0, 971	0, 946	1, 034
1, 601	2, 558	1, 59	2, 913	0, 716	1, 005	0, 947	1, 105
1, 599	2, 474	1, 608	2, 887	0, 735	1, 016	0, 992	0, 93
1, 592	2, 472	1, 597	2, 828	0, 73	0, 84	0, 992	0, 97
1, 608	2, 46	1, 58	2, 836	0, 73	0, 858	0, 932	0, 994
1, 606	2, 477	1, 595	2, 83	0, 749	0, 831	1, 054	1, 132
1, 595	2, 458	1, 596	2, 838	0, 708	0, 842	0, 959	1, 085

bigmem

`AMD64; Nehalem (206e6); 2010 Intel Xeon X7560; 32 x 2266MHz`

Table C.2: Time taken (in seconds) by different algorithms to process 1 GiB of data sliced in blocks of 52 KiB (MD5, SHA1, BLAKE2b, BLAKE2bp, iSHAKE 128 and iSHAKE 256).

MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (8 threads, generic optimisations)	**iSHAKE 256** (8 threads, generic optimisations)	**iSHAKE 128** (8 threads, lane complementing)	**iSHAKE 256** (8 threads, lane complementing)
2, 8	4, 21	2, 55	3, 91	1, 36	1, 63	1, 36	1, 58
2, 75	4, 37	2, 52	3, 81	0, 87	1, 46	0, 89	1, 07
2, 76	4, 16	2, 57	3, 58	1, 22	1, 14	0, 93	1, 4

Continued on next page

Table C.2 – continued from previous page

MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (8 threads, generic optimisations)	**iSHAKE 256** (8 threads, generic optimisations)	**iSHAKE 128** (8 threads, lane complementing)	**iSHAKE 256** (8 threads, lane complementing)
2,9	5,47	2,63	3,93	0,89	1,19	0,95	1,18
2,76	4,35	2,22	3,84	1,04	1,52	1,29	1,1
2,8	3,92	2,71	3,88	0,86	1,13	0,81	1,11
2,91	4,21	2,58	4,01	0,96	1,66	1,33	1,41
2,75	3,8	2,23	4,16	0,96	1,25	0,9	1,3
2,81	4,09	2,52	3,2	0,96	1,02	0,93	0,88
2,75	4,14	2,67	3,25	1,29	1,62	1,05	1,11
2,46	4,34	2,29	3,97	1,32	1,5	1,37	1,42
2,79	4,15	2,52	3,8	1,45	1,49	1,15	1,13
2,91	4,33	2,52	3,7	1,29	1,29	1,17	1,65
2,96	3,87	2,66	3,56	1,35	1,16	0,82	1,39
2,98	4,09	2,56	2,87	1,41	1,39	0,89	1,5
2,99	4,15	2,52	3,19	0,89	1,25	1,29	1,49
2,99	4,36	2,22	3,18	1,23	1,43	0,95	1,22
2,99	4,14	2,52	3,86	0,93	1,38	1,18	1,22
2,99	4,35	2,65	3,79	1,53	1,14	0,77	1,24
2,55	4,13	2,56	4,04	0,96	1,29	1,21	1,32
2,75	4,36	2,22	3,09	1,48	1,22	1,06	0,97
2,53	4,15	2,68	3,78	0,96	1,5	1,04	1,56
2,45	3,79	2,32	3,19	0,93	1,22	0,95	1
2,54	4,17	2,52	3,44	0,99	1,23	1,21	1,55
2,78	4,2	2,52	3,03	1,12	1,48	0,91	1,04
2,91	4,23	2,64	3,16	0,79	1,39	1,29	1,28
2,96	4,09	2,55	3,32	0,95	1,3	0,76	1,43
2,98	4,33	2,52	3,18	1,03	1,15	0,81	1,21
2,56	3,86	2,22	4,14	1,29	1,03	1,32	1,44
2,75	4,21	2,66	3,67	0,9	1,53	0,73	1,38

Continued on next page

Table C.2 – continued from previous page

MD5	**SHA1**	**BLAKE2b**	**BLAKE2bp**	**iSHAKE 128** (8 threads, generic optimisations)	**iSHAKE 256** (8 threads, generic optimisations)	**iSHAKE 128** (8 threads, lane complementing)	**iSHAKE 256** (8 threads, lane complementing)
2,8	3,79	2,29	3,98	1,4	1,14	1,21	1,51
2,75	4,2	2,68	4,4	0,79	1,28	0,72	1,36
2,76	4,37	2,57	3,65	1,18	1,47	0,83	1,03
2,89	3,89	2,52	3,7	0,91	1,49	1,1	0,97
2,96	4,22	2,78	3,75	1,02	1,28	0,78	0,94
2,98	3,79	2,64	3,81	1,32	1,25	1,01	1,4
2,56	4,1	2,55	4,22	1,25	1,56	1,41	1,02
2,76	4,14	2,22	2,53	1,01	1,2	0,88	1,34
2,45	4,34	2,68	2,29	0,91	1,38	0,93	1,5
2,56	5,03	2,57	3,35	0,88	1,48	1,37	1,52
2,77	4,12	2,22	3,89	1,29	1,15	0,81	1,03
2,9	4,34	2,68	3,19	0,89	1,16	0,93	1,05
2,75	4,13	2,57	3,88	0,98	1,04	1,22	1,22
2,81	4,36	2,53	3,21	1,05	1,56	0,76	1,04
2,76	4,15	2,22	3,2	1,34	1,35	0,91	1,56
2,46	4,33	2,67	3,22	0,92	1,48	1,08	1,36
2,46	4,8	2,56	4,2	1,06	1,08	0,86	1,31
2,8	4,26	2,22	3,8	1,18	1,59	1,31	1,5
2,91	4,09	2,68	4,23	1,14	1,26	1,13	1,02
2,95	4,34	2,57	4,14	1,53	1,78	1,32	1

C.2 iSHAKE evaluation

nepenthe

`AMD64; Haswell/Crystalwell (0306C3h); 2014 Intel Core i7 4980HQ; 4x2800MHz`

Table C.3: Performance measurements for **iSHAKE 128** with different configurations and a fixed block size of 8192 bytes (8 KiB), **including** the block header.

blocks	total bytes	threads	cycles/byte	wall time
1	8192	0	12, 923	0, 00004
2	16384	0	12, 919	0, 00008
4	32768	0	12, 814	0, 00015
8	65536	0	11, 406	0, 00027
16	131072	0	10, 553	0, 00049
32	262144	0	9, 623	0, 0009
64	524288	0	8, 941	0, 00168
1	8192	1	28, 547	0, 00008
2	16384	1	19, 507	0, 00012
4	32768	1	12, 093	0, 00014
8	65536	1	9, 172	0, 00022
16	131072	1	8, 691	0, 00041
32	262144	1	8, 404	0, 00079
64	524288	1	8, 369	0, 00157
1	8192	2	33, 235	0, 0001
2	16384	2	16, 859	0, 0001
4	32768	2	8, 153	0, 0001
8	65536	2	5, 771	0, 00014
16	131072	2	5, 078	0, 00024
32	262144	2	4, 634	0, 00044
64	524288	2	4, 47	0, 00084
1	8192	3	27, 65	0, 00008
2	16384	3	15, 731	0, 00009
4	32768	3	7, 061	0, 00008
8	65536	3	5, 155	0, 00012
16	131072	3	4, 332	0, 0002
32	262144	3	3, 647	0, 00034
64	524288	3	3, 144	0, 00059

Continued on next page

Table C.3 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
1	8192	4	27, 29	0, 00008
2	16384	4	14, 149	0, 00008
4	32768	4	10, 027	0, 00012
8	65536	4	5, 91	0, 00014
16	131072	4	3, 637	0, 00017
32	262144	4	3, 299	0, 00031
64	524288	4	2, 728	0, 00051
1	8192	5	31, 472	0, 00009
2	16384	5	17, 768	0, 0001
4	32768	5	8, 615	0, 0001
8	65536	5	5, 419	0, 00013
16	131072	5	3, 819	0, 00018
32	262144	5	3, 076	0, 00029
64	524288	5	2, 606	0, 00049
1	8192	6	34, 9	0, 0001
2	16384	6	18, 389	0, 00011
4	32768	6	10, 31	0, 00012
8	65536	6	5, 965	0, 00014
16	131072	6	3, 96	0, 00019
32	262144	6	3, 109	0, 00029
64	524288	6	2, 681	0, 0005
1	8192	7	38, 64	0, 00011
2	16384	7	25, 307	0, 00015
4	32768	7	14, 813	0, 00017
8	65536	7	6, 762	0, 00016
16	131072	7	4, 356	0, 0002
32	262144	7	3, 241	0, 00031
64	524288	7	2, 754	0, 00052
1	8192	8	45, 787	0, 00014
2	16384	8	23, 941	0, 00014
4	32768	8	12, 12	0, 00014

Continued on next page

Table C.3 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
8	65536	8	$7,156$	$0,00017$
16	131072	8	$4,397$	$0,00021$
32	262144	8	$3,384$	$0,00032$
64	524288	8	$2,78$	$0,00052$
1	8192	9	$64,761$	$0,00019$
2	16384	9	$30,952$	$0,00018$
4	32768	9	$16,564$	$0,0002$
8	65536	9	$9,397$	$0,00022$
16	131072	9	$6,133$	$0,00029$
32	262144	9	$4,591$	$0,00043$
64	524288	9	$3,481$	$0,00065$
1	8192	10	$76,751$	$0,00023$
2	16384	10	$30,936$	$0,00018$
4	32768	10	$17,84$	$0,00021$
8	65536	10	$9,893$	$0,00023$
16	131072	10	$5,77$	$0,00027$
32	262144	10	$4,004$	$0,00038$
64	524288	10	$3,225$	$0,00061$
1	8192	11	$59,6$	$0,00017$
2	16384	11	$30,821$	$0,00018$
4	32768	11	$16,722$	$0,0002$
8	65536	11	$9,485$	$0,00022$
16	131072	11	$6,102$	$0,00029$
32	262144	11	$4,208$	$0,0004$
64	524288	11	$3,253$	$0,00061$
1	8192	12	$69,769$	$0,00021$
2	16384	12	$42,364$	$0,00025$
4	32768	12	$18,804$	$0,00022$
8	65536	12	$10,659$	$0,00025$
16	131072	12	$6,432$	$0,0003$
32	262144	12	$4,366$	$0,00041$

Continued on next page

Table C.3 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
64	524288	12	3, 435	0, 00065
1	8192	13	64, 471	0, 00019
2	16384	13	35, 815	0, 00021
4	32768	13	21, 334	0, 00025
8	65536	13	12, 546	0, 0003
16	131072	13	7, 598	0, 00036
32	262144	13	5, 12	0, 00048
64	524288	13	4, 023	0, 00076

Table C.4: Performance measurements for **iSHAKE 128** with different configurations and a fixed block size of 8192 bytes (8 KiB), **excluding** the block header.

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
1	8192	0	10, 275	0, 00003
2	16384	0	10, 232	0, 00006
4	32768	0	9, 868	0, 00012
8	65536	0	9, 381	0, 00022
16	131072	0	8, 724	0, 00041
32	262144	0	8, 09	0, 00076
64	524288	0	7, 501	0, 00141
1	8192	1	16, 881	0, 00005
2	16384	1	10, 547	0, 00006
4	32768	1	9, 149	0, 00011
8	65536	1	7, 542	0, 00018
16	131072	1	7, 022	0, 00033
32	262144	1	6, 688	0, 00063
64	524288	1	6, 496	0, 00122
1	8192	2	18, 87	0, 00006
2	16384	2	9, 488	0, 00006
4	32768	2	6, 379	0, 00008

Continued on next page

Table C.4 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
8	65536	2	4, 842	0, 00011
16	131072	2	4, 241	0, 0002
32	262144	2	3, 747	0, 00035
64	524288	2	3, 519	0, 00066
1	8192	3	37, 812	0, 00011
2	16384	3	15, 382	0, 00009
4	32768	3	8, 086	0, 0001
8	65536	3	4, 458	0, 00011
16	131072	3	3, 548	0, 00017
32	262144	3	2, 857	0, 00027
64	524288	3	2, 673	0, 0005
1	8192	4	23, 252	0, 00007
2	16384	4	14, 845	0, 00009
4	32768	4	8, 669	0, 0001
8	65536	4	5, 237	0, 00012
16	131072	4	3, 807	0, 00018
32	262144	4	3, 588	0, 00034
64	524288	4	3, 262	0, 00061
1	8192	5	33, 728	0, 0001
2	16384	5	20, 05	0, 00012
4	32768	5	10, 349	0, 00012
8	65536	5	6, 702	0, 00016
16	131072	5	5, 481	0, 00026
32	262144	5	3, 468	0, 00033
64	524288	5	2, 694	0, 00051
1	8192	6	44, 468	0, 00013
2	16384	6	23, 143	0, 00014
4	32768	6	12, 02	0, 00014
8	65536	6	6, 505	0, 00015
16	131072	6	4, 125	0, 00019
32	262144	6	3, 044	0, 00029

Continued on next page

Table C.4 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
64	524288	6	$2,57$	$0,00048$
1	8192	7	$65,234$	$0,00019$
2	16384	7	$35,43$	$0,00021$
4	32768	7	$15,908$	$0,00019$
8	65536	7	$8,969$	$0,00021$
16	131072	7	$4,806$	$0,00023$
32	262144	7	$3,143$	$0,0003$
64	524288	7	$2,608$	$0,00049$
1	8192	8	$48,723$	$0,00014$
2	16384	8	$26,579$	$0,00016$
4	32768	8	$13,507$	$0,00016$
8	65536	8	$7,74$	$0,00018$
16	131072	8	$4,751$	$0,00022$
32	262144	8	$3,36$	$0,00032$
64	524288	8	$2,643$	$0,0005$
1	8192	9	$56,407$	$0,00016$
2	16384	9	$29,034$	$0,00017$
4	32768	9	$15,355$	$0,00018$
8	65536	9	$8,785$	$0,00021$
16	131072	9	$5,056$	$0,00024$
32	262144	9	$3,817$	$0,00036$
64	524288	9	$3,419$	$0,00064$
1	8192	10	$72,015$	$0,00021$
2	16384	10	$33,352$	$0,0002$
4	32768	10	$18,112$	$0,00021$
8	65536	10	$9,044$	$0,00021$
16	131072	10	$5,707$	$0,00027$
32	262144	10	$3,772$	$0,00036$
64	524288	10	$2,923$	$0,00055$
1	8192	11	$65,055$	$0,00019$
2	16384	11	$35,221$	$0,00021$

Continued on next page

Table C.4 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
4	32768	11	19, 537	0, 00023
8	65536	11	10, 675	0, 00025
16	131072	11	6, 041	0, 00028
32	262144	11	5, 024	0, 00047
64	524288	11	4, 066	0, 00076
1	8192	12	83, 212	0, 00024
2	16384	12	38, 766	0, 00023
4	32768	12	20, 274	0, 00024
8	65536	12	10, 726	0, 00025
16	131072	12	6, 352	0, 0003
32	262144	12	4, 125	0, 00039
64	524288	12	3, 461	0, 00065
1	8192	13	79, 755	0, 00023
2	16384	13	41, 705	0, 00024
4	32768	13	21, 467	0, 00025
8	65536	13	10, 992	0, 00026
16	131072	13	6, 57	0, 00031
32	262144	13	4, 827	0, 00045
64	524288	13	3, 637	0, 00068

Table C.5: Performance measurements for **iSHAKE 256** with different configurations and a fixed block size of 8192 bytes (8 KiB), **including** the block header.

blocks	total bytes	threads	cycles/byte	wall time
1	8192	0	12, 923	0, 00004
2	16384	0	12, 919	0, 00008
4	32768	0	12, 814	0, 00015
8	65536	0	11, 406	0, 00027
16	131072	0	10, 553	0, 00049
32	262144	0	9, 623	0, 0009

Continued on next page

Table C.5 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
64	524288	0	8,941	0,00168
1	8192	1	28,547	0,00008
2	16384	1	19,507	0,00012
4	32768	1	12,093	0,00014
8	65536	1	9,172	0,00022
16	131072	1	8,691	0,00041
32	262144	1	8,404	0,00079
64	524288	1	8,369	0,00157
1	8192	2	33,235	0,0001
2	16384	2	16,859	0,0001
4	32768	2	8,153	0,0001
8	65536	2	5,771	0,00014
16	131072	2	5,078	0,00024
32	262144	2	4,634	0,00044
64	524288	2	4,47	0,00084
1	8192	3	27,65	0,00008
2	16384	3	15,731	0,00009
4	32768	3	7,061	0,00008
8	65536	3	5,155	0,00012
16	131072	3	4,332	0,0002
32	262144	3	3,647	0,00034
64	524288	3	3,144	0,00059
1	8192	4	27,29	0,00008
2	16384	4	14,149	0,00008
4	32768	4	10,027	0,00012
8	65536	4	5,91	0,00014
16	131072	4	3,637	0,00017
32	262144	4	3,299	0,00031
64	524288	4	2,728	0,00051
1	8192	5	31,472	0,00009
2	16384	5	17,768	0,0001

Continued on next page

Table C.5 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
4	32768	5	$8,615$	$0,0001$
8	65536	5	$5,419$	$0,00013$
16	131072	5	$3,819$	$0,00018$
32	262144	5	$3,076$	$0,00029$
64	524288	5	$2,606$	$0,00049$
1	8192	6	$34,9$	$0,0001$
2	16384	6	$18,389$	$0,00011$
4	32768	6	$10,31$	$0,00012$
8	65536	6	$5,965$	$0,00014$
16	131072	6	$3,96$	$0,00019$
32	262144	6	$3,109$	$0,00029$
64	524288	6	$2,681$	$0,0005$
1	8192	7	$38,64$	$0,00011$
2	16384	7	$25,307$	$0,00015$
4	32768	7	$14,813$	$0,00017$
8	65536	7	$6,762$	$0,00016$
16	131072	7	$4,356$	$0,0002$
32	262144	7	$3,241$	$0,00031$
64	524288	7	$2,754$	$0,00052$
1	8192	8	$45,787$	$0,00014$
2	16384	8	$23,941$	$0,00014$
4	32768	8	$12,12$	$0,00014$
8	65536	8	$7,156$	$0,00017$
16	131072	8	$4,397$	$0,00021$
32	262144	8	$3,384$	$0,00032$
64	524288	8	$2,78$	$0,00052$
1	8192	9	$64,761$	$0,00019$
2	16384	9	$30,952$	$0,00018$
4	32768	9	$16,564$	$0,0002$
8	65536	9	$9,397$	$0,00022$
16	131072	9	$6,133$	$0,00029$

Continued on next page

Table C.5 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
32	262144	9	$4,591$	$0,00043$
64	524288	9	$3,481$	$0,00065$
1	8192	10	$76,751$	$0,00023$
2	16384	10	$30,936$	$0,00018$
4	32768	10	$17,84$	$0,00021$
8	65536	10	$9,893$	$0,00023$
16	131072	10	$5,77$	$0,00027$
32	262144	10	$4,004$	$0,00038$
64	524288	10	$3,225$	$0,00061$
1	8192	11	$59,6$	$0,00017$
2	16384	11	$30,821$	$0,00018$
4	32768	11	$16,722$	$0,0002$
8	65536	11	$9,485$	$0,00022$
16	131072	11	$6,102$	$0,00029$
32	262144	11	$4,208$	$0,0004$
64	524288	11	$3,253$	$0,00061$
1	8192	12	$69,769$	$0,00021$
2	16384	12	$42,364$	$0,00025$
4	32768	12	$18,804$	$0,00022$
8	65536	12	$10,659$	$0,00025$
16	131072	12	$6,432$	$0,0003$
32	262144	12	$4,366$	$0,00041$
64	524288	12	$3,435$	$0,00065$
1	8192	13	$64,471$	$0,00019$
2	16384	13	$35,815$	$0,00021$
4	32768	13	$21,334$	$0,00025$
8	65536	13	$12,546$	$0,0003$
16	131072	13	$7,598$	$0,00036$
32	262144	13	$5,12$	$0,00048$
64	524288	13	$4,023$	$0,00076$

Table C.6: Performance measurements for **iSHAKE 256** with different configurations and a fixed block size of 8192 bytes (8 KiB), **excluding** the block header.

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
1	8192	0	$13,585$	$0,00004$
2	16384	0	$11,863$	$0,00007$
4	32768	0	$11,908$	$0,00014$
8	65536	0	$11,601$	$0,00027$
16	131072	0	$11,496$	$0,00054$
32	262144	0	$10,632$	$0,001$
64	524288	0	$9,332$	$0,00175$
1	8192	1	$19,427$	$0,00006$
2	16384	1	$14,425$	$0,00009$
4	32768	1	$10,958$	$0,00013$
8	65536	1	$9,357$	$0,00022$
16	131072	1	$8,824$	$0,00041$
32	262144	1	$8,576$	$0,00081$
64	524288	1	$8,486$	$0,00159$
1	8192	2	$28,083$	$0,00008$
2	16384	2	$13,989$	$0,00008$
4	32768	2	$7,916$	$0,00009$
8	65536	2	$6,001$	$0,00014$
16	131072	2	$5,276$	$0,00025$
32	262144	2	$4,749$	$0,00045$
64	524288	2	$4,485$	$0,00084$
1	8192	3	$26,656$	$0,00008$
2	16384	3	$15,569$	$0,00009$
4	32768	3	$7,942$	$0,00009$
8	65536	3	$5,703$	$0,00013$
16	131072	3	$4,209$	$0,0002$
32	262144	3	$3,472$	$0,00033$
64	524288	3	$3,322$	$0,00062$

Continued on next page

Table C.6 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
1	8192	4	28, 717	0, 00009
2	16384	4	20, 981	0, 00012
4	32768	4	10, 83	0, 00013
8	65536	4	7, 284	0, 00017
16	131072	4	5, 59	0, 00026
32	262144	4	4, 615	0, 00043
64	524288	4	3, 481	0, 00065
1	8192	5	38, 603	0, 00011
2	16384	5	20, 861	0, 00012
4	32768	5	11, 388	0, 00013
8	65536	5	8, 219	0, 00019
16	131072	5	5, 498	0, 00026
32	262144	5	4, 095	0, 00039
64	524288	5	3, 225	0, 00061
1	8192	6	41, 196	0, 00012
2	16384	6	21, 919	0, 00013
4	32768	6	12, 36	0, 00015
8	65536	6	8, 323	0, 0002
16	131072	6	4, 811	0, 00023
32	262144	6	3, 956	0, 00037
64	524288	6	3, 054	0, 00057
1	8192	7	50, 592	0, 00015
2	16384	7	24, 555	0, 00014
4	32768	7	13, 016	0, 00015
8	65536	7	7, 224	0, 00017
16	131072	7	4, 837	0, 00023
32	262144	7	3, 407	0, 00032
64	524288	7	2, 773	0, 00052
1	8192	8	57, 056	0, 00017
2	16384	8	24, 032	0, 00014
4	32768	8	13, 039	0, 00015

Continued on next page

Table C.6 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
8	65536	8	7,196	0,00017
16	131072	8	4,674	0,00022
32	262144	8	3,374	0,00032
64	524288	8	2,817	0,00053
1	8192	9	60,215	0,00018
2	16384	9	30,75	0,00018
4	32768	9	15,779	0,00019
8	65536	9	9,059	0,00021
16	131072	9	5,842	0,00027
32	262144	9	4,369	0,00041
64	524288	9	3,686	0,00069
1	8192	10	74,138	0,00022
2	16384	10	39,939	0,00024
4	32768	10	18,974	0,00022
8	65536	10	9,028	0,00021
16	131072	10	5,916	0,00028
32	262144	10	3,855	0,00036
64	524288	10	3,097	0,00058
1	8192	11	58,93	0,00017
2	16384	11	31,502	0,00019
4	32768	11	16,355	0,00019
8	65536	11	9,275	0,00022
16	131072	11	5,481	0,00026
32	262144	11	3,788	0,00036
64	524288	11	2,996	0,00056
1	8192	12	64,286	0,00019
2	16384	12	33,937	0,0002
4	32768	12	18,006	0,00021
8	65536	12	10,871	0,00026
16	131072	12	6,049	0,00028
32	262144	12	4,135	0,00039

Continued on next page

Table C.6 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
64	524288	12	$3,075$	$0,00058$
1	8192	13	$72,235$	$0,00021$
2	16384	13	$37,597$	$0,00022$
4	32768	13	$19,374$	$0,00023$
8	65536	13	$11,421$	$0,00027$
16	131072	13	$7,091$	$0,00033$
32	262144	13	$5,464$	$0,00051$
64	524288	13	$4,27$	$0,0008$

bigmem

`AMD64; Nehalem (206e6); 2010 Intel Xeon X7560; 32 x 2266MHz`

Table C.7: Performance measurements for **iSHAKE 128** with different configurations and a fixed block size of 8192 bytes (8 KiB), **including** the block header.

blocks	total bytes	threads	cycles/byte	wall time
1	8192	0	$29,229$	$0,00011$
2	16384	0	$28,559$	$0,00021$
4	32768	0	$28,417$	$0,00041$
8	65536	0	$28,594$	$0,00083$
16	131072	0	$28,799$	$0,00167$
32	262144	0	$28,668$	$0,00332$
64	524288	0	$28,987$	$0,00672$
1	8192	1	$89,766$	$0,00033$
2	16384	1	$58,776$	$0,00043$
4	32768	1	$43,165$	$0,00063$
8	65536	1	$34,661$	$0,00101$
16	131072	1	$31,007$	$0,0018$
32	262144	1	$30,125$	$0,00349$
64	524288	1	$27,914$	$0,00647$
1	8192	2	$83,987$	$0,0003$

Continued on next page

Table C.7 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
2	16384	2	$55,3$	$0,0004$
4	32768	2	$35,594$	$0,00052$
8	65536	2	$21,371$	$0,00062$
16	131072	2	$19,43$	$0,00113$
32	262144	2	$17,368$	$0,00201$
64	524288	2	$15,85$	$0,00368$
1	8192	3	$89,756$	$0,00033$
2	16384	3	$50,92$	$0,00037$
4	32768	3	$44,08$	$0,00064$
8	65536	3	$20,012$	$0,00058$
16	131072	3	$13,44$	$0,00078$
32	262144	3	$11,548$	$0,00134$
64	524288	3	$10,534$	$0,00244$
1	8192	4	$92,304$	$0,00033$
2	16384	4	$39,148$	$0,00028$
4	32768	4	$23,31$	$0,00034$
8	65536	4	$15,598$	$0,00045$
16	131072	4	$12,224$	$0,00071$
32	262144	4	$9,394$	$0,00109$
64	524288	4	$10,237$	$0,00237$
1	8192	5	$97,623$	$0,00035$
2	16384	5	$48,434$	$0,00035$
4	32768	5	$26,101$	$0,00038$
8	65536	5	$16,393$	$0,00048$
16	131072	5	$11,118$	$0,00064$
32	262144	5	$8,96$	$0,00104$
64	524288	5	$8,905$	$0,00207$
1	8192	6	$121,867$	$0,00044$
2	16384	6	$57,198$	$0,00042$
4	32768	6	$31,307$	$0,00045$
8	65536	6	$19,588$	$0,00057$

Continued on next page

Table C.7 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
16	131072	6	$11,338$	$0,00066$
32	262144	6	$9,596$	$0,00111$
64	524288	6	$6,626$	$0,00154$
1	8192	7	$138,573$	$0,0005$
2	16384	7	$63,084$	$0,00046$
4	32768	7	$31,847$	$0,00046$
8	65536	7	$20,718$	$0,0006$
16	131072	7	$12,596$	$0,00073$
32	262144	7	$8,349$	$0,00097$
64	524288	7	$6,338$	$0,00147$
1	8192	8	$143,224$	$0,00052$
2	16384	8	$79,999$	$0,00058$
4	32768	8	$37,76$	$0,00055$
8	65536	8	$21,348$	$0,00062$
16	131072	8	$12,541$	$0,00073$
32	262144	8	$9,166$	$0,00106$
64	524288	8	$5,784$	$0,00134$
1	8192	9	$469,506$	$0,0017$
2	16384	9	$217,667$	$0,00158$
4	32768	9	$122,623$	$0,00178$
8	65536	9	$61,78$	$0,00179$
16	131072	9	$29,555$	$0,00171$
32	262144	9	$19,217$	$0,00223$
64	524288	9	$9,817$	$0,00228$
1	8192	10	$524,895$	$0,0019$
2	16384	10	$246,581$	$0,00179$
4	32768	10	$117,896$	$0,00171$
8	65536	10	$59,237$	$0,00172$
16	131072	10	$30,33$	$0,00176$
32	262144	10	$17,8$	$0,00206$
64	524288	10	$10,803$	$0,00251$

Continued on next page

Table C.7 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
1	8192	11	492, 951	0, 00179
2	16384	11	248, 64	0, 0018
4	32768	11	129, 264	0, 00187
8	65536	11	59, 13	0, 00171
16	131072	11	35, 054	0, 00203
32	262144	11	18, 308	0, 00212
64	524288	11	11, 75	0, 00273
1	8192	12	484, 063	0, 00175
2	16384	12	276, 596	0, 002
4	32768	12	123, 083	0, 00178
8	65536	12	62, 741	0, 00182
16	131072	12	32, 126	0, 00186
32	262144	12	19, 865	0, 0023
64	524288	12	10, 537	0, 00244
1	8192	13	526, 782	0, 00191
2	16384	13	250, 106	0, 00181
4	32768	13	96, 3	0, 0014
8	65536	13	65, 471	0, 0019
16	131072	13	35, 295	0, 00205
32	262144	13	20, 021	0, 00232
64	524288	13	13, 069	0, 00303

Table C.8: Performance measurements for **iSHAKE 128** with different configurations and a fixed block size of 8192 bytes (8 KiB), **excluding** the block header.

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
1	8192	0	28, 422	0, 0001
2	16384	0	28, 628	0, 00021
4	32768	0	28, 285	0, 00041
8	65536	0	28, 614	0, 00083

Continued on next page

Table C.8 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
16	131072	0	29, 033	0, 00168
32	262144	0	29, 077	0, 00337
64	524288	0	28, 999	0, 00673
1	8192	1	87, 932	0, 00032
2	16384	1	58, 8	0, 00043
4	32768	1	43, 547	0, 00063
8	65536	1	36, 569	0, 00106
16	131072	1	31, 273	0, 00181
32	262144	1	28, 368	0, 00329
64	524288	1	28, 13	0, 00652
1	8192	2	86, 422	0, 00031
2	16384	2	45, 238	0, 00033
4	32768	2	42, 462	0, 00062
8	65536	2	18, 981	0, 00055
16	131072	2	30, 845	0, 00179
32	262144	2	20, 335	0, 00236
64	524288	2	15, 494	0, 00359
1	8192	3	93, 024	0, 00034
2	16384	3	61, 451	0, 00045
4	32768	3	28, 603	0, 00041
8	65536	3	24, 652	0, 00071
16	131072	3	19, 625	0, 00114
32	262144	3	23, 204	0, 00269
64	524288	3	11, 366	0, 00264
1	8192	4	92, 472	0, 00034
2	16384	4	48, 22	0, 00035
4	32768	4	24, 159	0, 00035
8	65536	4	16, 168	0, 00047
16	131072	4	13, 627	0, 00079
32	262144	4	9, 761	0, 00113
64	524288	4	8, 094	0, 00188

Continued on next page

Table C.8 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
1	8192	5	99, 421	0, 00036
2	16384	5	43, 012	0, 00031
4	32768	5	25, 47	0, 00037
8	65536	5	14, 648	0, 00043
16	131072	5	10, 918	0, 00063
32	262144	5	8, 592	0, 001
64	524288	5	7, 048	0, 00163
1	8192	6	411, 368	0, 00149
2	16384	6	223, 716	0, 00162
4	32768	6	106, 573	0, 00155
8	65536	6	56, 873	0, 00165
16	131072	6	30, 265	0, 00176
32	262144	6	17, 83	0, 00207
64	524288	6	11, 213	0, 0026
1	8192	7	456, 75	0, 00166
2	16384	7	198, 515	0, 00144
4	32768	7	105, 965	0, 00154
8	65536	7	58, 873	0, 00171
16	131072	7	31, 483	0, 00183
32	262144	7	17, 966	0, 00208
64	524288	7	11, 996	0, 00278
1	8192	8	498, 047	0, 00181
2	16384	8	231, 011	0, 00167
4	32768	8	109, 866	0, 00159
8	65536	8	61, 743	0, 00179
16	131072	8	33, 29	0, 00193
32	262144	8	18, 957	0, 0022
64	524288	8	10, 8	0, 0025
1	8192	9	468, 419	0, 0017
2	16384	9	240, 483	0, 00174
4	32768	9	121, 924	0, 00177

Continued on next page

Table C.8 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
8	65536	9	59, 833	0, 00173
16	131072	9	27, 72	0, 00161
32	262144	9	18, 205	0, 00211
64	524288	9	11, 368	0, 00264
1	8192	10	734, 45	0, 00266
2	16384	10	355, 304	0, 00258
4	32768	10	147, 487	0, 00214
8	65536	10	91, 926	0, 00267
16	131072	10	47, 687	0, 00277
32	262144	10	26, 496	0, 00307
64	524288	10	15, 596	0, 00362
1	8192	11	768, 357	0, 00278
2	16384	11	397, 019	0, 00288
4	32768	11	183, 576	0, 00266
8	65536	11	94, 718	0, 00275
16	131072	11	49, 611	0, 00288
32	262144	11	28, 368	0, 00329
64	524288	11	15, 351	0, 00356
1	8192	12	764, 029	0, 00277
2	16384	12	352, 14	0, 00255
4	32768	12	190, 034	0, 00275
8	65536	12	95, 802	0, 00278
16	131072	12	59, 823	0, 00347
32	262144	12	25, 621	0, 00297
64	524288	12	16, 201	0, 00376
1	8192	13	970, 83	0, 00352
2	16384	13	512, 628	0, 00372
4	32768	13	227, 928	0, 0033
8	65536	13	129, 791	0, 00376
16	131072	13	69, 338	0, 00402
32	262144	13	33, 437	0, 00388

Continued on next page

Table C.8 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
64	524288	13	$18,661$	$0,00433$

Table C.9: Performance measurements for **iSHAKE 256** with different configurations and a fixed block size of 8192 bytes (8 KiB), **including** the block header.

blocks	total bytes	threads	cycles/byte	wall time
1	8192	0	$37,205$	$0,00013$
2	16384	0	$37,05$	$0,00027$
4	32768	0	$36,891$	$0,00053$
8	65536	0	$37,097$	$0,00108$
16	131072	0	$37,351$	$0,00217$
32	262144	0	$37,293$	$0,00432$
64	524288	0	$37,371$	$0,00867$
1	8192	1	$101,506$	$0,00037$
2	16384	1	$67,057$	$0,00049$
4	32768	1	$52,315$	$0,00076$
8	65536	1	$44,863$	$0,0013$
16	131072	1	$39,512$	$0,00229$
32	262144	1	$37,077$	$0,0043$
64	524288	1	$35,927$	$0,00833$
1	8192	2	$94,357$	$0,00034$
2	16384	2	$51,023$	$0,00037$
4	32768	2	$51,095$	$0,00074$
8	65536	2	$43,793$	$0,00127$
16	131072	2	$29,924$	$0,00174$
32	262144	2	$31,81$	$0,00369$
64	524288	2	$19,745$	$0,00458$
1	8192	3	$96,422$	$0,00035$
2	16384	3	$52,869$	$0,00038$
4	32768	3	$35,434$	$0,00051$

Continued on next page

Table C.9 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
8	65536	3	$25,663$	$0,00074$
16	131072	3	$22,643$	$0,00131$
32	262144	3	$15,109$	$0,00175$
64	524288	3	$14,285$	$0,00331$
1	8192	4	$100,922$	$0,00037$
2	16384	4	$52,284$	$0,00038$
4	32768	4	$36,136$	$0,00052$
8	65536	4	$26,284$	$0,00076$
16	131072	4	$14,204$	$0,00082$
32	262144	4	$11,923$	$0,00138$
64	524288	4	$10,367$	$0,0024$
1	8192	5	$93,427$	$0,00034$
2	16384	5	$58,128$	$0,00042$
4	32768	5	$29,654$	$0,00043$
8	65536	5	$21,745$	$0,00063$
16	131072	5	$16,104$	$0,00093$
32	262144	5	$10,802$	$0,00125$
64	524288	5	$8,854$	$0,00205$
1	8192	6	$114,999$	$0,00042$
2	16384	6	$67,556$	$0,00049$
4	32768	6	$36,882$	$0,00054$
8	65536	6	$21,563$	$0,00063$
16	131072	6	$15,309$	$0,00089$
32	262144	6	$9,758$	$0,00113$
64	524288	6	$7,974$	$0,00185$
1	8192	7	$143,705$	$0,00052$
2	16384	7	$68,658$	$0,0005$
4	32768	7	$41,405$	$0,0006$
8	65536	7	$20,516$	$0,0006$
16	131072	7	$13,953$	$0,00081$
32	262144	7	$9,872$	$0,00115$

Continued on next page

Table C.9 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
64	524288	7	7, 376	0, 00171
1	8192	8	167, 44	0, 00061
2	16384	8	76, 654	0, 00056
4	32768	8	37, 235	0, 00054
8	65536	8	21, 466	0, 00062
16	131072	8	13, 39	0, 00078
32	262144	8	9, 292	0, 00108
64	524288	8	6, 664	0, 00155
1	8192	9	183, 672	0, 00067
2	16384	9	92, 294	0, 00067
4	32768	9	42, 062	0, 00061
8	65536	9	23, 862	0, 00069
16	131072	9	15, 171	0, 00088
32	262144	9	9, 298	0, 00108
64	524288	9	7, 474	0, 00173
1	8192	10	205, 677	0, 00075
2	16384	10	100, 861	0, 00073
4	32768	10	41, 985	0, 00061
8	65536	10	28, 108	0, 00082
16	131072	10	15, 484	0, 0009
32	262144	10	12, 838	0, 00149
64	524288	10	6, 575	0, 00153
1	8192	11	232, 17	0, 00084
2	16384	11	119, 235	0, 00086
4	32768	11	49, 777	0, 00072
8	65536	11	32, 233	0, 00093
16	131072	11	18, 074	0, 00105
32	262144	11	10, 596	0, 00123
64	524288	11	8, 553	0, 00198
1	8192	12	475, 669	0, 00172
2	16384	12	248, 697	0, 0018

Continued on next page

Table C.9 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
4	32768	12	132, 389	0, 00192
8	65536	12	68, 574	0, 00199
16	131072	12	37, 949	0, 0022
32	262144	12	20, 793	0, 00241
64	524288	12	11, 82	0, 00274
1	8192	13	613, 048	0, 00222
2	16384	13	253, 259	0, 00184
4	32768	13	146, 518	0, 00212
8	65536	13	50, 642	0, 00147
16	131072	13	41, 983	0, 00243
32	262144	13	22, 107	0, 00256
64	524288	13	12, 085	0, 0028

Table C.10: Performance measurements for **iSHAKE 256** with different configurations and a fixed block size of 8192 bytes (8 KiB), **excluding** the block header.

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
1	8192	0	36, 96	0, 00013
2	16384	0	36, 928	0, 00027
4	32768	0	36, 836	0, 00053
8	65536	0	36, 646	0, 00106
16	131072	0	36, 768	0, 00213
32	262144	0	36, 831	0, 00427
64	524288	0	36, 716	0, 00851
1	8192	1	103, 955	0, 00038
2	16384	1	66, 627	0, 00048
4	32768	1	52, 354	0, 00076
8	65536	1	44, 385	0, 00129
16	131072	1	39, 191	0, 00227
32	262144	1	37, 855	0, 00439

Continued on next page

Table C.10 – continued from previous page

blocks	**total bytes**	**threads**	**cycles/byte**	**wall time**
64	524288	1	36,019	0,00835
1	8192	2	97,829	0,00035
2	16384	2	51,946	0,00038
4	32768	2	53,022	0,00077
8	65536	2	43,912	0,00127
16	131072	2	22,537	0,00131
32	262144	2	21,564	0,0025
64	524288	2	19,626	0,00455
1	8192	3	94,4	0,00034
2	16384	3	57,31	0,00042
4	32768	3	34,197	0,0005
8	65536	3	24,59	0,00071
16	131072	3	29,635	0,00172
32	262144	3	14,496	0,00168
64	524288	3	14,988	0,00348
1	8192	4	85,656	0,00031
2	16384	4	45,868	0,00033
4	32768	4	35,222	0,00051
8	65536	4	17,108	0,0005
16	131072	4	15,966	0,00093
32	262144	4	14,127	0,00164
64	524288	4	13,467	0,00312
1	8192	5	96,674	0,00035
2	16384	5	56,023	0,00041
4	32768	5	26,755	0,00039
8	65536	5	21,63	0,00063
16	131072	5	14,267	0,00083
32	262144	5	13,704	0,00159
64	524288	5	9,539	0,00221
1	8192	6	111,392	0,0004
2	16384	6	61,868	0,00045

Continued on next page

Table C.10 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
4	32768	6	$35,679$	$0,00052$
8	65536	6	$22,543$	$0,00065$
16	131072	6	$13,568$	$0,00079$
32	262144	6	$11,395$	$0,00132$
64	524288	6	$7,909$	$0,00183$
1	8192	7	$136,687$	$0,0005$
2	16384	7	$75,356$	$0,00055$
4	32768	7	$38,489$	$0,00056$
8	65536	7	$19,894$	$0,00058$
16	131072	7	$14,035$	$0,00081$
32	262144	7	$12,545$	$0,00145$
64	524288	7	$7,853$	$0,00182$
1	8192	8	$141,972$	$0,00051$
2	16384	8	$76,495$	$0,00055$
4	32768	8	$38,907$	$0,00056$
8	65536	8	$23,596$	$0,00068$
16	131072	8	$13,881$	$0,00081$
32	262144	8	$10,898$	$0,00126$
64	524288	8	$6,7$	$0,00155$
1	8192	9	$172,404$	$0,00063$
2	16384	9	$88,157$	$0,00064$
4	32768	9	$42,85$	$0,00062$
8	65536	9	$23,184$	$0,00067$
16	131072	9	$15,347$	$0,00089$
32	262144	9	$9,554$	$0,00111$
64	524288	9	$6,704$	$0,00156$
1	8192	10	$468,928$	$0,0017$
2	16384	10	$243,014$	$0,00176$
4	32768	10	$115,936$	$0,00168$
8	65536	10	$54,121$	$0,00157$
16	131072	10	$32,379$	$0,00188$

Continued on next page

Table C.10 – continued from previous page

blocks	total bytes	threads	cycles/byte	wall time
32	262144	10	19, 114	0, 00222
64	524288	10	11, 279	0, 00262
1	8192	11	508, 409	0, 00184
2	16384	11	274, 688	0, 00199
4	32768	11	97, 175	0, 00141
8	65536	11	61, 38	0, 00178
16	131072	11	31, 303	0, 00182
32	262144	11	19, 412	0, 00225
64	524288	11	12, 538	0, 00291
1	8192	12	524, 559	0, 0019
2	16384	12	269, 17	0, 00195
4	32768	12	119, 405	0, 00173
8	65536	12	64, 805	0, 00188
16	131072	12	36, 757	0, 00213
32	262144	12	20, 454	0, 00237
64	524288	12	13, 761	0, 00319
1	8192	13	515, 4	0, 00187
2	16384	13	289, 856	0, 0021
4	32768	13	124, 705	0, 00181
8	65536	13	66, 514	0, 00193
16	131072	13	36, 184	0, 0021
32	262144	13	21, 498	0, 00249
64	524288	13	13, 041	0, 00302

C.3 ParallelHash evaluation

nepenthe

`AMD64; Haswell/Crystalwell (0306C3h); 2014 Intel Core i7 4980HQ; 4x2800MHz`

Table C.11: Performance measurements for **ParallelHash 128** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation.

bytes	**cycles**	**cycles/byte**
1625	13036	$8,022$
2048	15358	$7,499$
2580	18314	$7,098$
3251	22232	$6,839$
4096	27190	$6,638$
5161	32946	$6,384$
6502	40776	$6,271$
8192	49356	$6,025$
10321	61934	$6,001$
13004	77336	$5,947$
16384	95314	$5,818$
20643	121130	$5,868$
26008	148852	$5,723$
32768	184062	$5,617$
41285	233802	$5,663$
52016	293844	$5,649$
65536	365728	$5,581$
82570	499937	$6,055$
104032	630156	$6,057$
131072	748924	$5,714$
165140	925024	$5,601$
208064	1258436	$6,048$
262144	1578525	$6,022$
330281	1990620	$6,027$
416128	2506891	$6,024$
524288	2840172	$5,417$

Table C.12: Performance measurements for **ParallelHash 256** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation.

bytes	**cycles**	**cycles/byte**
1625	15251	$9,385$
2048	18352	$8,961$
2580	21290	$8,252$
3251	24956	$7,676$
4096	31406	$7,667$
5161	37938	$7,351$
6502	52553	$8,083$
8192	60858	$7,429$
10321	77388	$7,498$
13004	96436	$7,416$
16384	115362	$7,041$
20643	146392	$7,092$
26008	189672	$7,293$
32768	233290	$7,119$
41285	282942	$6,853$
52016	362748	$6,974$
65536	453244	$6,916$
82570	558350	$6,762$
104032	721844	$6,939$
131072	882496	$6,733$
165140	1112986	$6,74$
208064	1407912	$6,767$
262144	1810230	$6,905$
330281	2266610	$6,863$
416128	2876304	$6,912$
524288	3919630	$7,476$

Table C.13: Performance measurements for **ParallelHash 128** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation, featuring *lane complementing.*

bytes	**cycles**	**cycles/byte**
1625	13431	8, 265
2048	19370	9, 458
2580	23005	8, 917
3251	25412	7, 817
4096	30950	7, 556
5161	34614	6, 707
6502	39522	6, 078
8192	49072	5, 99
10321	63270	6, 13
13004	78840	6, 063
16384	100640	6, 143
20643	127118	6, 158
26008	157514	6, 056
32768	193436	5, 903
41285	243444	5, 897
52016	315598	6, 067
65536	385456	5, 882
82570	486480	5, 892
104032	626574	6, 023
131072	770488	5, 878
165140	994666	6, 023
208064	1251802	6, 016
262144	1578364	6, 021
330281	1938644	5, 87
416128	2501184	6, 011
524288	3074050	5, 863

Table C.14: Performance measurements for **ParallelHash 256** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation, featuring *lane complementing*.

bytes	**cycles**	**cycles/byte**
1625	15709	$9,667$
2048	17898	$8,739$
2580	20792	$8,059$
3251	25486	$7,839$
4096	32088	$7,834$
5161	38764	$7,511$
6502	48172	$7,409$
8192	60536	$7,39$
10321	77052	$7,466$
13004	95946	$7,378$
16384	125618	$7,667$
20643	152466	$7,386$
26008	200750	$7,719$
32768	239706	$7,315$
41285	302120	$7,318$
52016	381438	$7,333$
65536	478320	$7,299$
82570	603862	$7,313$
104032	758628	$7,292$
131072	955674	$7,291$
165140	1203966	$7,291$
208064	1516310	$7,288$
262144	1910794	$7,289$
330281	2406896	$7,287$
416128	3029420	$7,28$
524288	3815476	$7,277$

bigmem

`AMD64; Nehalem (206e6); 2010 Intel Xeon X7560; 32 x 2266MHz`

Table C.15: Performance measurements for **ParallelHash 128** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation.

bytes	cycles	cycles/byte
1625	24247	14, 921
2048	29540	14, 424
2580	34697	13, 448
3251	41783	12, 852
4096	50566	12, 345
5161	61234	11, 865
6502	75452	11, 604
8192	93237	11, 381
10321	117654	11, 399
13004	145896	11, 219
16384	186045	11, 355
20643	233736	11, 323
26008	291774	11, 219
32768	367458	11, 214
41285	462425	11, 201
52016	584211	11, 231
65536	731323	11, 159
82570	924831	11, 201
104032	1159445	11, 145
131072	1460779	11, 145
165140	1842004	11, 154
208064	2314646	11, 125
262144	2916593	11, 126
330281	3673312	11, 122
416128	4624589	11, 113
524288	5844027	11, 147

Table C.16: Performance measurements for **ParallelHash 256** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation.

bytes	**cycles**	**cycles/byte**
1625	27727	17,063
2048	34767	16,976
2580	40041	15,52
3251	48770	15,002
4096	61206	14,943
5161	73446	14,231
6502	91069	14,006
8192	113974	13,913
10321	145758	14,122
13004	180928	13,913
16384	233747	14,267
20643	293074	14,197
26008	366409	14,088
32768	458694	13,998
41285	577677	13,992
52016	732054	14,074
65536	911727	13,912
82570	1153815	13,974
104032	1447003	13,909
131072	1818688	13,875
165140	2294419	13,894
208064	2881789	13,85
262144	3635362	13,868
330281	4578420	13,862
416128	5768391	13,862
524288	7263717	13,854

Table C.17: Performance measurements for **ParallelHash 128** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation, featuring *lane complementing*.

bytes	**cycles**	**cycles/byte**
1625	21604	$13,295$
2048	26412	$12,896$
2580	31147	$12,072$
3251	37507	$11,537$
4096	45665	$11,149$
5161	55086	$10,674$
6502	68040	$10,464$
8192	83785	$10,228$
10321	105851	$10,256$
13004	131470	$10,11$
16384	168416	$10,279$
20643	211525	$10,247$
26008	263809	$10,143$
32768	333449	$10,176$
41285	419758	$10,167$
52016	529867	$10,187$
65536	663958	$10,131$
82570	837417	$10,142$
104032	1052648	$10,119$
131072	1327232	$10,126$
165140	1670885	$10,118$
208064	2096867	$10,078$
262144	2643296	$10,083$
330281	3328642	$10,078$
416128	4190177	$10,069$
524288	5284901	$10,08$

Table C.18: Performance measurements for **ParallelHash 256** with a block size of 8192 bytes (8 KiB) and a variable size input. Generically optimised 64-bit implementation, featuring *lane complementing.*

bytes	**cycles**	**cycles/byte**
1625	24823	$15,276$
2048	31090	$15,181$
2580	35975	$13,944$
3251	43886	$13,499$
4096	55174	$13,47$
5161	66190	$12,825$
6502	82147	$12,634$
8192	102924	$12,564$
10321	131727	$12,763$
13004	163551	$12,577$
16384	209525	$12,788$
20643	261959	$12,69$
26008	327114	$12,577$
32768	413250	$12,611$
41285	519888	$12,593$
52016	656690	$12,625$
65536	825322	$12,593$
82570	1042564	$12,626$
104032	1307597	$12,569$
131072	1646640	$12,563$
165140	2074586	$12,563$
208064	2609123	$12,54$
262144	3290191	$12,551$
330281	4144787	$12,549$
416128	5219482	$12,543$
524288	6578617	$12,548$