Specification and verification of a hashing device

MITB Project Report Number MITB.1.0

(DRAFT: last modified on December 3, 2013)

Mike Gordon, Robert Künnemann, Graham Steel[1]

Contents

[1]Listed in alphabetical order.

Specification and verification of a hashing device

Mike Gordon, Robert Künnemann, Graham Steel[†]

December 3, 2013

Abstract

MITB ("MAC In The Box") is a standalone device that computes a MAC[1] using the KECCAK sponge function[2]. This document describes both a specification of MITB based on a design by Robert Künnemann and also a high-level implementation as a state-machine. The implementation has been verified to meet the specification by mechanised formal proof. The MITB project was proposed by Graham Steel as part of a collaboration between Cambridge and INRIA to investigate the design, implementation and formal verification of security devices, such as cryptographic tokens.

1 Introduction

MITB is a device with two 1-bit control inputs `skip_inp`, `move_inp`, two data inputs `block_inp` and `size_inp`, a 1-bit control output `ready_out` and a data output `digest_out`.

```
                    |-----------------------------------|
               1    |                                   |  1
  skip_inp ---/-->|                                   |---/--> ready_out
                    |                                   |
               1    |                                   |
  move_inp ---/-->|                                   |
                    |          MITB (r,c,n) f           |
               r    |                                   |
  block_inp --/-->|                                   |
                    |                                   |
         ⌈lg2r⌉    |                                   |  n
  size_inp ---/-->|                                   |---/--> digest_out
                    |                                   |
                    |-----------------------------------|
```

MITB is parametrised on three numbers (r, c, n) and a permutation function f. These are part of the KECCAK specification (see Section 3). An actual device would be manufactured with specific values for the parameters.

The input `block_inp` is r-bits wide and the output `digest_out` is n-bits wide. The input `size_inp` has sufficient bits to represent a number of size r or less. For convenience it is modelled as a number rather than a bitstring.

MITB runs continuously after being switched on . It is implemented as a state-machine using combinational logic and registers (see Section 5). All a user can observe (assuming tamper-resistant manufacture) are the sequences of values appearing on the outputs `ready_out` and `digest_out`, which depend on the values input via `skip_inp`, `move_inp`, `block_inp` and `size_inp`.

From a user's point of view MITB can be in either of two states: *ready* or *absorbing*. It powers up into state *ready*. The 1-bit output `ready_out` indicates whether the state is *ready* (`T` output) or *absorbing* (`F` output).[3]

[†]Listed in alphabetical order.

[1]`http://en.wikipedia.org/wiki/Message_authentication_code`.

[2]KECCAK: `http://keccak.noekeon.org/`; SHA-3: `http://en.wikipedia.org/wiki/SHA-3)`.

[3]The boolean truth-values `T` and `F` are used to model bits 1 and 0, respectively.

The input `skip_inp` 'freezes' MITB: holding it `T` stops the state changing on successive cycles. The input `move_inp` causes the state to change on the next cycle; in particular it is used to signal that MITB should start absorbing a message.

The MAC of a message M is specified as the KECCAK hash of the result of concatenating the secret key onto the front of the message, i. e. the hash of $key\|M$, where $\|$ denotes bitstring concatenation. The Hash algorithm is defined in Section 3.2. The protocol for using MITB to compute the MAC of a message is described below. The main correctness property of the device is that if the specified protocol is used to input a message then its MAC will appear on `digest_out`. The main security property is that no matter what inputs are supplied, the secret key cannot be revealed. These properties will be expressed as constraints on what sequences of inputs and outputs are possible using a temporal logic notation.

MITB has a permanent memory for holding an r-bit secret key. The key can be set or changed by holding both `skip_inp` and `move_inp` `F` in the *ready* state. The data being input on `block_inp` then overwrites the stored key. As long a `skip_inp` and `move_inp` are held `F`, the stored key is updated on each cycle (discussed in Section 6).

MITB is ready to compute the MAC of a message in state *ready*. The protocol for computing the MAC of M is as follows ($|B|$ denotes the number of bits in B):

1. The user splits M into a sequence of blocks, $M = B_1\|B_2\|\cdots\|B_{m-1}\|B_m$, such that all blocks except the last one are r-bits wide, i. e. $|B_i| = r$ for $1 \leq i < m$ and $|B_m| < r$. If r divides exactly into $|M|$, then B_n is taken to be the empty block (so $|B_m| = 0$).

2. When `ready_out` is `T` the user puts MITB into the *absorbing* state by inputting `F` on `skip_inp` and `T` on `move_inp` (`block_inp` and `size_inp` are ignored during this step).

3. Starting on the next cycle, and continuing for m cycles, the user inputs `F` on both `move_inp` and `skip_inp`, B_i on `block_inp` and $|B_i|$ on `size_inp`, where $1 \leq i \leq m$. During this time `F` will be output on `ready_out`.

4. After inputting B_m, the user keeps inputting `F` on `skip_inp` and `move_inp` until `ready_out` becomes `T`. On the cycle when this happens the hash of $key\|M$ will appear on `digest_out`. The number of cycles taken depends on $|B_m|$. If $|B_m| \neq r-1$ then `ready_out` will become `T` on the cycle after B_m is input. If $|B_m| = r-1$ then `ready_out` will become `T` the cycle after the cycle after B_m is input.

The timing diagrams below illustrate this protocol. The MAC computation starts at cycle t and `X` means 'don't care' (if `skip_inp` is `T` then other inputs are ignored). The first line shows the cycle count. The message M is split by the user into blocks B1,. . .,Bn: $M =$ B1$\|\cdots\|$Bm. When MITB is started the volatile memory holds zeros, so `digest_out` is all zeros too.

If the size of the last block Bm is not $r-1$ (i. e. $|M|$ `MOD` $r \neq r-1$).

Cycles:	0	1	2	...	t	t+1	t+2	...	t+m	t+(m+1)
skip_inp	T	T	T	...	F	F	F	...	F	X
move_inp	X	X	X	...	T	F	F	...	F	X
block_inp	X	X	X	...	X	B1	B2	...	Bm	X
size_inp	X	X	X	...	X	r	r	...	|Bm|	X
ready_out	T	T	T	...	T	F	F	...	F	T
digest_out	0	0	0	...	0	0	0	...	0	Hash($key\|M$)

If the last block has size $r-1$.

Cycles:	0	1	2	...	t	t+1	t+2	...	t+m	t+(m+1)	t+(m+2)
skip_inp	T	T	T	...	F	F	F	...	F	F	F
move_inp	X	X	X	...	T	F	F	...	F	F	F
block_inp	X	X	X	...	X	B1	B2	...	Bn	X	X
size_inp	X	X	X	...	X	r	r	...	r-1	X	X
ready_out	T	T	T	...	T	F	F	...	F	F	T
digest_out	0	0	0	...	0	0	0	...	0	0	Hash($key\|M$)

Once the MAC, Hash($key\|M$), has been computed, it can be forced to persist on the output `digest_out` by holding `T` on `skip_inp`.

The security property that MITB guarantees is that no matter what inputs are supplied, the value output on `digest_out` is always either 0 (i. e. an n-bit bitstring representing 0) or the hash of some message. Assuming KECCAK is secure, then MITB does not reveal any information about the stored key.

2 Formal specification using temporal logic

An implementation of MITB is modelled with a next-state function `MITB` (defined in Section 5) that gives the next state s' when input i is received in state s. The observable outputs are determined by the current state, so the model is a Moore machine. MITB is parametrised on the KECCAK parameters (r, c, n) and permutation function f, so the next-state function `MITB` takes these as arguments, hence $s' = \texttt{MITB}\ (r, c, n)\ f\ (i, s)$.[4]

A user of MITB can supply inputs on `skip_inp`, `move_inp`, `block_inp` and `size_inp` and observe the resulting outputs on `ready_out` and `digest_out`, where `ready_out` is a boolean value showing which state the device is in and `digest_out` is an n-bit word consisting of zeros in the *absorbing* state and the bottom n bits of the volatile memory in the *ready* state.

Using a standard method [10], the implementation of MITB will be represented by a formula:

$$\begin{array}{l}\texttt{MITB\_IMP}\ key\ (r, c, n)\ f \\ \quad (\texttt{cntl\_sig}, \texttt{pmem\_sig}, \texttt{vmem\_sig}) \\ \quad (\texttt{skip\_inp}, \texttt{move\_inp}, \texttt{block\_inp}, \texttt{size\_inp}, \texttt{ready\_out}, \texttt{digest\_out})\end{array}$$

where `MITB_IMP` is a predicate defined in terms of `MITB`.

The 6-tuple (`skip_inp`, `move_inp`, `block_inp`, `size_inp`, `ready_out`, `digest_out`) has components that are functions from time to values representing the sequence of values on the four user-supplied inputs and two device-generated outputs. The triple (`cntl_sig`, `pmem_sig`, `vmem_sig`) contains the function `cntl_sig` representing a sequence of values that the control register can take, `pmem_sig` representing values of the permanent memory and `vmem_sig` representing the values of the volatile memory. The notation $\texttt{MITB\_IMP}\ (r, c, n)\ f \models \phi$ means that all runs of MITB satisfy the property ϕ.

$$\texttt{MITB\_IMP}\ (r, c, n)\ f \models \phi \Leftrightarrow \forall \sigma_1\ \sigma_2.\ \texttt{MITB\_IMP}\ (r, c, n)\ f\ \sigma_1\ \sigma_2 \Rightarrow \phi(\sigma_1, \sigma_2)$$

where the quantified variable σ_1 ranges of triples of state functions and σ_2 ranges over 6-tuples of port functions. Properties ϕ are expressed as Linear Temporal Logic (LTL) formulas. The specification given in Section 4 consists of a conjunction of temporal logic formulas that all executions of MITB are required to satisfy.

[4]Following standard λ-calculus notation, functions may be curried and brackets around arguments omitted.

2.1 Atomic formulas

The *atomic formulas* skipInp(b), moveInp(b), blockInp(bs), sizeInp(len), readyOut(b), digestOut(bs) are true of an execution σ if and only if the argument is the value of the corresponding input or output at cycle 0 of σ. Here b ranges over bits (modelled as truth-values), bs ranges over bitstrings (modelled as lists of truth-values) and len ranges over natural numbers. For example, `MITB_IMP` (r, c, n) f $\models$ readyOut `T` specifies that any execution of MITB must output `T` on `ready_out` during the first cycle.

Non-atomic properties specify relationships between values input and output at times other than the first cycle; for this the following temporal operators are used.

2.2 Temporal specification operators

The operators below are mostly standard concepts from LTL, though sometimes with different names from those commonly used. Syntactically these operators combine temporal formulas ϕ, ϕ_1, ϕ_2 etc. Semantically they are predicates on pairs of tuples of functions as in $\phi(\sigma_1, \sigma_2)$ as occurring in the definition of `MITB_IMP` (r, c, n) f $\models$ ϕ.

The first table defines 'lifted' logical operators that are useful for combining temporal formulas built using the temporal operator in the later tables.

Lifted logical operators

`Bool` b	$=$	$\lambda\sigma.\ b$
`Not` ϕ	$=$	$\lambda\sigma.\ \neg(\phi\ \sigma)$
$(\phi_1$ `And` $\phi_2)$	$=$	$\lambda\sigma.\ (\phi_1\ \sigma) \wedge (\phi_2\ \sigma)$
$(\phi_1$ `Or` $\phi_2)$	$=$	$\lambda\sigma.\ (\phi_1\ \sigma) \vee (\phi_2\ \sigma)$
$(\phi_1$ `Implies` $\phi_2)$	$=$	$\lambda\sigma.\ (\phi_1\ \sigma) \Rightarrow (\phi_2\ \sigma)$
(`Forall` $x.\ \phi$)	$=$	$\lambda\sigma.\ \forall x.\ \phi\ \sigma$
(`Exists` $x.\ \phi$)	$=$	$\lambda\sigma.\ \exists x.\ \phi\ \sigma$

If π is a function representing a sequence of values, then $\pi{\downarrow}n = \lambda t.\ \pi(t + n)$, which is π with n elements chopped off the front. If σ is a tuple of functions, then $\sigma{\downarrow}n$ is defined recursively through its tuple structure: $(\sigma_1, \ldots, \sigma_m){\downarrow}n = (\sigma_1{\downarrow}n, \ldots, \sigma_m{\downarrow}n)$. `Next` (see below) uses this.

Standard LTL operators

`Next` ϕ	$=$	$\lambda\sigma.\ \phi(\sigma{\downarrow}1)$
`Always` ϕ	$=$	$\lambda\sigma.\ \forall t.\ \phi(\sigma{\downarrow}t)$
`Sometime` ϕ	$=$	$\lambda\sigma.\ \exists t.\ \phi(\sigma{\downarrow}t)$
ϕ_1 `Until` ϕ_2	$=$	$\lambda\sigma.\ \exists t_1.\ \phi_2(\sigma{\downarrow}t_1) \wedge \forall t_2.\ t_2 < t_1 \Rightarrow \phi_1(\sigma{\downarrow}t_2)$

An additional operator `UntilN`(t), parametrised on a number t, is defined recursively in terms of the standard operators above: ϕ_1 `UntilN`(t) ϕ_2 is like ϕ_1 `Until` ϕ_2, but with t specifying the exact number of cycles taken to reach a state in which ϕ_2 is true.

Cycle counting `Until` operator

ϕ_1 `UntilN`(0) ϕ_2	$=$	ϕ_2
ϕ_1 `UntilN`$(t{+}1)$ ϕ_2	$=$	ϕ_1 `And Next`$(\phi_1$ `UntilN`(t) $\phi_2)$

It is easy to show by induction on t that ϕ_1 `Until` ϕ_2 $=$ `Exists` $t.$ ϕ_1 `UntilN`(t) ϕ_2.

3 The Keccak sponge function

In this section relevant parts of the Keccak algorithm are described and then formalised (see Section 3.2). In order to make it easy to check that the Keccak algorithm is being correctly formalised, extracts from the specification will be cut-and-pasted from the official Keccak reference document [1]. These, and other imported extracts, are included in boxes such as:

> Keccak (pronounced [kɛtʃak]) is a family of sponge functions [8] that use as a building block a permutation from a set of 7 permutations. In this chapter, we introduce our conventions and notation, specify the 7 permutations underlying Keccak and the Keccak sponge functions. We also give conventions for naming parts of the Keccak state.

Note that the citation "[8]" in the box above refers to the citations in the Keccak reference document, not to the references at the end of this paper.

Keccak is based on a 'sponge construction' in which an arbitrary length message is iteratively 'absorbed' into a finite state. The number of iterations depending on the length of the message. Once all of the message has been absorbed, the resulting state can be 'squeezed' to extract a digest. For the general Keccak algorithm this squeezing can also iterative; the number of iterations depending on the desired length of the digest. However, for the SHA-3 application, the digest length is such that no squeeze iterations are actually needed (see also Section 3.2.4).

Keccak is a family of algorithms. Each algorithm in the family corresponds to a choice of values for parameters r, called the *bitrate* and c, called the *capacity*. Keccak$[r, c]$ denotes the specific algorithm for the indicated parameter values. SHA-3 recognises four instances of Keccak, namely Keccak$[1152, 448]$, Keccak$[1088, 512]$, Keccak$[832, 768]$, Keccak$[576, 1024]$. For each of these instances, a length n of digest is specified, namely: 224, 256, 384, 512, respectively. $\lfloor \text{Keccak}[r, c] \rfloor_n$ denotes Keccak$[r, c]$ with a digest of length n. Thus:

$$(r, c, n) \in \{ (1152, 448, 224), (1088, 512, 256), (832, 768, 384), (576, 1024, 512) \}$$

SHA-3 recommends using the strongest parameter values; smaller values of the parameters are for testing and, possibly, lightweight hashing. The state of the SHA-3 sponge algorithm thus consists of $r + c = 1600$ bits, which is called the *width* and denoted by b. Note that $b = 25 \times 2^6$. More generally, the width b is defined to be 25×2^l, where l is another parameter that can take on one of the seven values in $\{0, 1, 2, 3, 4, 5, 6\}$. The seven choices for l correspond to the "7 permutations" mentioned in the box above. The SHA-3 instance of Keccak fixes the value of l to be 6. The initial value of the state before a message has been absorbed by the sponge algorithm is 0^b, i.e. each of the 1600 state bits is 0.

3.1 The sponge algorithm

Keccak is based on a function $f : \mathbb{Z}_2^b \rightarrow \mathbb{Z}_2^b$ that permutes the state, where $\mathbb{Z}_2 = \{0, 1\}$ is the set of the two bits 0 and 1, and $\mathbb{Z}_2^b$ is the set of bitstrings of length b ($b = 1600$ for SHA-3).

The sponge algorithm applies the state permutation f on each iteration of absorbing a message M into the state. The absorption algorithm is outlined below. A detailed formal specification of f is not given, but an informal description can be found in Appendix A. For the specification and proofs here f is treated as an uninterpreted parameter. This is discussed in Section 6.

3.1.1 Padding

The algorithm is parametrised on numbers r and c, as discussed above. The first step is to pad the message so that it can be split into an exact number, k say, of blocks of length r. This is done by appending "a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length". If the message length is already a multiple of the block length r, then it is still padded by appending $10^{r-2}1$, where 0^p denotes a bitstring of length p with each bit being 0. If the message needs only one bit added to make its length a multiple of r, then $10^{r-1}1$, which has length $r+1$, is appended. Thus padding appends "at least 2 bits and at most the number of bits in a block plus one".

In the KECCAK reference, the result of padding a message M using a block size x is denoted by $M||\mathrm{pad}[x](|M|)$ and the specific padding described above is called *multi-rate* padding and denoted by pad10*1. Here is the actual text from the reference:

> For the padding rule we use the following notation: the padding of a message M to a sequence of x-bit blocks is denoted by $M||\mathrm{pad}[x](|M|)$. This notation highlights that we only consider padding rules that append a bitstring that is fully determined by the bitlength of M and the block length x. We may omit $[x]$, $(|M|)$ or both if their value is clear from the context.
>
> KECCAK makes use of the *multi-rate* padding.
>
> **Definition 1.** Multi-rate padding, *denoted by* pad10*1, *appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length.*
>
> Multi-rate padding appends at least 2 bits and at most the number of bits in a block plus one.

3.1.2 Absorbing

The absorption algorithm takes a message M as input and returns a digest h. It consists of three steps: padding the input, iteratively computing a sequence of b-bit states $s_0 \ldots s_m$ ($b = 1600$), extracting the digest from the final state s_m. In more details the three steps are:

1. Apply padding to M. Let the resulting blocks be $B_1, \ldots, B_m$; each of length r.
2. $s_0 = 0^b$ and for $i = 1, \ldots, m$ iteratively compute $s_i \leftarrow f(s_{i-1} \oplus (Bi||0^{b-r}))$, where $||$ is bitstring concatenation, $\oplus$ is bitwise XOR and f is the permutation function.
3. Output $h = \lfloor s_m \rfloor_n$, where $\lfloor s_m \rfloor_n$ is the first n bits of s_m and it is guaranteed that $n < b$ as $b = 1600$ and $n \in \{224, 256, 384, 512\}$.

The Keccak-reference [1] description of the algorithm is in the box below, where the parameters for the sponge construction are the permutation f of width b, a padding rule "pad" and the bitrate $r < b$. The input is M and the output (h in step 3 of the pseudo-code above) is $\lfloor Z \rfloor_l$.

The individual blocks are named P_i in the box below, rather than Bi as above.

Algorithm 1 The sponge construction SPONGE$[f, \text{pad}, r]$

```
Require: $r < b$

  Interface: $Z = \text{sponge}(M, \ell)$ with $M \in \mathbb{Z}_2^*$, integer $\ell > 0$ and $Z \in \mathbb{Z}_2^\ell$
  $P = M||\text{pad}[r](|M|)$
  $s = 0^b$
  for $i = 0$ to $|P|_r - 1$ do
    $s = s \oplus (P_i||0^{b-r})$
    $s = f(s)$
  end for
  $Z = \lfloor s \rfloor_r$
  while $|Z|_r r < \ell$ do
    $s = f(s)$
    $Z = Z||\lfloor s \rfloor_r$
  end while
  return $\lfloor Z \rfloor_\ell$
```

3.2 Padding and absorbing

`MITB` (r, c, n) f is designed to compute `Hash` (r, c, n) f s_0 $(key\|M)$, where s_0 is the initial state, M is the message whose MAC is required and `Hash` is the KECCAK hash function:

`Hash` (r, c, n) f s m $=$ `Squeeze` n (`Absorb` f c s (`Split` r (`Pad` r m)))

The function `Hash` is defined for arbitrary values of the parameters r, c, n, f and arbitrary states s and bitstrings m (representing messages). The auxiliary functions `Pad`, `Split`, `Absorb` and `Squeeze` are defined below. The notation $[b_1, \ldots, b_n]$ denotes a bitstring consisting of the n bits $b_1, \ldots, b_n$. In particular [T] is the bitstring consisting of exactly on bit (T representing 0). The function `Zeros` u maps a number to a bitstring consisting of u zeros, e. g. `Zeros` $3 = [\mathtt{F}, \mathtt{F}, \mathtt{F}]$. `Zeros` u is another notation for 0^u.

3.2.1 `Pad`

`Pad` r m pads bitstring m, using the KECCAK rules described in Section 3.1.1. It can be concisely defined by:

`Pad` r m $= m\|[\mathtt{T}]\|\mathtt{Zeros}((r - (|m| + 2)\ \mathtt{MOD}\ r)\ \mathtt{MOD}\ r)\|[\mathtt{T}]$

Whilst it is clear that `Pad` r m appends a bitstring $[T]\|\mathtt{Zeros}(x)\|[T]$ to m, it may not be obvious that x should be $(r - (|m| + 2)\ \mathtt{MOD}\ r)\ \mathtt{MOD}\ r$. However, for all $r > 1$:

$(r - (|m| + 2)\ \mathtt{MOD}\ r)\ \mathtt{MOD}\ r\ =$
  `if` $|m|\ \mathtt{MOD}\ r = r{-}1$ `then` $r{-}1$ `else` $r - (|m|\ \mathtt{MOD}\ r) - 2$

so the complicated formula x is correct.

3.2.2 `Split`

`Split` $r\ m$ splits m into a list of blocks of length r, except for the last one, which has length $|m|$ `MOD` r. This has already been described in Section 1. To define it formally the list-processing functions `Cons`, `Take` and `Drop` are used.

`Cons` $e\ l$ adds an element e to the front of list l, for example `Cons` $0\ [1,2,3] = [0,1,2,3]$. `Take` $u\ l$ returns the first u elements of l, for example `Take` $3\ [0,1,2,3,4,5] = [0,1,2]$. `Take` $u\ l$ is the same as the $\lfloor l \rfloor_u$, as used in Section 3.1.2. `Drop` $u\ l$ removes the first u elements of l, for example `Drop` $3\ [0,1,2,3,4,5] = [3,4,5]$. If $u \leq |l|$ then $|$`Take` $u\ l| = u$, $|$`Drop` $u\ l| = |l| - u$ and `Take` $u\ l \| $`Drop` $u\ l = l$.

The function `Split` is then defined recursively by:

$$\texttt{Split}\ r\ m\ =\ \texttt{if}\ (r=0) \vee |m| \leq r\ \texttt{then}\ [m]\ \texttt{else}\ \texttt{Cons}\ (\texttt{Take}\ r\ m)\ (\texttt{Split}\ r\ (\texttt{Drop}\ r\ m))$$

It is straightforward to verify that, if $r > 0$, then the result of concatenating all the blocks in `Split` $r\ m$ is m, that all blocks in `Split` $r\ m$, except the last one, have size r and that the last block in `Split` $r\ m$ has size $|m|$ `MOD` r.

3.2.3 `Absorb`

`Absorb` $f\ c\ s\ bkl$ absorbs the blocks in a list of blocks bkl starting from a state s as described in Section 3.1.2. It is defined recursively on bkl by:

$$\texttt{Absorb}\ f\ c\ s\ [\,]\ =\ s$$
$$\texttt{Absorb}\ f\ c\ s\ (\texttt{Cons}\ bk\ bkl)\ =\ \texttt{Absorb}\ f\ c\ (f(s \oplus (bk \| \texttt{Zeros}\ c)))\ bkl$$

An equivalent alternative definition of `Absorb` uses the standard iteration combinator `Foldl` that is widely used in functional programming:

$$\texttt{Absorb}\ f\ c\ =\ \texttt{Foldl}\ (\lambda s\ bk.\ f(s \oplus (bk \| \texttt{Zeros} c)))$$

where: `Foldl` $fn\ e\ [\,] = e$ and `Foldl` $fn\ e\ (\texttt{Cons}\ x\ l) = \texttt{Foldl}\ fn\ (fn\ e\ x)\ l$.

3.2.4 `Squeeze`

An unusual feature of the general KECCAK hash algorithm is that it can generate output digests of arbitrary length using an iterative 'squeezing' algorithm. However, for SHA-3 the recommended digest size n is 224. This is specified to be the bottom n bits in the final state after absorbing the message. Thus: `Squeeze` $n\ s$ = `Take` $n\ s$ or just `Squeeze` = `Take`.

3.3 MACs based on Keccak

The KECCAK designers claim that secure message authentication codes (MACS) can be computed by simply hashing the result of concatenating a key onto the front of a message. The KECCAK specification [1] says:

> Unlike SHA-1 and SHA-2, Keccak does not have the length-extension weakness, hence does not need the HMAC nested construction. Instead, MAC computation can be performed by simply prepending the message with the key.

This KECCAK MAC of message M using key key is `Hash` $(r,c,n)\ f\ (\texttt{Zeros}(r{+}c))\ (key\|M)$.

4 Formal specification of MITB

The specification of MITB requires that `MITB` (r, c, n) $f \models \phi$, where ϕ is a conjunction of properties. The particular properties in the conjunction are given mnemonic names and explained and defined separately in the boxes below. Some of these properties need to refer to the secret key *key* which is stored as $f(key \| \texttt{Zeros}(c))$ in MITB's permanent memory. The atomic property pmemState is used to state such properties: `MITB` (r, c, n) $f \models \mathsf{pmemState}(s)$ is true if and only if MITB is storing s in its permanent memory.

4.1 Initialisation: `Init`

The property `Init` specifies that the 'power up' state of MITB is *ready*, that zeros are being output on `digest_out` and that $f(key \| \texttt{Zeros}(c))$ is stored in permanent memory.

```
Init key c n f = readyOut(T) And digestOut(Zeros n) And pmemState(f(key||Zeros(c)))
```

4.2 Freezing the state: `Freeze`

The property `Freeze` specifies that the state and outputs of MITB remains unchanged as long as `T` is input on `skip_inp`.

```
Freeze =
 Always
  (Forall s b1 b2
    skipInp(T) And pmemState(s) And readyOut(b1) And digestOut(b2)
    Implies
    Next(pmemState(s) And readyOut(b1) And digestOut(b2)))
```

4.3 Resetting: `Reset`

The property `Reset` specifies that inputting `T` on moveInp and `F` on skipInp in an *absorbing* state, i. e. when `ready_out` is `F`, results in a return to the *ready* state on the next cycle, with permanent memory unchanged and `Zeros` n being output at `ready_out`.

```
Reset n =
 Always
  (Forall s.
    (moveInp(T) And skipInp(F) And readyOut(F) And pmemState(s))
    Implies
    Next(readyOut(T) And pmemState(s) And digestOut(Zeros n)))
```

4.4 Installing a new key: `KeyUpdate`

The property `KeyUpdate` entails that when in state *ready*, inputting `F` on both `skip_inp` and `move_inp` and inputting *key* (where $|key| = r$) on `block_inp`, results in $f(key\|\texttt{Zeros}(c))$ being stored in the permanent memory and then remaining in the *ready* state on the next cycle.

```
KeyUpdate $r\ c\ f$ =
 Always
  ((readyOut(T) And skipInp(F) And moveInp(F))
   Implies
   (Forall $key$.
     blockInp($key$) And Bool($|key| = r$)
     Implies Next(readyOut(T) And pmemState($f(key\|\texttt{Zeros}(c))$)))))
```

The key installed by `KeyUpdate` overwrites the existing one in the permanent memory.

4.5 Computing a MAC: `ComputeMAC`

The definition of `ComputeMAC` below specifies that if the user follows the correct protocol for inputting a message then its MAC is computed.

The user is required to split the message into blocks and then input them and their lengths.

An individual block *bk* is input by putting it on `block_inp` and its size $|bk|$ on `size_inp` whilst holding both `skip_inp` and `move_inp` at `F`. This is represented by the temporal formula `InputBlock` *bk* defined by:

```
InputBlock $bk$ = blockInp($bk$) And sizeInp $|bk|$ And skipInp(F) And moveInp(F)
```

The recursively defined property `InputBlocks` $r\ [bk_1, \ldots, bk_m]$ uses `InputBlock` and specifies the procedure for inputting the sequence of blocks $bk_1, \ldots, bk_m$. It is a temporal formula which the user must ensure holds. The complexity in the definition is (i) to ensure that an extra empty block is added when the last block has size r and (ii) to drive the device for an extra cycle when the size of the last block is $r-1$.

```
InputBlocks $r$ [ ] = Bool(F)        (This case should not arise in practice.)
InputBlocks $r$ (Cons $bk$ $bkl$) =
  if $bkl$ = [ ]
    then if $|bk| = r$
            then InputBlock $r$ $bk$ And Next(InputBlock $r$ [ ])
            else if $|bk| = r - 1$
                   then InputBlock $r$ $bk$ And Next(skipInp(F) And moveInp(F))
                   else InputBlock $r$ $bk$
    else InputBlock $r$ $bk$ And Next(InputBlocks $r$ $bkl$) :
```

In the definition of `ComputeMAC` in the box below, the lines are numbered for use in the detailed explanation given after the definition.

```
1  ComputeMAC $(r, c, n)$ $f$ =
2  Always
3   (Forall $key$ $m$.
4     (readyOut(T) And skipInp(F) And moveInp(T)
5      And Bool($|key| = r$) And pmemState($f(key \| \mathsf{Zeros}(c))$))
6     Implies
7     Next
8      (InputBlocks $r$ (Split $r$ $m$)
9       Implies
10      (readyOut(F) And digestOut(Zeros $n$))
11      UntilN(if $|m|$ MOD $r = r{-}1$ then $|m|$ DIV $r + 2$ else $|m|$ DIV $r + 1$)
12      (readyOut(T) And digestOut(MAC $key$ $(r, c, n)$ $m$) And pmemState($f(key \| \mathsf{Zeros}(c))$))))
```

The definition of `ComputeMAC` should be compared to the informal description on page 2. The following detailed description of the formula refers to the lines in the box above.

Line 1: The property `ComputeMAC` is parametrised on the Keccak parameters (r, c, n) and a permutation function f (see Section 3).

Line 2: `Always` specifies that the property holds at all times; without this `ComputeMAC` would only be true at time 0.

Line 3: The `Forall` quantification is inside the scope of the enclosing `Always` and so is over the key *key* and message m at the time the transaction holds.

Line 4: A MAC computation can only be started when MITB is in state *ready*. To start the computation in this state the user inputs `F` on input `skip_inp` and`T` on input `move_inp`. This causes MITB to go into the *absorbing* state on the next cycle.

Line 5: The key *key* used for the computation is assumed to be of size r (i. e. 1152 bits for the SHA-3 recommended instance of Keccak); $f(key \| \mathsf{Zeros}(c))$ is assumed to be stored in the permanent memory of the device (see Section 4.4).

Lines 6-7: The body of the `Forall` is of the form '*precondition* `Implies Next` *absorb*' which should be read 'if *precondition* holds then on the next cycle *absorb* holds', where *absorb* has the form '*input* `Implies` *invariant* `UntilN`(*number−of−steps*) *result*'. and occupies lines 8-12.

Line 8: The message is split into blocks and these are input on successive cycles.

Line 10: During the computation `ready_out` shows `F` and 0s are output on `digest_out`.

Line 11: The computation takes $|m|$ `DIV` $r + 2$ steps if $|m|$ `MOD` r is $r{-}1$, otherwise it takes $|m|$ `DIV` $r + 1$ steps.

Line 12: The output `ready_out` changes to `T` and the MAC appears on `digest_out`. The value in the permanent memory is unchanged from the start of the computation.

4.6 All reachable state are secure: `Secure`

The Property `Secure` entails that in all reachable states the value output at digestOut is either `Zeros` n or `Hash` $(r,c,n)\ f\ m$ for some message m.

$$\texttt{Secure}\ (r,c,n)\ f\ =\ \texttt{Always}(\mathsf{digestOut}(\texttt{Zeros}\ n)\ \texttt{Or}\ \texttt{Exists}\ m.\ \mathsf{digestOut}(\texttt{Hash}\ (r,c,n)\ f\ (\texttt{Zeros}(r{+}c))\ m))$$

4.7 Complete specification

The specification of MITB is the conjunction of the properties defined in the preceding sections.

$$\begin{array}{lll} (\texttt{MITB}\ (r,c,n)\ f, s_0) \models & \texttt{Init}\ \textit{key}\ c\ n\ f & \texttt{And} \\ & \texttt{Freeze} & \texttt{And} \\ & \texttt{Reset}\ n & \texttt{And} \\ & \texttt{KeyUpdate}\ r\ c\ f & \texttt{And} \\ & \texttt{ComputeMAC}\ (r,c,n)\ f & \texttt{And} \\ & \texttt{Secure}\ (r,c,n)\ f & \end{array}$$

Whether or not this is either a correct or sufficient specification is discussed in Section 6.

5 Implementation

Recall from Section 2 that for any temporal formula ϕ:

$$\texttt{MITB\_IMP}\ (r,c,n)\ f\ \models\ \phi\ \Leftrightarrow\ \forall \sigma_1\ \sigma_2.\ \texttt{MITB\_IMP}\ (r,c,n)\ f\ \sigma_1\ \sigma_2 \Rightarrow \phi(\sigma_1,\sigma_2)$$

where `MITB_IMP` is a predicate defined in terms of a next-state function `MITB` representing a Moore machine and parametrised on the Keccak parameters (r,c,n) and the Keccak permutation function f.

This section describes a concrete definition of `MITB_IMP` that has been proved to implement the specification given in Section 4.7.[5] The description is presented in two stages: first a curried function `MITB_FUN` specifies the behaviour abstractly, and then `MITB_FUN` is refined to `MITB`, which is then used to define `MITB_IMP` that models a high level register transfer level (RTL) implementation. A diagram of the states and transitions of `MITB_FUN` is on page 14.

5.1 Behavioural specification: `MITB_FUN`

`MITB_FUN` takes an abstract state, which is a triple (`cntl`, `pmem`, `vmem`), and an input i (elaborated below) and returns the next state (`cntl′`, `pmem′`, `vmem′`). The first component, `cntl`, can have one of three values: `Ready`, `Absorbing` and `AbsorbEnd`. `Ready` corresponds to the *ready* state described in Section 1 and both `Absorbing` and `AbsorbEnd` correspond to the *absorbing* state. The second and third components of an abstract state, `pmem` and `vmem`, are bit-strings of length $r{+}c$ and represent the values of the permanent and volatile memory.

[5] See `http://www.cl.cam.ac.uk/~mjcg/MITB/` for details.

An input i can either be `Move`, `Skip` or `Input` bk len, where bk is a bitstring of size r and len is the number of bits of bk that constitutes the block being input (thus $len \leq r$). The bitstring bk and number len represent values being input on `block_inp` and `size_inp`.

The definition of `MITB_FUN`,and other functions that follow, have been cut-and-pasted from the files input to the HOL4 proof assistant[6] used for the verification and then lightly edited to improve readability and to make them compatible with the notation used elsewhere in this document. The definition of `MITB_FUN` uses ML-style pattern matching, so there are separate equations for the various combinations of values of `cntl` and the input i. These equation are numbered for easy reference (the numbers are not in the original source).

```
1:  (MITB_FUN (r,c,n) f (cntl,pmem,vmem) Skip =
     (cntl,pmem,vmem))
    ∧
2:  (MITB_FUN (r,c,n) f (Ready,pmem,vmem) (Input key len) =
     (Ready,f(key ∥ Zeros c),Zeros(r+c)))
    ∧
3:  (MITB_FUN (r,c,n) f (Ready,pmem,vmem) Move =
     (Absorbing,pmem,pmem))
    ∧
4:  (MITB_FUN (r,c,n) f (Absorbing,pmem,vmem) Move =
     (Ready,pmem,Zeros(r+c)))
    ∧
5:  (MITB_FUN (r,c,n) f (Absorbing,pmem,vmem) (Input blk len) =
     if len ≤ r-2 then
       (Ready,pmem,
        f(vmem ⊕ (Take len blk ∥ [T] ∥ Zeros((r-len)-2) ∥ [T] ∥ Zeros c)))
     else if len = r-1 then
       (AbsorbEnd,pmem,f (vmem ⊕ (Take len blk ∥ [T] ∥ Zeros c)))
     else (Absorbing,pmem,f(vmem ⊕ (blk ∥ Zeros c))))
    ∧
6:  (MITB_FUN (r,c,n) f (AbsorbEnd,pmem,vmem) Move =
     (Ready,pmem,Zeros(r+c)))
    ∧
7:  (MITB_FUN (r,c,n) f (AbsorbEnd,pmem,vmem) (Input blk len) =
     (Ready,pmem,f (vmem ⊕ (Zeros(r-1) ∥ [T] ∥ Zeros c))))
```

Equation `1` says that if `Skip` is input, then the state stays the same. Equations `2` and `3` describe what happens in the *ready* state (i. e. `cntl` = `Ready`). If the input is `Input` *key len* then the permanent memory `pmem` is set to $f(key \| \texttt{Zeros}\ c)$, the volatile memory `vmem` is set to `Zeros`$(r{+}c)$, and the state remains *ready.* If the input is `Move` then the next state is *absorbing* (`cntl` = `Absorbing`) with the permanent memory unchanged and the volatile memory set to the value of the permanent memory. Equations `4` and `6` specify that if `Move` is input whilst *absorbing*, then the volatile memory is reset to zeros and the device returns to the *ready* state.

The most complex equation is `5`, which specifies the state transition corresponding to absorbing a block. What happens depends on whether the block is the last one, which is signalled by the input length being less than r and corresponds to the first two branches of the conditional. The complexity here is because the devices does the padding, as described in Section 3.1.1. If the last block is one bit short of being a full block of length r (len = $r{-}1$) then one bit

[6]`http://hol.sourceforge.net/`

is added and the device enters the sub-state of *absorbing* with `cntl` = `AbsorbEnd`, then on the next cycle, described in equation 7, the remaining padding (i. e. $r-1$ zeros and a final `T`) is added and the permutation f applied before transitioning back to the *ready* state. The final `else`-clause in equation 5 specifies the absorption of a non-final block, as described in Section 3.1.2. Such a block must have size exactly r and is absorbed by: (i) appending c zeros to it, (ii) then XOR-ing the result with the current value, `vmem`, of the volatile memory, then (iii) applying the KECCAK permutation f to the result of the XOR-ing, and finally (iv) the volatile memory is updated to the result of this application of f.

Here is an overview of `MITB_FUN` in the form of an ASCII art state transition diagram.

```
                   |-----------|                 |-----------|
                   |           |       Move      |           |
                   |           |<--------------->|           |
               +---|           |                 |           |---+
               |   |           |                 |           |   |
Input key len  |   |  Ready    |                 | Absorbing |   | Input blk len
               |   |           |   Input blk len |           |   |   (len = r)
               |-->|           |    (len < r-1)  |           |<--|
                   |           |<----------------|           |
                   |           |                 |           |
                   |-----+-----|                 |-----------|
                         ↑                             |
                         |                             | Input blk len
                         |                             |   (len = r-1)
                         |                             ↓
                         |                       |-----+-----|
                         |                       |           |
                         |                       |           |
                         |                       |           |
                         |                       |           |
                         |-----------------------| AbsorbEnd |
                                                 |           |
                                                 |           |
                                                 |           |
                                                 |           |
                                                 |-----+-----|
```

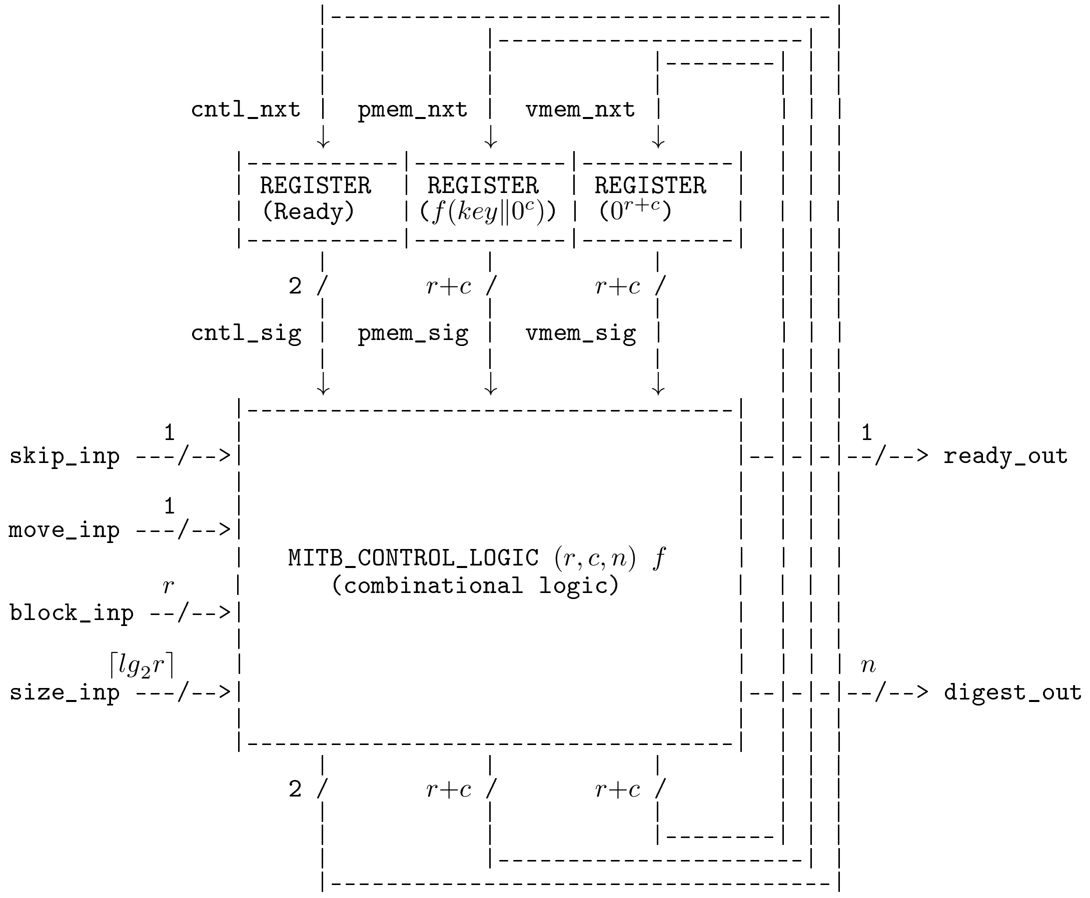
The function `MITB` is similar `MITB_FUN` except that it decodes the inputs into abstract commands `Skip`, `Move` and `Input` *bk len*.

```
MITB (r,c,n) f ((skip,move,block,size),(cntl,pmem,vmem)) =
 MITB_FUN (r,c,n) f (cntl,pmem,vmem)
  (if skip = [T] then Skip
   else if move = [T] then Move
   else Input block size)
```

5.2 Register transfer behaviour and structure: `MITB` and `MITB_DEV`

To make a concrete device based on the behaviour specified in `MITB`, a hardware structure implementing the state transitions needs to be designed and verified. This will consist of registers for storing the state (`cntl`, `pmem`, `vmem`) – probably using different memory technologies for `pmem` and for `cntl` and `vmem` – and a specification of the control logic.

The diagram that follows on page 15 shows such a design, albeit one that is still quite abstract and missing many details (see Section 6 for some discussion). In this diagram the datapaths are labelled with the names used in the formal specification given later (e.g. `cntl_sig`, `cntl_nxt`) and a number followed by "\" indicates the width of the labelled datapath in bits. Note, however, that in the model the values on `cntl_sig`, `cntl_nxt` and `size_inp` have not been coded as bitstring and, for simplicity, are kept abstract. The `cntl` component holds one of three values `Ready`, `Absorbing` or `AbsorbEnd`, so two bit are sufficient to encode these. The input `size_inp` is a number less than or equal to r, so can be encoded in $\lceil lg_2 r \rceil$ bits.



This diagram is expressed in logic in a standard way using relations.[7] Melham's book is a comprehensive reference [10]. The definition of `MITB` below uses the relation `REGISTER` to model registers as a unit delay:

```
REGISTER init_state (inp,out) ⇔
 (out 0 = init_state)
 ∧
 ∀t. out (t+1) = inp t
```

Three instances of `REGISTER` are used in the definition of `MITB`: to store `cntl`, `pmem` and `vmem`. Their initial values are indicated in brackets (using 0^c to abbreviate `Zeros` c and 0^{r+c} to abbreviate `Zeros`$(r+c)$).

The relation `MITB_CONTROL_LOGIC` packages up `MITB` as a relation and adds some logic to drive the outputs `ready_out` and `digest_out`.

[7]See `http://www.cl.cam.ac.uk/~mjcg/WhyHOL.pdf`.

```
MITB_CONTROL_LOGIC (r,c,n) f
 (cntl_sig,pmem_sig,vmem_sig,skip_inp,move_inp,block_inp,size_inp,
  cntl_nxt,pmem_nxt,vmem_nxt,ready_out,digest_out)
 ⇔
 (∀t.
   (cntl_nxt t,pmem_nxt t,vmem_nxt t) =
   MITB (r,c,n) f
     ((skip_inp t,move_inp t,block_inp t,size_inp t),cntl_sig t,
      pmem_sig t,vmem_sig t))
 ∧
 (∀t. ready_out t = [cntl_sig t = Ready])
 ∧
 (∀t. digest_out t =
      if cntl_sig t = Ready then Take n (vmem_sig t) else Zeros n)
```

The diagram on page 15 shows widths of the various datapaths. In an HDL like Verilog these would be expressed in the type system, but here the predicate `WIDTH` is used.

```
Width sig n ⇔ ∀t. |sig t| = n
```

Also the verification requires the KECCAK parameters to satisfy $2 < r$, $0 < c$ and $n \leq r$, which is clearly satisfied by the particular values used by SHA-3. This constraint is enforced using the predicate `GoodParameters`.

```
GoodParameters (r,c,n) ⇔ 2 < r ∧ 0 < c ∧ n ≤ r
```

The implementation `MITB_IMP` combines the registers for holding the various state components with the combinational logic `MITB_CONTROL_LOGIC`.

```
MITB_IMP key (r,c,n) f
  (cntl_sig,pmem_sig,vmem_sig)
  (skip_inp,move_inp,block_inp,size_inp,ready_out,digest_out) ⇔
 ∃cntl_nxt pmem_nxt vmem_nxt.
  GoodParameters (r,c,n) ∧ (∀s. |f s| = |s| ∧
  (|key| = r) ∧ Width pmem_sig (r+c) ∧
  Width vmem_sig (r+c) ∧ Width pmem_nxt (r+c) ∧
  Width vmem_nxt (r+c) ∧ Width skip_inp 1 ∧ Width move_inp 1 ∧
  Width block_inp r ∧ (∀t. size_inp t ≤ r) ∧
  Width ready_out 1 ∧ Width digest_out n ∧
  REGISTER Ready (cntl_nxt,cntl_sig) ∧
  REGISTER (f(key‖Zeros c)) (pmem_nxt,pmem_sig) ∧
  REGISTER (Zeros(r+c)) (vmem_nxt,vmem_sig) ∧
  MITB_CONTROL_LOGIC (r,c,n) f
    (cntl_sig,pmem_sig,vmem_sig,skip_inp,move_inp,block_inp,
     size_inp,cntl_nxt,pmem_nxt,vmem_nxt,ready_out,digest_out)
```

A non-vacuity theorem verifying that every sequence of inputs on `skip_inp`, `move_inp`, `block_inp` and `size_inp` determines sequences of outputs on `ready_out` and `digest_out` has been proved:

```
⊢ ∀key r c n f skip_inp move_inp block_inp size_inp.
    (∀s. (|f s| = |s|) ∧ GoodParameters (r,c,n) ∧
    (|key| = r) ∧ Width skip_inp 1 ∧ Width move_inp 1 ∧
    Width block_inp r ∧ (∀t. size_inp t <= r)
    ⇒
    ∃cntl_sig pmem_sig vmem_sig ready_out digest_out.
      MITB_IMP key (r,c,n) f
        (cntl_sig,pmem_sig,vmem_sig)
        (skip_inp,move_inp,block_inp,size_inp,ready_out,digest_out)
```

6 Discussion and further work

The methods used here, which combine temporal logic specifications with mechanised proofs about state machines represented in higher order logic, are not new and date from the 1980s [8, 7, 6, 10, 3]. The contribution of this case study is to show how these old ideas may possibly be useful on a timely example. Although this study is both trivial and incomplete, it is hoped it may be a first step towards something significant.

In the rest of this section the incompleteness of the current work is described. Following that, there is a first partial attempt at a security assessment of the MITB project. Finally, possible future work is outlined.

6.1 Adequacy and incompleteness

The current specification has not been validated as a usable API description. All that has been done is to prove that Künnemann's machine implements it. The list of properties in Section 4 are ad hoc. Whilst they are consistent, since the `MITB` implementation is a model, how can one know if they are adequate? Adding and verifying simple properties can be mostly automatic (though `ComputeMAC` required a complicated interactive proof).

The design of MITB is also ad hoc in that the effects of the various inputs may well be badly designed. For example, having the stored key reset whenever `move_inp` is `F` in the *ready* state may be dangerous (e. g. an attacker could surreptitiously reset the key). Adjusting the API would be easy (e. g. just removing the ability to reset the key).

There are two major omissions in the work so far. The first major omission is that the MITB design is not realistic hardware. The input `block_inp` is 1600-bits wide and the input `size_inp` is a number. Whilst the latter is trivial to fix by encoding the numbers as bitstrings, the former will require sequential buffering, say to accumulate blocks 16-bits at a time over a 100 cycles. Adapting the state machine to do this should be straightforward, and the temporal logic specifications should be easy to adapt to work with the additional data acquisition cycles, and most of the existing proofs should be reusable. The work involved is classical data and temporal abstraction [10]. However, this is all work which has not been done.

The second major omission is that the multi-round KECCAK-f permutation function described in Appendix A has not been implemented. It is treated as an uninterpreted function. This nicely separates the API aspects of MITB from the cryptographic computation concerns, but a complete implementation would need hardware implementing f, which might need several cycles (e.g. one for each sub-round), so also requiring temporal refinement as discussed in the previous paragraph. There is a discussion in Chapter 4 (Hardware) of the KECCAK implementation overview [2]. Although the KECCAK-f is quite complicated (see Appendix A) creating a verified hardware designs implementing it should be straightforward (though possibly a lot of work due to all the intricate details). An approach using verifying synthesis might be appropriate.

6.2 Security assessment

Imagine the MITB design and verification have been completed, say down to synthesisable RTL represented in logic, but resembling a standard HDL like Verilog. What would have been achieved? All that would have been shown is that the API functionality specified in LTL is realised by the HDL model. However:

1. Maybe the verifier is lying about having completed the proof?
   - This could be mitigated by replaying the proof.
2. Maybe the proof tool (HOL4) is unsound?
   - This could be mitigated by using another independent tool (e. g. HOL Light, ProofPower, Isabelle/HOL) together with an expert audit of the tools used, which should have a strong soundness pedigree.
3. How can the design model be securely manufactured?
   - There are many challenges here ranging from unsound synthesis tools, to unsafe implementation technologies [5].
4. What threats does the model ignore?
   - It is assumed KECCAK has good cryptographic properties. Being an NSA approved standard, this may raise 'Snowden worries'. There is no modelling of side-channels or tampering attacks which could extract $f(key\|0^c)$ from the permanent memory.

Cohn [4] give an early discussion of issues related to some of those above that provoked a controversy [9].

Finally, although MITB was conceived as part of a secure password secrecy system, it has not been discussed how such a complete system would work, so there is no analysis of what actual contribution to security an MITB device might make.

6.3 Next steps: short and long term

Completing the design and implementation would seem to be the essential first next step, but maybe it would be better to step back now and decide where the MITB project is going and what is most critical from a total system security perspective.

Careful thinking is needed to evaluate what contribution a formal verification of MITB could make to enhancing the security of using hashing in a real world setting.

References

[1] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak specifications, see `http://keccak.noekeon.org/`, 2009.

[2] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Kleer. Keccak implementation overview, see `http://keccak.noekeon.org/Keccak-implementation-3.2.pdf`, 2012.

[3] C.-T. Chou. Predicates, temporal logic, and simulations. In J. Joyce and C.-J. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 310–323. Springer Berlin Heidelberg, 1994.

[4] A. Cohn. The notion of proof in hardware verification. *J. Autom. Reasoning*, 5(2):127–139, 1989.

[5] S. Drimer. Security for volatile FPGAs. Technical Report UCAM-CL-TR-763, University of Cambridge, Computer Laboratory, Nov. 2009.

[6] J. J. Joyce. Formal specification and verification of asynchronous processes in higher-order logic. Technical Report UCAM-CL-TR-136, University of Cambridge, Computer Laboratory, June 1988.

[7] J. J. Joyce. Totally verified systems: linking verified software to verified hardware. Technical Report UCAM-CL-TR-178, University of Cambridge, Computer Laboratory, Sept. 1989.

[8] J. J. Joyce. Multi-level verification of microprocessor-based systems. Technical Report UCAM-CL-TR-195, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, May 1990.

[9] D. MacKenzie. The fangs of the VIPER. *Nature*, 352:467–468, 1991.

[10] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.

7 Appendix A. The Keccak permutation

The diagram below from the Keccak reference describes the structure of the state.

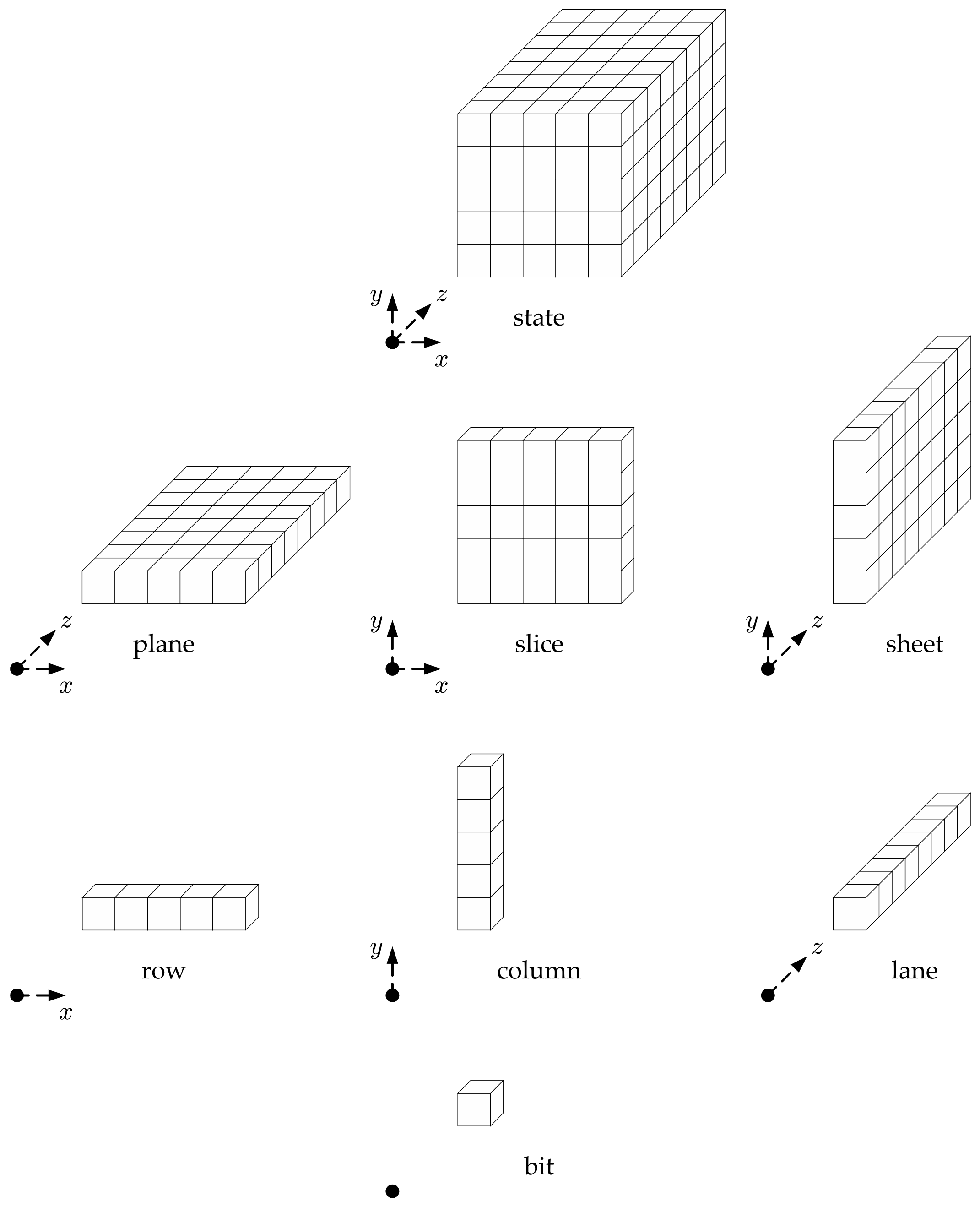


Figure 1.1: Naming conventions for parts of the Keccak-f state

Each cell in the state contains a single bit. The state is structured as a $5{\times}5{\times}w$ three-dimensional array a, where $w = 2^l = 2^6 = 64$ for SHA-3. The state thus contains 1600 bits.

The state variable s, used above when describing the absorption process, is intuitively a 1600-bit word. The array variable a, used below when describing the permutation f, consists of the same number of bits, but thought of as a $5 \times 5 \times w$ three-dimensional array. The mapping between the bits of a and those of s is given by: $a[x][y][z] = s[64(5y + x) + z]$ where $a[x][y][z]$ is the bit at co-ordinates (x, y, z) of a and $s[n]$ is the nth bit of s. The following is from the KECCAK reference:

> The permutation KECCAK-$f[b]$ is described as a sequence of operations on a state a that is a three-dimensional array of elements of GF(2), namely $a[5][5][w]$, with $w = 2^\ell$. The expression $a[x][y][z]$ with $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_w$, denotes the bit in position (x, y, z). It follows that indexing starts from zero. The mapping between the bits of s and those of a is $s[w(5y + x) + z] = a[x][y][z]$. Expressions in the x and y coordinates should be taken modulo 5 and expressions in the z coordinate modulo w. We may sometimes omit the $[z]$ index, both the $[y][z]$ indices or all three indices, implying that the statement is valid for all values of the omitted indices.

The KECCAK permutation f is $12 + 2l$ (i.e. 24) iterations of a function R, called a *round*, where R is the function composition ($\circ$) of five sub-round functions θ, ρ, π, χ, ι, thus:

$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$

The sub-round functions are described in the following sub-sections. The new states resulting from applying the sub-round functions θ, ρ, π, χ and ι to a state a are denoted by θa, ρa, πa, χa and ιa, respectively.

In what follows, the arithmetical operations are interpreted modulo 2 on single-bit state elements $s[n] \in \{0, 1\}$. They are interpreted modulo 5 on row x and column y coordinates, where $x, y \in \{0, 1, 2, 3, 4\}$. They are interpreted modulo 64 on lane coordinates $z \in \{0, \ldots, 63\}$.

Note also that addition ($+$) modulo 2 is exclusive-or (XOR or $\oplus$), that multiplication modulo 2 (juxtaposition or $\times$) is conjunction ($\wedge$) and that addition-by-1 modulo 2 ($s[n]{+}1$) is negation ($\neg s[n]$).

7.1 Sub-round function θ

The KECCAK reference specifies:

$$\boxed{\theta : \quad a[x][y][z] \quad \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1],}$$

This is taken to mean:

$$\begin{array}{l}
\forall x \in \{0\ldots4\}\ \forall y \in \{0\ldots4\}\ \forall z \in \{0\ldots63\}.\\
(\theta a)[x][y][z]\\
\quad = a[x][y][z] + \sum_{y'=0}^{4} a[x{-}1][y'][z] + \sum_{y'=0}^{4} a[x{+}1][y'][z{-}1]\\
\quad = a[x][y][z]\\
\qquad \oplus a[x{-}1][0][z] \oplus a[x{-}1][1][z] \oplus a[x{-}1][2][z] \oplus a[x{-}1][3][z] \oplus a[x{-}1][4][z]\\
\qquad \oplus a[x{+}1][0][z{-}1] \oplus a[x{+}1][1][z{-}1] \oplus a[x{+}1][2][z{-}1] \oplus a[x{+}1][3][z{-}1] \oplus a[x{+}1][4][z{-}1]
\end{array}$$

7.2 Sub-round function ρ

The KECCAK reference specifies:

$$\boxed{\begin{array}{lll}
\rho : & a[x][y][z] & \leftarrow a[x][y][z-(t+1)(t+2)/2],\\
& & \text{with } t \text{ satisfying } 0 \le t < 24 \text{ and } \begin{pmatrix}0 & 1\\ 2 & 3\end{pmatrix}^t \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}x\\y\end{pmatrix} \text{ in } \mathrm{GF}(5)^{2\times2},\\
& & \text{or } t = -1 \text{ if } x = y = 0,
\end{array}}$$

For each natural number n define x_n and y_n by:

$$\begin{pmatrix}x_n\\y_n\end{pmatrix} = \begin{pmatrix}0 & 1\\ 2 & 3\end{pmatrix}^n \begin{pmatrix}1\\0\end{pmatrix}$$

where the vector and matrix components are interpreted in GF(5) – i.e. the arithmetical operations are performed modulo 5. The definition of ρ is then:

```
$\forall x \in \{0\ldots4\}\ \forall y \in \{0\ldots4\}\ \forall z \in \{0\ldots63\}.$
 $(\rho a)[x][y][z]$
   = if $x = 0$ and $y = 0$
       then $a[0][0][z]$
       else if $(x, y) \in \{(x_n, y_n) \mid 0 \le n < 24\}$
              then let t = choose $n \in \{0\ldots23\}$ such that $(x, y) = (x_n, y_n)$
                     in $a[x][y][z-(t+1)(t+2)/2]$
              else $a[x][y][z]$
```

This definition of ρ appears non-deterministic as the value of t specified in the let-binding by "choose $n \in \{0\ldots23\}$ such that $(x, y) = (x_n, y_n)$" suggests the possibility that there may be more than one $n \in \{0 \ldots 23\}$ with $(x, y) = (x_n, y_n)$. However, this is not the case in fact as (x_n, y_n) are different for different values of $n \in \{0 \ldots 23\}$, so there is at most one such n with $(x, y) = (x_n, y_n)$ and thus t is uniquely defined.

7.3 Sub-round function π

The Keccak reference specifies:

$$\boxed{\pi: \quad a[x][y] \quad \leftarrow a[x'][y'], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix},}$$

Define:

$\mathsf{Pi}(x, y, x', y')$ if and only if $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$

The definition of π is then:

```
$\forall x \in \{0 \ldots 4\}\ \forall y \in \{0 \ldots 4\}\ \forall z \in \{0 \ldots 63\}.$
  $(\pi a)[x][y][z]$
    $=$ if $(\exists i\ j.\ \mathsf{Pi}(x, y, i, j))$
        then let $(x', y') =$ choose $(i, j)$ such that $\mathsf{Pi}(x, y, i, j)$
              in $a[x'][y'][z]$
        else $a[x][y][z]$
```

Although the use of "choose" suggests the definition of π might might be non-deterministic, in fact it isn't. For each (x, y) there is at most one (x', y') satisfying $\mathsf{Pi}(x, y, x', y')$.

7.4 Sub-round function χ

The Keccak reference specifies:

$$\boxed{\chi: \quad a[x] \quad \leftarrow a[x] + (a[x+1] + 1)a[x+2],}$$

The definition of χ is thus:

$$\begin{aligned} &\forall x \in \{0 \ldots 4\}\ \forall y \in \{0 \ldots 4\}\ \forall z \in \{0 \ldots 63\}. \\ &\quad (\chi a)[x][y][z] = a[x][y][z] + (a[x+1][y][z] + 1)a[x+2][y][z] \\ &\qquad\qquad\qquad\ = a[x][y][z] \oplus (\neg a[x+1][y][z] \wedge a[x+2][y][z]) \end{aligned}$$

7.5 Sub-round function ι

The Keccak reference specifies the rounds count i_r as ranging from 0 to the number of rounds $n_r = 12 + 2l = 24$. The sub-round function ι varies from round to round, i.e. depends on i_r.

The specification of ι in the Keccak reference is below.

$$\iota : \quad a \quad \leftarrow a + \mathrm{RC}[i_\mathrm{r}].$$

The additions and multiplications between the terms are in GF(2). With the exception of the value of the round constants $\mathrm{RC}[i_\mathrm{r}]$, these rounds are identical. The round constants are given by (with the first index denoting the round number)

$$\mathrm{RC}[i_\mathrm{r}][0][0][2^j - 1] = \mathrm{rc}[j + 7i_\mathrm{r}] \text{ for all } 0 \leq j \leq \ell,$$

and all other values of $\mathrm{RC}[i_\mathrm{r}][x][y][z]$ are zero. The values $\mathrm{rc}[t] \in \mathrm{GF}(2)$ are defined as the output of a binary linear feedback shift register (LFSR):

$$\mathrm{rc}[t] = \left(x^t \bmod x^8 + x^6 + x^5 + x^4 + 1\right) \bmod x \text{ in } \mathrm{GF}(2)[x].$$

The number of rounds n_r is determined by the width of the permutation, namely,

$$n_\mathrm{r} = 12 + 2\ell.$$

In the box above $\mathrm{RC}[i_r][0][0][2^j - 1] = \mathrm{rc}[j + 7i_r]$ and $\mathrm{rc}[t]$ is "defined as the output of a binary linear feedback shift register (LFSR)", where the LFSR is specified by a polynomial in $\mathrm{GF}(2)[x]$ (note the variable x in the polynomial is not the x-coordinate of the state).

Ideally the definition of RC and rc should be based on a formal representation of this polynomial, but this may be complex. For now, the definition will follow the Keccak specification summary web page[8] which gives explicit rounds constants using the following table defining, in hex, a 64-bit word `RC`$[i_r]$, for each round $i_r \in \{0 \dots 63\}$.

```
RC[ 0] = 0x0000000000000001   RC[12] = 0x000000008000808B
RC[ 1] = 0x0000000000008082   RC[13] = 0x800000000000008B
RC[ 2] = 0x800000000000808A   RC[14] = 0x8000000000008089
RC[ 3] = 0x8000000080008000   RC[15] = 0x8000000000008003
RC[ 4] = 0x000000000000808B   RC[16] = 0x8000000000008002
RC[ 5] = 0x0000000080000001   RC[17] = 0x8000000000000080
RC[ 6] = 0x8000000080008081   RC[18] = 0x000000000000800A
RC[ 7] = 0x8000000000008009   RC[19] = 0x800000008000000A
RC[ 8] = 0x000000000000008A   RC[20] = 0x8000000080008081
RC[ 9] = 0x0000000000000088   RC[21] = 0x8000000000008080
RC[10] = 0x0000000080008009   RC[22] = 0x0000000080000001
RC[11] = 0x000000008000000A   RC[23] = 0x8000000080008008
```

In the definition of ι below, `RC`$[i_r][z]$ denotes bit z of `RC`$[i_r]$, where z counts from the least significant end, so bit 0 is the rightmost bit of the binary representation of the hex number and bit 63 is the leftmost bit.

$$\forall x \in \{0 \dots 4\}\ \forall y \in \{0 \dots 4\}\ \forall z \in \{0 \dots 63\}.$$
$$(\iota a)[i_r][x][y][z] \ = \ \text{if } x = 0 \text{ and } y = 0 \text{ then } a[x][y][z] \oplus \texttt{RC}[i_r][z] \text{ else } a[x][y][z]$$

[8] `http://keccak.noekeon.org/specs_summary.html`