MAC-in-the-Box: Verifying a Minimalistic Hardware Design for MAC Computation

Abstract In this work, we study the verification of security properties of a minimalistic device called the MAC-in-the-Box (MITB) at its state machine level. This device computes a message authentication code based on the SHA-3 hash function and a key that is stored on device, but never output directly. It is designed for secure password storage, but may also be used for secure key-exchange and second-factor authentication. We formally verify, in the HOL4 theorem prover, that no outside observer can distinguish this device from an ideal functionality that provides only access to a hashing oracle. Furthermore, we propose protocols for the MITB's use in password storage, key-exchange and second-factor authentication, and formally show that it improves resistance against host-compromise in these three application scenarios.

1 Introduction

Practically all large providers of communication and banking services employ cryptographic hardware in their critical infrastructure. This ranges from expensive hardware security modules, used in the web's public-key infrastructure and the banking network, to low-cost devices like smart cards, used in mobile communication and health care. Their purpose is to separate and encapsulate sensitive cryptographic operations in a device that is (a) designed for security and (b) small enough to be audited. By encapsulating sensitive information within these small, purportedly secure devices, the surrounding system can exploit the flexibility of general-purpose operating systems to interoperate with its complex environment.

Despite this simplicity, and even despite their ubiquity — it is estimated that there are at least 30 billion smart cards in circulation [1] — the formal verification of security properties in cryptographic hardware designs (i.e. at the state machine level) has received little attention. So far, formal verification focused on functional correctness, i.e., the correctness w.r.t. the mathematical description of the algorithm, while the security of the algorithm was (hopefully) shown in a pen-and-paper proof. Historically, this was due to a lack of support for reasoning over probabilistic systems. Over the last years, this support was continuously improved with standalone proof assistants [2], as well as, frameworks for Coq [2], [3] and Isabelle/HOL [4], [5]. They were successfully used to show security properties for mathematical algorithms and even for software implementations, but not for hardware, due to their focus on probabilistic programs. By contrast, hardware is typically verified in higher-order logic, using mathematical functions to model its components [6].

In this work, we demonstrate the practicability of traditional hardware verification techniques for providing strong security guarantees, even when probabilistic reasoning is not available.[1] We develop a minimalistic device for the computation of *message authentication codes* (MACs) based on the recently standardised SHA-3 hash function. We call it *MAC-in-the-box* (MITB).

This device stores and protects a user-generated key. We can show that this minimalistic device provides strong guarantees such as confidentiality and unpredictability, given the usual assumption that the hash function behaves like a so-called *random oracle*. This holds even if its computing environment is under attack: In HOL4, we formally verify that, to any outside observer, the information gathered by an active attacker capable of compromising the MITB's host is limited by the information that can be gained from accessing a hashing oracle. In the random oracle model, this provides the desired guarantees by construction. This case study in security hardware design also shows the potential of formal analysis of hardware in the security setting: the verification helped us to identify three bugs in the early design of the MITB. We elaborate on these discovered issues in Sec. 7.

Despite its minimalism, the MITB can be used for various applications, e.g., establishment of secure channels and second-factor authentication. Its main application is to secure password databases. Password databases are frequently targeted to expose millions of passwords and exploit their reuse on other web pages. The MITB is initialised with a cryptographic key and stores a MAC of the password instead of the password itself. Even if this MAC is leaked, it is neither possible to recover the password from it, nor to guess popular passwords ('12345') without online access to the MITB. We formalise the password storage protocol, and protocols for two further applications, showing their security against host compromise in the symbolic model. Proof scripts and case studies are available online [7].

Paper organisation: In Sec. 2, we discuss related work on formally verified cryptographic implementations. In Sec. 3 we introduce the MITB. We define the security goal in Sec. 4. Sec. 5 to 7 describe the formalisation of the MITB, the threat model and the proof. Before we conclude, we outline three applications where the MITB improves the security after host-compromise in Sec. 8.

2 Related work

There are various approaches to support cryptographic reasoning in mainstream theorem provers. The most important aspects here are reasoning about outcome distributions of random processes described in terms of simple probabilistic programming languages, and reasoning about their runtime. Both CertiCrypt [9] and Verypto [4] were the pioneers in this regard, providing a deep embedding in Coq and Isabelle, respectively. EasyCrypt [2] is CertiCrypt's successor and

[1] We verify the MITB at the state machine level and leave the refinement to the gate-level for future work.

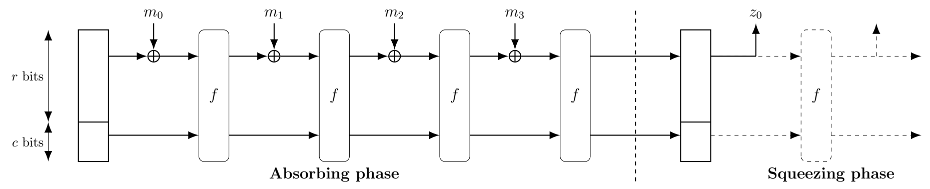

Figure 1. Sponge construction as in SHA-3 (adapted from [8]). The final output consists of the first n bits of z_0.

provides better automation by calling external SMT solvers. It is essentially a theorem prover on its own, with the downside that it is not designed to be as trustworthy as traditional theorem provers — a trade-off to speed up development. It is sometimes unsound [10], and sometimes the built-in tactics are just not expressive enough to prove properties that should follow from the semantics [11]. More recent approaches prefer a (semi-) shallow embedding to make it easier to use the theorem prover's libraries and reasoning infrastructure. The Foundational Cryptography Framework (FCF) extends Coq's built-in functional language Gallina with probabilistic semantics [3]. CryptoHOL [5] provides a shallow embedding in Isabelle/HOL. All of these approaches have not been designed to reason about hardware designs, which are typically described in terms of higher-order functions [6]. We side-step the need for probabilistic reasoning in this work and exemplify that, for some cases, it is possible to describe and prove cryptographic properties like secrecy with standard techniques. While FCF and CryptoHOL would certainly be useful to formalise surrounding protocols using the MITB, they currently both have drawbacks preventing that use. FCF's probabilistic semantics do not allow for recursions or exceptions, which would be used for modelling network routing and communication between the MITB and the protocol using it. CryptoHOL cannot express polynomial run-time, which is a prerequisite to formalising the threat model.

To our knowledge, all approaches for formal verification on the hardware level were showing the correctness w.r.t. a functional specification, i.e., in absence of an adversary [12], [13]. There are, however, formalised proofs for implementations written in C, e.g., for the random number generator HMAC-DRBG [14] and the HMAC construction in OpenSSL [15], and even for an x86-64 implementation of SHA-3 [16]. The latter uses Easycrypt and the Jasmin framework [17] to show the indistinguishability of the Sponge construction from a random oracle, as well as side-channel resistance of a highly-optimised implementation. Our goals are orthogonal: first, we want guarantees for a hardware design, not a software implementation. Second, we want to demonstrate how probabilistic reasoning can be avoided and traditional theorem provers be used.

Existing work on cryptographic hardware like the TPM [18]–[20], hardware security modules [21]–[25] or authentication tokens [26], [27] operate on the specification level, abstracting cryptographic bitstrings using a term algebra.

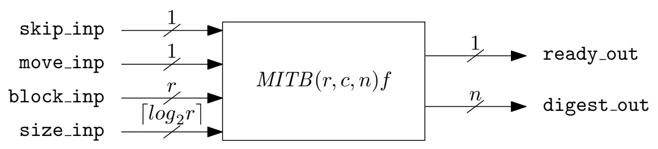

Figure 2. MITB: Inputs and Outputs

Implementation-specific aspects like the complicated state-machine needed to correctly apply padding or the low-level access available to an adversary after compromise are not represented in these models.

3 Hardware design

The MITB is a standalone device that computes a MAC using the Keccak family of hash functions [28], which NIST standardized as SHA-3 [29]. A nice feature of SHA-3 is that it can be used to compute a message authentication code (MAC) by simply prepending a secret key to the message, i.e., the function

$$mac(k, m) := \mathsf{SHA3}(k \| m)$$

is a valid MAC [30]. MACs operate as follows: to ensure the integrity of a message m, one computes $mac(k, m)$ and attaches it to the message. The communication partner, who also knows k, can recompute this function and compare the result to the MAC received. If the result is the same, the communication partner can be sure the message could only have been created by a party that knows k (typically either the sender or the receiver himself) and that m was not modified in transit. Previous hash functions like SHA-1, SHA-2 and MD5 were vulnerable to length-extension attacks and thus required more complicated constructions like HMAC to serve as MACs. (Cryptographic) hash functions themselves are functions that are difficult to invert and are resistant to collision attacks. In contrast to unforgeability, the main property of a MAC, it is difficult to formalise these properties, hence hash functions are often abstracted in terms of random oracles. A random oracle is a randomly chosen function from $\{0,1\}^* \to \{0,1\}^n$ where n is the length of the hash, i.e., each new input appears to be freshly sampled, but the function itself is deterministic.

The sponge construction SHA-3 is based on the 'sponge construction' in which an arbitrary length message is iteratively 'absorbed' into a finite state (see Figure 1). The number of iterations depends on the length of the message. Once all message blocks have been absorbed, the resulting state can be 'squeezed' to extract a digest. In general, this squeezing can be used to derive a digest of any desired length, however, all instances of fixed-length SHA-3 require only a single squeeze operation.

SHA-3 defines four instances of this construction, but recommends only one (the others are for testing and light-weight hashing). This instance is defined by

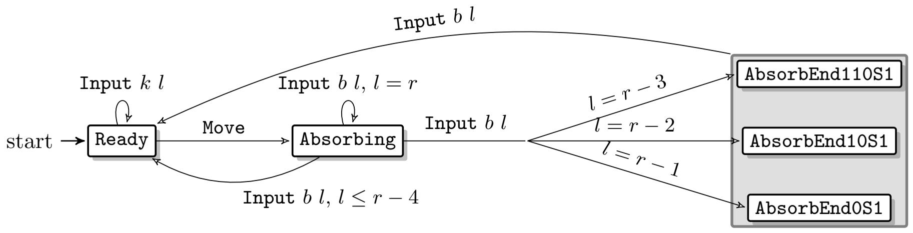

Figure 3. State transition diagram. Not shown: (a) `Skip` preserves state. (b) In any absorbing state, `Move` returns to `Ready`, but resets volatile memory to 0^{r+c}. Here, k and b are input blocks and l represent input size.

a *bitrate* of $r = 576$, a *capacity* of $c = 1024$ and an *output size* $n = 512 < r$, i.e., the number of bits to which the output is truncated. The state of the SHA-3 sponge algorithm thus consists of $r + c = 1600$ bits. Initially, the state is 0^{r+c}, i.e. each of the 1600 state bits is 0. In each iteration, a state permutation $f : \mathbb{Z}_2^{r+c} \to \mathbb{Z}_2^{r+c}$ is applied on the state. A detailed formal specification of f is not given here; f is treated as an uninterpreted parameter in the specification and proofs. That is, we do not make any assumptions about f itself other than that the sponge construction with f can be abstracted by a random oracle.

By itself, the sponge construction is only defined for input sizes that are multiples of r. Therefore, a 10^*1 padding is used, i.e., a bitstring with 1 at the beginning and end, and sufficiently many 0s in between. In addition, the SHA-3 specification requires two bits 01 to be added to distinguish fixed-length SHA-3 from its variable-length siblings SHAKE-128 and SHAKE-256. Hence any message m is padded to a multiple of r bits by appending at least four bits: first 011, then sufficiently many ($|m| + 4 \mod r$) zeroes, and finally a trailing 1. An empty message, e.g., is padded to a block $0110^{r-4}1$.

State machine The MITB computes $mac(k, m)$, but keeps k secret. It provides two operations: overwriting k (reading is not possible by an outsider), and computing $mac(k, m)$ for a given m. As we detail in Section 8, this functionality by itself is sufficient to improve the resilience of password databases, secure channel establishment and two-factor authentication against host compromise. As m can be arbitrarily long, the device operates in a blockwise fashion, absorbing 512 bits per call. Each full block is applied to the current state. If the user indicates an input of shorter length, a padding is applied and the operation finalised. Only if the padding was correctly applied, the state may be output.

The MITB has two 1-bit control inputs `skip_inp` and `move_inp`, two data inputs `block_inp` and `size_inp`, a 1-bit control output `ready_out` and a data output `digest_out`. It 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, and we require that $4 < r$, $0 < c$ and $n \leq r$.

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. Truth-values `T`, `F` model bits 1, 0, respectively.

The 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, the 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` (`ready_out` = `T`) or in some absorbing state (`ready_out` = `F`).

The input `skip_inp` 'freezes' the MITB: holding it `T` stops the state changing on successive cycles. If `skip_inp` is `F`, then 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. In the state machine, this is represented as a transition `Move`, no matter what values `block_inp` and `size_inp` have. If both `move_inp` and `skip_inp` are `F`, we consider this as a transition `Input block_inp size_inp`.

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.

In `Absorbing` state, if both `move_inp` and `skip_inp` are `F`, we absorb `block_inp` to to compute the MAC in a blockwise fashion. Depending on the message length, the padding might cause the message to extend to another block, in which case this additional block needs to be absorbed. As described in the previous section, the padding adds at least four bits, so a message that is 1, 2 or 3 bits short to the block length needs to have an extra block for the parts of the padding. Thus the need for three states that finalise the padding block. E.g., if the message is two bits short, i.e., `size_inp` is $r-2$ in `Absorbing`, then 01 is appended to the last block before absorbing it, and MITB moves into state `AbsorbEnd10S1` (the '`S`' can be read as '*'). There, a block $10^{r-2}1$ is absorbed (inputs `block_inp` and `size_inp` are ignored) before moving to `Ready` in the next cycle. Note that `Absorbing` also moves back to `Ready` if `size_inp` $\leq r-4$.

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 nor any other information than a valid MAC. In particular, no chain of inputs can leak parts of the state of the sponge-construction before the padding has been completed.

MAC computation protocol The protocol for computing the MAC of m, i.e., the SHA-3 hash of $k \| M$, is as follows:

1. If `ready_out` $= 0$, i.e., the device is not in `Ready` state, transition to the `Ready` state using `Move`, i.e., by inputting `F` on `skip_inp` and `T` on `move_inp` (`block_inp` and `size_inp` are ignored during this step).
2. The device is put into `Absorbing` state using another `Move`.
3. The user splits m into a sequence of blocks, $m = b_1\|b_2\|\cdots\ \|b_{m-1}\|b_n$, such that all blocks except the last one are r-bits wide, i.e., $|b_i| = r$ for $1 \leq i < n$ and $|b_n| < r$. If r divides exactly into $|m|$, then b_n is taken to be the empty block (so $|b_n| = 0$).
4. Starting on the next cycle, and continuing for n cycles, the user performs transitions `Input` b_i $|b_i|$, i.e., inputs `F` on both `move_inp` and `skip_inp`, b_i on `block_inp` and $|b_i|$ on `size_inp`, where $1 \leq i \leq n$. During this time, `F` will be output on `ready_out`.
5. After inputting b_n, the user keeps inputting `F` on `skip_inp` and `move_inp` for one more cycle, until `ready_out` becomes `T`. On the cycle when this happens the hash of $k\|n$ will appear on `digest_out`. The number of cycles taken depends on $|b_n|$. If $|b_n| \leq r{-}4$ then `ready_out` will become `T` on the cycle after b_n is input. If $r{-}4 < |b_n| < r$, then `ready_out` will become `T` the cycle after the cycle after b_n is input.

Key update protocol The protocol for updating the key to value k performs only two steps.

1. Exactly as step 1 in the MAC computation protocol.
2. Perform transitions `Input` k 576 by setting both `move_inp` and `skip_inp` to `F`, `block_inp` to k and `size_inp` to 576.

4 Security goals

The MITB is designed to protect password databases in case of a server breach, but can be used for many other different applications (see Section 8). We assume that an attacker may eventually gain control over this server, in which case the secrecy of the key should be preserved, but the attacker can compute MACs of her choice, re-set the key or send arbitrary other commands to the MITB. Before the attacker gains control, she shall not be able to compute or predict MACs.

Real-world / Ideal-world formulation Complex cryptographic properties are often formulated using the real-world / ideal-world paradigm. The real world describes how the cryptographic primitive or protocol interacts with the adversary, i.e., the threat model. The ideal world describes an idealised setup that provides the necessary guarantees by construction. In the case of signatures, e.g., all signatures are created by a central authority that keeps a list of message-signature pairs, so only message-signature pairs that it constructed itself are accepted, thus providing unforgeability by construction. Or, for encryption schemes, it outputs random bitstrings instead of a cyphertexts, thus providing confidentiality by construction. If it is not possible to distinguish both worlds, then the real-world scenario must be sufficiently close to the ideal world that it can be considered secure.

In our case, the real world consists of the MITB in communication with some environment, e.g., one of the applications in Section 8. The environment uses the protocol from Section 3 to compute a MAC or update the key, but can also bypass this protocol by declaring the host computer corrupted. In this case, the environment's inputs are directly transferred to the MITB.

The ideal world is specified by a simple machine that (a) stores or overwrites a key k; (b) for every MAC request m, calls a hash oracle with $k \| m$. (c) If the environment declares the host system corrupted, the attacker can do nothing more than to continue to query this oracle, in particular, she does not get access to k. Our security result in Theorem 1 can thus be informally stated as follows:

For all parameters r, c and n such that $r > 4$, $c > 0$ and $n \leq r$, and any sequence of inputs i, the sequence of outputs obtained by sending i to the real-world is equal to the sequence of outputs obtained by sending i to the ideal-world.

Hash functions and the random oracle model So far, a security definition for cryptographic hash functions that is both formal and directly applies to real-life hash functions has not been found. Properties like collision resistance postulate that there is no *known* adversary that can provoke a collision, but fundamentally, there are adversaries that can create collisions, due to the pigeonhole principle. We cannot formally reason about all known algorithms.

In cryptographic proofs, hash functions are thus usually abstracted using random oracles (ROs). A RO has two properties: First, when queried for a new bitstring m, RO draws a bitstring from the uniform distribution of bitstrings of length n. Second, if m was queried before, the RO responds with the same bitstring as before. Given that hash functions are deterministic, an RO can be distinguished from a hash function. Cryptographic results, e.g., indifferentiability [30] or PRF security [31] hence consider a sponge construction that calls an oracle to evaluate a randomly chosen permutation (and/or a keyed variant of the construction [31]). But in SHA-3, the permutation is public and fixed.

Theorem 1 *complements* these result. It relates the MITB to FMAC with the sponge construction and an arbitrary but fixed permutation. Indifferentiability relates FMAC with a random permutation to FMAC with a random oracle and holds within certain bounds on the adversary's computation time and the number of queries to the oracle. But observe that the step from ($\mathsf{Hash}\ f\ 0$) to a sponge with oracle access or a keyed sponge remains a heuristic; it is (provably) incorrect.

Guarantees provided by construction Due to the heuristic nature of the RO, we cannot perform this abstraction step in HOL4. Instead, the ideal world specification is parametric in the hash function. Once we instantiate the hash function in the ideal world with a randomly chosen hash function, as in the RO model,[2] we obtain a very clear interpretation of the guarantees that the ideal world we previously described provides.

1. *Confidentiality:* The output contains neither information about k, nor the message m (as it is merely a randomly chosen bitstring).
2. *Unpredictability:* The MAC computation yields unpredictable values for each k and m, as long as (k, m) were not queried before (as the result is chosen freshly).
3. *Determinism:* If (k, m) were previously queried, the MAC will be the same (as the RO is deterministic and $k \| m$ constitutes the query). This guarantees that a MAC can be verified later.
4. *Resistance against compromise:* The above guarantees hold even if the host system is compromised.

These guarantees go beyond the guarantees of message authentication codes (which are allowed to leak information about the authenticated message) or hash functions (which may leak information about the hashed message, but can be forged by everyone). We have designed the functionality specifically to suit secure password storage, our main application.

5 Formalising the MITB

We base the MITB's definition on a curried function `MITB_FUN`, which specifies the behaviour abstractly. `MITB_FUN` takes an abstract state $s \in \mathbb{S}$ and an input i, and returns the next state. The abstract state $s = \langle \texttt{cntl}, \texttt{pmem}, \texttt{vmem} \rangle$ is a tuple of control flags (i.e. $\texttt{cntl} \in \{\texttt{Ready}, \texttt{Absorbing}, \texttt{AbsorbEnd}(\texttt{0S1}|\texttt{10S1}|\texttt{110S1})\}$) that the control register can take, and permanent (`pmem`), as well as volatile (`vmem`) memories which are bit-strings of length $r+c$. The control flags correspond to the states described in Section 3. An input i can either be `Move`, `Skip` or `Input` *bk len*, where *bk* is a bitstring of size r and *len* corresponds to the number of bits of *bk* that constitutes the input block (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` uses ML-style pattern matching. Due to lack of space we skip presenting `MITB_FUN`'s formal definition here. However, its possible transitions are depicted in Figure 3. The most complex part of `MITB_FUN` specifies the state transition corresponding to absorbing a block. What happens depends on the input length. The complexity here is due to the padding applied by the devices, as described in Section 3. If the block length is less or equal to $r - 4$, e.g. 0, the device applies the padding and sets `cntl` to `Ready`. If the last block is one bit short of being a full block ($len = r - 1$) then one bit of padding is added and the device enters an absorbing state with `cntl` = `AbsorbEnd0S1`. On the next cycle, the remaining padding (i.e. $r - 1$ zeros and a final `T`) is added and the

[2] A randomly chosen hash function or an oracle that samples random hash values on demand are equivalent formulations of the RO model.

```
$\mathsf{MITB}\ f\ (cntl, pmem, vmem)$
        $(skip, move, block, size)\ =$
  $\mathsf{MITB\_FUN}\ f\ (cntl, pmem, vmem)$
   $($**if** $skip \iff \mathsf{T}$ **then** $\mathsf{Skip}$
    **else if** $move \iff \mathsf{T}$ **then** $\mathsf{Move}$
    **else if** $size \le r$ **then**
      $\mathsf{Input}\ block\ size$
    **else** $\mathsf{Skip})$
```

Figure 4. Definition of MITB function.

```
$\mathsf{MITB\_STEP}\ f\ s\ i\ =$
 **let** $(cntl', pmem', vmem')\ =\ \mathsf{MITB}\ f\ s\ i;$
   $digest\ =$
        $($**if** $cntl' \iff \mathsf{Ready}$
         **then** $(\mathsf{T}, (r\ -\ 1 >< 0)\ vmem')$
         **else** $(\mathsf{F}, \mathsf{ZERO}))$
 **in** $((cntl', pmem', vmem'), digest)$
```

Figure 5. Definition of step function.

permutation f is applied to `vmem` before transitioning back to the ready state. Similar steps are taken if the input block length is equal to $r-2$ or $r-3$. When the block size is exactly r, the device starts absorbing a non-final block, which is done by: (i) appending zero to it, (ii) XOR-ing the result with the current value of the volatile memory `vmem`, (iii) applying the KECCAK permutation f to the result of the XOR-ing, and finally, (iv) updating the volatile memory with the result of applying the function f.

Figure 4 shows the definition of the function MITB which decodes the inputs into abstract commands `Skip`, `Move` and `Input` and calls `MITB_FUN`.

We also define a *step function* (Figure 5) which yields the next state of the system. The step function behaves like the MITB, but defines the output, too. The step function takes a permutation f, the current state of the MITB, denoted as s, and the input $i = (\texttt{skip\_inp}, \texttt{move\_inp}, \texttt{block\_inp}, \texttt{size\_inp})$. It returns the next state of the MITB together with an output. The returned output depends on the value of `cntl` in the next cycle. In the definition, $(h >< l)\ w$ represents the HOL4 *bit extraction* function for input word w, and h and l are the upper and lower bound for the number of extracted bits, respectively.

6 Formalising security

The security of the MITB is defined in terms of a *functionality*, an idealised specification of both the functional correctness of the device, and the amount of information the adversary can learn from it. The way that this functionality is set up and relates to the MITB is defined in the popular 'universal composability' framework.

6.1 Universal composability

Our security definition follows the real-world/ideal-world paradigm, which can be generalised into a security notion called *emulation*. Emulation provides properties similar to refinement and entails a property called *universal composability*. Canetti introduced universal composability (UC) in a framework that goes by the same name [32], but there are several variations of it [33], [34]. If a protocol

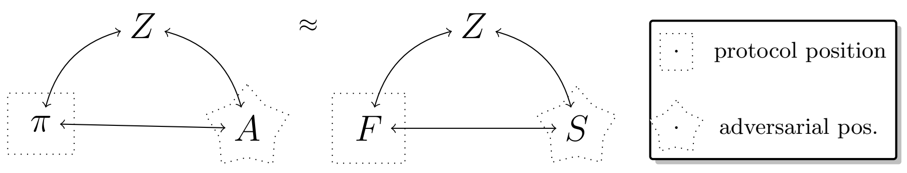

Figure 6. π (perfectly) emulates F iff, for all A, there exists S such that for all Z, the lhs network is indistinguishable (or instead: identical) from the rhs.

or cryptographic primitive π UC-emulates an 'ideal-world' system F (called the *functionality*), then π provides universal composability w.r.t. F. This means that for the analysis of any higher-level system ρ that uses π, it is sufficient to analyse the more abstract and thus simpler system where ρ interacts with F instead — even if multiple copies of π are used in parallel.

To formulate the security property, we formalise UC's communication framework and *perfect emulation*, the strongest variant of their refinement notion. We did not seek to prove that UC-emulation implies universal composability.

The *real world* is characterised by the protocol π communicating on two interfaces, the honest and the adversarial interfaces. The honest interface provides inputs, possibly from higher level protocols. A secure channel protocol like TLS, e.g., receives the instruction 'Party A requests sending m to Party B' and may later output 'Party B received m from A'. The adversarial interface models the network: all network communication, e.g., the encrypted TLS records, is sent to and received from an *adversary* A. The attacker can hence eavesdrop messages, but also drop or modify them.

In the *ideal world*, the functionality F receives the same high-level instructions on the honest interface.[3] Ideally, the messages on the network provide no useful information for the adversary. However, often enough this is impossible; encryption, e.g., reveals the length of the plaintext. Therefore, the functionality makes this so-called 'leakage' explicit by outputting, e.g., the length of the plaintext on the adversarial interface. To show that the network traffic leaks no information besides F's output on the dishonest interface, most emulation proofs construct a simulator S that imitates A's behaviour using only this intended leakage. Thus, to an outsider, no matter what inputs the protocol receives, for every attacker A there should be a simulator S such that π interacting with A is indistinguishable from F interacting with S. The comparison assures *correctness* (as any difference between the honest protocol and the ideal functionality on the honest interface can be observed) as well as *confidentiality up to the leakage in* F (if there is anything an adversary can learn and that the simulator cannot fake from the input provided by F, then there would not be a simulator that achieves indistinguishability). The inputs to the protocol can come from a high-level protocol using it (e.g., a p2p system using TLS for communication

[3] More precisely, F specifies this interface, and π tries to implement it accordingly.

between peers), and are abstracted as a Turing Machine Z. In the real world, Z interacts with π (on the honest interface) and with A (on the adversarial interface). In the ideal world, Z interacts with F and S.

It was shown[4] that for any environment Z and adversary A that are successful in distinguishing real world and ideal world, a new environment Z' can be constructed that simulates A and only relies on a special attacker A_d, the so-called *dummy-adversary.* The dummy-adversary only forwards messages between Z' and π. Thus, in practice, one shows the existence of a simulator S such that, for all environments Z, π and A_d are indistinguishable from F and S.

We formalise the communication structure in UC as follows. A message datatype indicates the routing for messages between the protocol position (occupied by π in real world, F in the ideal world), the adversary position (occupied by A_d in the real word, S in the ideal world) and the environment position (occupied by Z in both cases).

$$\mathsf{Message} = \mathsf{EnvtoP}\ \textbf{of}\ \alpha \mid \mathsf{EnvtoA}\ \textbf{of}\ \beta \mid \mathsf{PtoEnv}\ \textbf{of}\ \gamma \mid \mathsf{PtoA}\ \textbf{of}\ \delta \mid \mathsf{AtoEnv}\ \textbf{of}\ \eta \mid \mathsf{AtoP}\ \textbf{of}\ \phi$$

Initially, the environment sends a message of type EnvtoP or EnvtoA to either the adversary or the protocol. The function ROUTE p a, for a protocol step function p and an adversary step function a, expects the previous state of p and a and a value of type Message to be routed. It computes the next state of the protocol and adversary, as well as the next message to be routed. If the message matches EnvtoP or AtoP, the protocol step function is applied, the protocol's state updated, and the output converted into type Message, e.g.:

$$\begin{array}{l}\mathsf{ROUTE}\ \ p\ \ a\ \ ((state\_p, state\_a), \mathsf{EnvtoP}\ \ m) = \\ \quad (\textbf{let}\ (state\_p\_n, out)\ =\ p\ state\_p\ (\mathsf{EnvtoP}\ m) \\ \qquad \textbf{in}\ ((state\_p\_n, state\_a), \mathsf{Proto\_Wrapper}\ out))\end{array}$$

The wrapper Proto_Wrapper transforms a datatype for protocol output into Message, i.e., values that match either PtoEnv or PtoA. This ensures that the protocol cannot send messages that appear to originate from the adversary, and vice versa. Message addressing the adversary are handled analogously.

Before the environment is addressed again, there can be additional routing steps between the protocol and the adversary. Messages to the environment, however, terminate a routing step and are returned. We will later restrict the communication to three routing steps before the environment is again in control, which is sufficient for our case (otherwise, a routing error is produced).

$$\mathsf{ROUTE}\ p\ a\ ((state\_p, state\_a), \mathsf{PtoEnv}\ m)\ = ((state\_p, state\_a), \mathsf{PtoEnv}\ m)$$

Note that the scheduling model of UC gives control to the party that received a message. More elaborate scheduling mechanisms are modelled by including scheduling requests to the adversary in the protocol.

In UC, the environment is a Turing Machine, however, in this work, we consider the strongest notion of emulation, called *perfect emulation.* Here, the

[4] This simplification was proven sound for the UC framework [32], GNUC framework [33] and the IITM framework [34], so for the sake of brevity, we will assume it part of the definition.

sequence of messages the environment receives is the same (rather than indistinguishable to all polynomial time environments). We furthermore do not assume any runtime bounds on the participants.[5] This simplifies the analysis: We can model the set of environments as the set of input sequences and consider all other participants in terms of mathematical functions, and thus avoid probabilistic reasoning altogether. This is sufficient for the MITB because it is entirely deterministic, due to the key being generated outside the device and the deterministic nature of the hash function. We hence define ($\mathsf{EXEC}\ p\ a$), again on protocol and adversary step functions p and a, that applies a sequence of inputs i to an initial protocol and adversary state $s = (s_a, s_p)$ until one of these two parties outputs a message addressed to the environment.

We define a single execution step from the perspective of the environment as follows, where $\mathsf{ROUTE\_THREE}$ is just the threefold composition of ROUTE.

$$\begin{array}{l}
\mathsf{EXEC\_STEP}\ \ p\ \ a\ \ ((\mathit{state\_p}, \mathit{state\_a}), \mathit{input})\ \ = \\
(\mathbf{let} \\
\quad ((\mathit{state\_p\_n}, \mathit{state\_a\_n}), \mathit{out})\ \ = \\
\qquad \mathsf{ROUTE\_THREE}\ \ p\ \ a \\
\qquad\quad ((\mathit{state\_p}, \mathit{state\_a}), \mathsf{ENV\_WRAPPER}\ \ \mathit{input}) \\
\ \mathbf{in} \\
\quad ((\mathit{state\_p\_n}, \mathit{state\_a\_n}), \\
\quad\ \mathsf{GAME\_OUT\_WRAPPER}\ \ \mathit{out}))
\end{array}$$

The environment is fully described by the sequence of inputs it sends to the protocol or the adversary, hence an execution is defined as follows:

$$\begin{array}{l}
\mathsf{EXEC}\ \ p\ a\ s\ []\ =\ [] \\
\mathsf{EXEC}\ p\ \ a\ \ s\ \ (i :: il)\ \ = (\mathbf{let}\ (s', \mathit{out}) = \mathsf{EXEC\_STEP}\ p\ \ a\ \ (s, i)\ \mathbf{in}\ (s', \mathit{out}) :: \mathsf{EXEC}\ p\ \ a\ \ s'\ \ il)
\end{array}$$

6.2 Security definition

The real world The environment Z communicates with parties that compute MACs using the MITB via a library, as well as an attacker, who can take control over the machine the MITB is attached to and thus bypass this library. The attacker also communicates with Z and can thus provide Z with information that allows it to distinguish real world and ideal world. As the attacker is instantiated with the dummy-attacker, which is defined as follows, Z can access the adversarial interface, in this case the MITB, via this indirection.

$$\begin{array}{lcl}
\mathsf{DUMMY\_ADV}\ \ v_0\ \ (\mathsf{EnvtoA}\ \ m) & = & (0, \mathsf{Adv\_toP}\ \ m) \\
\mathsf{DUMMY\_ADV}\ \ v_1\ \ (\mathsf{PtoA}\ \ m) & = & (0, \mathsf{Adv\_toEnv}\ \ m)
\end{array}$$

For messages from the environment, PROTO models the protocols for MAC computation and key updates we defined in Section 3 (see Appendix B for the precise modelling).

[5] This is w.l.o.g. for all participants except for the simulator, which, however, is obvious to run in polynomial time in our case.

The ideal world In the ideal world, Z receives 'correct' output for whatever message it inputs. 'Correct' means the following: given a message $(SetKey, k)$, it stores k. For any subsequent message (Mac, m), it outputs $\mathcal{H}(k||m)$, where $\mathcal{H}$ is a Hash function. The function (FMAC H s) describes the output and next state of the ideal-world functionality in state s, parametrised with the hash function H to represent the hashing oracle $\mathcal{H}$. The only state that FMAC holds is the stored key k and the corruption status (`T` iff corrupted).

$$
\begin{aligned}
&\mathsf{FMAC}\ \ H\ \ (K', \mathsf{F})\ \ (\mathsf{EnvtoP}\ \ (\mathsf{SetKey}\ \ k))\ =\ ((k, \mathsf{F}), \mathsf{Proto\_toEnv}\ \ 0w)\\
&\mathsf{FMAC}\ \ H\ \ (K', \mathsf{F})\ \ (\mathsf{EnvtoP}\ \ (\mathsf{Mac}\ \ m))\ =\\
&\qquad\qquad ((K', \mathsf{F}), \mathsf{Proto\_toEnv}\ \ (H\ \ (\mathsf{word\_to\_bits}\ \ K'\ \ \|\ \ m)))
\end{aligned}
$$

After the corruption signal was received (and forwarded to the adversary), FMAC responds to oracle queries, computing $\mathcal{H}(k\|m)$:

$$
\begin{aligned}
&\mathsf{FMAC}\ \ H\ \ (K', \mathsf{F})\ \ (\mathsf{EnvtoP\ Corrupt})\ = ((K', \mathsf{T}), \mathsf{Proto\_toA\ WasCorrupted})\\
&\mathsf{FMAC}\ \ H\ \ (K', \mathsf{T})\ \ (\mathsf{AtoP}\ \ \mathsf{CorruptACK})\ \ = ((K', \mathsf{T}), \mathsf{Proto\_toEnv}\ \ 0w)\\
&\mathsf{FMAC}\ \ H\ \ (K', \mathsf{T})\ \ (\mathsf{AtoP}\ \ (\mathsf{OracleQuery}\ \ m))\ \ =\\
&\quad ((K', \mathsf{T}),\ \mathsf{Proto\_toA}\ (\mathsf{OracleResponse}\ (H\ \ (\mathsf{word\_wo\_bits}\ \ K'\ \ \|\ \ m))))
\end{aligned}
$$

The responses on the attacker interface formalise that the attacker does not receive information beyond the ability to compute MACs: Via the adversarial interface of FMAC, the simulator has access to the hash function $\mathcal{H}$, but not to the MITB. Note that, in contrast to the real world, FMAC notifies the simulator that it was corrupted, so the simulator knows whether it has to deny or simulate direct access to the MITB. After corruption, all honest queries are ignored:

$$
\begin{aligned}
&\mathsf{FMAC}\ \ H\ \ (K', \mathsf{T})\ \ (\mathsf{EnvtoP}\ \ (\mathsf{SetKey}\ \ v_{16}))\ = ((K', \mathsf{T}), \mathsf{Proto\_toEnv}\ \ 0w)\\
&\mathsf{FMAC}\ \ H\ \ (K', \mathsf{T})\ \ (\mathsf{EnvtoP}\ \ (\mathsf{Mac}\ \ v_{17}))\ \ = ((K', \mathsf{T}), \mathsf{Proto\_toEnv}\ \ 0w)
\end{aligned}
$$

Our main result is that, no matter which inputs the environment sends, the outputs are the same. In the next section, we will define a simulator SIM that mimics the behaviour of the MITB with access only to the hashing oracle provided by FMAC. We establish its existence by constructing it so that the following holds.

Theorem 1. *For all parameters r, c and n such that $r > 4$, $c > 0$ and $n \leq r$ and permutations $f : \{0,1\}^{r+c} \to \{0,1\}^{r+c}$, if the protocol and dummy adversary, as well as the functionality FMAC and the simulator are in their respective initial states s and s', then, for any sequence of inputs i, the output sequences $trace_{\mathsf{real}} = \mathit{EXEC}\ (\mathit{PROTO}\ (\mathit{MITB\_STEP}\ f))\ \mathit{DUMMY\_ADV}\ s\ i$, and $trace_{\mathsf{ideal}} = \mathit{EXEC}\ (\mathit{FMAC}\ (\mathit{Hash}\ f\ 0))\ (\mathit{SIM}\ \mathit{MITB\_STEP}\ f)\ s'\ i$ are equal.*

Note that f is a free variable in this theorem, and can stand for any permutation. The function (Hash f 0) formalises the sponge construction with permutation f and initial state 0^{r+c}, including the SHA-3 padding and the truncation to n bits. For lack of space, we will not elaborate on its formalisation. The protocol, which is parametric in the MITB step function, is instantiated with $\mathsf{MITB\_STEP}$ from Section 5, which itself is parametric in the underlying permutation f.

7 Proof overview

We proceed to outline the proof of Theorem 1. To this end, we first present the simulator and the relational invariant used to characterise possible states that the system can enter at runtime.

The simulator pretends to be the attacker from the real world, i.e., the dummy adversary. It simulates the information the functionality outputs in the ideal world, in particular the MITB's output after corruption. To imitate the MITB, without knowing the last key that was stored, the simulator uses the oracle $\mathcal{H}(k \| m)$, where $\mathcal{H}$ is a Hash function and $||$ denotes the bit-string concatenation function. The simulator SIM ignores queries until the variable *corrupted* is set. Afterwards, it parses each message m sent by the environment into an input $(\texttt{skip\_inp}, \texttt{move\_inp}, \texttt{block\_inp}, \texttt{size\_inp}) \in \mathbb{B} \times \mathbb{B} \times \{0,1\}_r \times \mathbb{N}_r$. We formulate the behaviour of SIM for the case where $\mathit{Corrupt} = \texttt{T}$ as a function on its state $(\texttt{cntl}, \texttt{vmem}, m, \texttt{overwt}, s) \in \{\texttt{Ready}, \texttt{Absorbing}, \texttt{AbsorbEnd}(\texttt{0S1}|\texttt{10S1}|\texttt{110S1})\} \times \{0,1\}^n \times \{0,1\}^* \times \mathbb{B} \times \mathbb{S}$ and the input $(\texttt{skip\_inp}, \texttt{move\_inp}, \texttt{block\_inp}, \texttt{size\_inp})$. The output of the function is a new state, and the simulated output of the MITB, i.e., $(\texttt{ready\_out}, \texttt{digest\_out}) \in \mathbb{B} \times \{0,1\}^n$. Due to lack of space, the detailed definition of SIM is included in Appendix C.

Here, we only present an extract from SIM's definition to show challenges involved in its formalization. The most interesting part of SIM's definition is how it acts as a buffer between the environment, which may successively send message blocks to be MACed, and the hashing oracle, which expects a single oracle query.

```
$\mathsf{SIM}\ \mathit{mitbf}\ f\ (\mathsf{Absorbing}, \mathit{vmem}, m, \mathsf{F}, c)\ (\mathsf{EnvtoA}\ (\mathsf{F}, \mathsf{F}, \mathit{inp}, \mathit{inp\_size}))\ =$
  if $\mathit{inp\_size} > r$
  then $((\mathsf{Absorbing}, \mathit{vmem}, m, \mathsf{F}, c), \mathsf{AtoEnv}\ (\mathsf{F}, \mathsf{ZERO}))$
  else if $\mathit{inp\_size} = r$
  then $((\mathsf{Absorbing}, \mathsf{ZERO}, m\ ||\ \mathsf{w2b}\ \mathit{inp}, \mathsf{F}, c),\ \mathsf{AtoEnv}\ (\mathsf{F}, \mathsf{ZERO}))$
  else if $\mathit{inp\_size} \leq r - 4$
  then if $\mathit{inp\_size} = 0$
  then$((\mathsf{Absorbing}, \mathit{vmem}, [\,], \mathsf{F}, c), \mathsf{AtoP}(\mathsf{OracleQuery}\ m))$
  else $((\mathsf{Absorbing}, \mathit{vmem}, [\,], \mathsf{F}, c),$
   $\mathsf{AtoP}\ (\mathsf{OracleQuery}(m\ ||\ \mathsf{TAKE}^{6}\mathit{inp\_size}(\mathsf{w2b}^{7}((\mathit{inp\_size} - 1 >< 0)\ \mathit{inp})))))$
  else
   (let $\mathit{state}$ = if $\mathit{inp\_size} = r - 1$ then $\mathsf{AbsorbEnd110S1}$
              else if $\mathit{inp\_size} = r - 2$ then $\mathsf{AbsorbEnd10S1}$
              else $\mathsf{AbsorbEnd0S1}$
   in
    $((\mathit{state}, \mathsf{ZERO},\ m\ ||\ \mathsf{TAKE}\ \mathit{inp\_size}\ ($
        $\mathsf{w2b}\ ((\mathit{inp\_size} - 1 >< 0)\ \mathit{inp})), \mathsf{F}, c),\ \ \mathsf{AtoEnv}\ (\mathsf{F}, \mathsf{ZERO}))))$
```

[6] The function $\mathsf{TAKE}\ i\ l$ returns the first i elements of the list l.
[7] $\mathsf{w2b}$ denotes the HOL4 function to convert words to bit string.

Table 1. Size of formal proof (in lines of code).

	definitions	theorems and proofs
universal composability	137	—
sponge construction	58	512
MITB	547	1962

In this case, depending on the input size, the simulator has to match the MITB's behaviour: If the input size is greater than r the simulator, like the MITB, skips this input and its state remains unchanged. When the input size is equal to r, the simulator adds the full block to its buffer m. If the input size is between 0 and $r-4$, the simulator adds the corresponding prefix of the input block to the buffer, and queries the oracle. If the input size is between $r-3$ and $r-1$, the appropriate `AbsorbEnd`(`0S1`|`10S1`|`110S1`) needs to be selected, as the MITB will not be able to output the MAC before yet another input is given. After sending the oracle query, the simulator awaits FMAC's response.

$$\begin{array}{l}\mathsf{SIM}\ \ \mathit{mitbf}\ \ f\ \ (-, \mathit{vmem}, m, \mathsf{F}, c)\\ \qquad (\mathsf{PtoA}\ \ (\mathsf{OracleResponse}\ \ \mathit{hashValue}))\ \ =\\ \quad ((\mathsf{Ready}, \mathit{hashValue}, [\,], \mathsf{F}, c), \mathsf{AtoEnv}\ \ (\mathsf{T}, \mathit{hashValue}))\end{array}$$

In order to prove emulation, we define an invariant which restricts valid states of the system. The invariant has three parts, each relating specific parts of real and ideal world states. The first part of the invariant relates the permanent and volatile memory in the real world to the key and the messages received so far in the real world, in case the MITB was corrupted. For the given key k in the real world, the permanent memory should always be $f(0^c||k)$. The volatile memory should be the result of absorbing, but not yet padding, the messages buffered by SIM according to the specification of the sponge construction. Depending on the simulated control-state, the length of the buffered messages should be off by 1,2,3 or 0 from a multiple of the block length and, in `Ready`-state, the first n bit of the volatile memory should equal the previous response from the hashing oracle. The case where the key was overwritten is an exception: now the simulator can execute the step function itself, without the oracle. In this case, the volatile and permanent memory of the MITB and the simulated counterpart have to match.

In addition to the memory invariant, the invariant consists of two other properties to guarantee that (i) corruption status in real and ideal world correspond, and that (ii) if the real game is corrupted, the control state of the MITB simulated by SIM and the actual MITB in the real game correspond. The proof of Theorem 1 proceeds by induction on the length of the input i. The base case ($i = [\ ]$) holds trivially. In the inductive step, we first prove that the starting states satisfy the constraints of the invariant. Then we show that, given the same input, a single step by the real and ideal worlds preserves the relational invariant. From that, we follow that the output in the successor state must be the same in both worlds. Tab. 1 gives details on the specification and proof size.

Intuitively, the existence of the simulator shows that all outputs of the MITB are hashes of correctly padded messages, and therefore non-revealing. The construction of the simulator pointed us to the need of an extra absorbing state — our initial design trusted the library to remember to request a final block. We incorporated the two bits that are meant to distinguish SHA-3 from the SHAKE family only later. Contrary to our initial intuition, this required the introduction of another two absorbing states, which manifested in the impossiblity of proving the memory invariant with only one absorbing state. Finally, failed attempts in showing the state invariant indicated the need for an initalisation procedure in MAC computation protocol (as well as the key update protocol), to ensure that the device is indeed in `Ready` state. These three flaws, which we discovered early on, while proving emulation of the ideal functionality, seem to be stereotypical flaws when designing hardware for a hostile environment. None of them would have been discovered by tests for functional correctness, and neither did our background in security prevent us from making these mistakes.

8 Applications

We propose three applications for the MITB. In each, it provides improved resilience against host-compromise. All properties we mention have been verified using off-the-shelf protocol verification tools. As the verification of protocols is notoriously difficult due to the complex interleaving of a possibly unbounded number of small programs running in concurrency and requires a large degree of automation. Hence, these tools operate in the symbolic model, where cryptographic outputs are abstracted using a term algebra, e.g., a MAC is a term of form $mac(k, m)$, where mac is a function symbol, and k, m themselves are terms. In these models, due to the use of the MITB, the key k remains secret, even if the attacker gains control over its host system. the Appendix D provides more details on the models.

Secure password storage: All businesses that store password data need to secure these password databases for the case where they get stolen. To store passwords securely, the MITB is initialised with a fresh key during set-up, and then used to compute MACs on the hashed and salted password. We used the recent DEEPSEC decision procedure [35] to show strong secrecy, i.e., resistance against offline password guessing. For a database of six passwords, i.e., $k = 6$, the decision procedure gives a positive results after 90s[8]. We also note that the MITB could replace the YubiHSM in an even more elaborate password storage scheme by Almeshekah et. al, which costs about $ 650.[9]

Establishing a secure channel: The MITB can be used to harden a variations of the signed Diffie-Hellman key-exchange protocol, which is used, e.g., in TLS

[8] Computed on a MacBook Pro with 3,1GHz Intel i7 and 16GB RAM.

[9] Pessl and Hutter managed to implement SHA-3 on an RFID token [36], citing cost estimates of about $ 0.05. As the MITB's state machine and key storage does not fundamentally add to that, the production cost of the MITB will likely be dominated by the bus technology connecting it to its host, e.g., USB.

and IPsec. Due to the MITB, this protocol provides *perfect forward secrecy* (even if the adversary gains control over one of the MITBs, all sessions keys established prior to this event remain secret) and *post-compromise security* (even if the adversary temporarily gains control over one of the MITBs, once the participants come together and set up a new key, future session keys will again remain secure.) We used the SAPIC/tamarin [37], [38] toolchain to establish both properties for an unbounded number of sessions. The proof is automated and terminates in 1516s.[12]

Two-factor authentication: We demonstrate that the MITB is compatible with the FIDO standard for universal 2nd factor authentication [39] (see Figure 7 in Appendix **??**). We again used SAPIC/tamarin to establish perfect forward security and post-compromise security for authentication, i.e., the property that any successful login on the webserver was initiated by the user. SAPIC/tamarin proves this theorem without any user intervention or helping lemmas within 9s.[12]

9 Conclusion

With the *MAC-in-the-box* we presented the first full fledged formal security argument for a hardware design. Despite its simplicity, the device has various applications. It also demonstrates that interactive theorem provers, which have an excellent track record for hardware verification, can in some cases be directly applied to the analysis of cryptographic constructions — even if support for probabilistic reasoning is missing or insufficient.

This technique can be applied where common abstractions in cryptography are heuristics rather than mathematically valid simplifications. Examples are random oracles for hash functions, or pseudorandom functions for block cyphers. Designs based on these primitives essentially argue that they provide proper access to these abstractions. For cases where this property holds unconditionally, our approach has advantages over cryptographic frameworks that come with additional proof obligations, or are not available for the theorem prover of choice.

References

[1] G. NV, *Smart card basics – a short guide (2019).* [Online]. Available: `https://www.gemalto.com/companyinfo/smart-cards-basics`.

[2] G. Barthe, B. Grégoire, S. Heraud and S. Zanella Béguelin, 'Computer-aided security proofs for the working cryptographer', in *Advances in Cryptology*, Springer, 2011, pp. 71–90.

[3] A. Petcher and G. Morrisett, 'The foundational cryptography framework', in *Proc. of 4th International Conference on Principles of Security and Trust (POST'15)*, 2015, pp. 53–72.

[4] M. Backes, M. Berg and D. Unruh, 'A formal language for cryptographic pseudocode', in *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, ser. Lecture Notes in Computer Science, Springer, 2008, pp. 353–376.

[5] A. Lochbihler, 'Probabilistic functions and cryptographic oracles in higher order logic', in *Programming Languages and Systems*, Springer Berlin Heidelberg, 2016, pp. 503–531.

[6] A. Camilleri, M. Gordon and T. Melham, 'Hardware verification using higher-order logic', University of Cambridge, Computer Laboratory, Tech. Rep., 1986.

[7] *Mac-in-the-box: Verifying a minimalistic hardware design for mac computation (extended).* [Online]. Available: `https://bit.ly/2TLpClW`.

[8] J. Jean, *TikZ for Cryptographers*, `https://www.iacr.org/authors/tikz/`, 2016.

[9] G. Barthe, B. Grégoire and S. Z. Béguelin, 'Formal certification of code-based cryptographic proofs', in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, ACM, 2009, pp. 90–101.

[10] D. Unruh, *Unsoundness in definition of 'glob M'*, Easycrypt Bug #17132, 2014. [Online]. Available: `https://www.easycrypt.info/trac/ticket/17132`.

[11] R. Künnemann, F. Dupressoir and D. Unruh, *Equivalence between while-loops w/ 1:1 mapping*, Thread on the Easycrypt-club mailing list, 2015. [Online]. Available: `http://lists.gforge.inria.fr/pipermail/easycrypt-club/2015-March/000292.html`.

[12] J.-P. Deschamps, *Hardware Implementation of Finite-Field Arithmetic*, 1st ed. McGraw-Hill, Inc., 2009.

[13] L. Erkök, M. Carlsson and A. Wick, 'Hardware/software co-verification of cryptographic algorithms using cryptol', in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, IEEE, 2009, pp. 188–191.

[14] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher and A. W. Appel, 'Verified correctness and security of mbedTLS HMAC-DRBG', in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, ACM, 2017, pp. 2007–2020.

[15] L. Beringer, A. Petcher, Q. Y. Katherine and A. W. Appel, 'Verified correctness and security of openssl hmac.', in *USENIX Security Symposium*, 2015, pp. 207–221.

[16] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton and P. Strub, 'Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3', in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications*

Security, CCS 2019, London, UK, November 11-15, 2019, ACM, 2019, pp. 1607–1622.

[17] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt and P. Strub, 'Jasmin: High-assurance and high-speed cryptography', in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, ACM, 2017, pp. 1807–1823.

[18] S. Delaune, S. Kremer, M. Ryan and G. Steel, 'A formal analysis of authentication in the TPM', in *Formal Aspects of Security and Trust*, ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 111–125.

[19] S. Delaune, S. Kremer, M. D. Ryan and G. Steel, 'Formal analysis of protocols based on TPM state registers', in *24th IEEE Computer Security Foundations Symposium (CSF'11)*, IEEE Comp. Soc., 2011, pp. 66–82.

[20] J. Shao, Y. Qin, D. Feng and W. Wang, 'Formal analysis of enhanced authorization in the TPM 2.0', in *10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*, ACM, 2015, pp. 273–284.

[21] S. Delaune, S. Kremer and G. Steel, 'Formal analysis of PKCS#11 and proprietary extensions', *Journal of Computer Security*, vol. 18, no. 6, pp. 1211–1245, 2010.

[22] M. Bortolozzo, M. Centenaro, R. Focardi and G. Steel, 'Attacking and fixing PKCS#11 security tokens', in *17th ACM Conference on Computer and Communications Security (CCS'10)*, ACM, 2010, pp. 260–269.

[23] S. Kremer, R. Künnemann and G. Steel, 'Universally composable key-management', in *European Symposium on Research in Computer Security*, Springer, 2013, pp. 327–344.

[24] G. Scerri and R. Stanley-Oakes, 'Analysis of key wrapping APIs: Generic policies, computational security', in *29th Computer Security Foundations Symposium*, IEEE Computer Society, 2016, pp. 281–295.

[25] A. Dax, S. Tangermann, R. Künnemann and M. Backes, 'How to wrap it up - a formally verified proposal for the use of authenticated wrapping in PKCS#11', in *Computer Security Foundations Symposium*, 2019.

[26] R. Künnemann and G. Steel, 'Yubisecure? formal security analysis results for the yubikey and yubiHSM', in *Proc. 8th Workshop on Security and Trust Management (STM'12)*, ser. LNCS, 2012, pp. 257–272.

[27] C. Jacomme and S. Kremer, 'An extensive formal analysis of multi-factor authentication protocols', in *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, IEEE Computer Society, 2018, pp. 1–15.

[28] G. Bertoni, J. Daemen, M. Peeters and G. V. Assche, *Online keccak specifications*, 2009. [Online]. Available: `http://keccak.noekeon.org/`.

[29] M. J. Dworkin, 'Sha-3 standard: Permutation-based hash and extendable-output functions', Tech. Rep., 2015.

[30] G. Bertoni, J. Daemen, M. Peeters and G. V. Assche, ‘On the indifferentiability of the sponge construction’, in *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, ser. Lecture Notes in Computer Science, Springer, 2008, pp. 181–197.

[31] P. Gaži, K. Pietrzak and S. Tessaro, ‘The exact prf security of truncation: Tight bounds for keyed sponges and truncated cbc’, in *Annual Cryptology Conference*, Springer, 2015, pp. 368–387.

[32] R. Canetti, ‘Universally composable security: A new paradigm for cryptographic protocols’, in *Foundations of Computer Science*, IEEE Computer Society, 2001, pp. 136–145.

[33] D. Hofheinz and V. Shoup, *GNUC: A new universal composability framework*, Cryptology ePrint Archive, 2011. [Online]. Available: `http://eprint.iacr.org/`.

[34] R. Küsters and M. Tuengerthal, ‘The IITM Model: a Simple and Expressive Model for Universal Composability’, Cryptology ePrint Archive, Tech. Rep. 2013/025, 2013. [Online]. Available: `http://eprint.iacr.org/`.

[35] V. Cheval, S. Kremer and I. Rakotonirina, ‘The DEEPSEC prover’, in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, Springer, 2018, pp. 28–36.

[36] P. Pessl and M. Hutter, ‘Pushing the limits of SHA-3 hardware implementations to fit on RFID’, in *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, ser. Lecture Notes in Computer Science, Springer, 2013, pp. 126–141.

[37] B. Schmidt, S. Meier, C. Cremers and D. Basin, ‘The tamarin prover for the symbolic analysis of security protocols’, in *25th International Conference on Computer Aided Verification (CAV’13)*, ser. LNCS, Springer, 2013, pp. 696–701.

[38] S. Kremer and R. Künnemann, ‘Automated analysis of security protocols with global state’, *Journal of Computer Security*, P. Samarati and A. Myers, Eds., 2016.

[39] S. Srinivas, D. Balfanz, E. Tiffany, F. Alliance and A. Czeskis, ‘Universal 2nd factor (U2F) overview’, *FIDO Alliance Proposed Standard*, pp. 1–5, 2015.

[40] L. C. Paulson, ‘The inductive approach to verifying cryptographic protocols’, *Journal of Computer Security*, vol. 6, no. 1-2, pp. 85–128, 1998.

[41] Solar Designer, *Password hashing at scale*, Slides for a talk at YaC 2012 (see Slide 9), 2012. [Online]. Available: `http://www.openwall.com/presentations/YaC2012-Password-Hashing-At-Scale/`.

[42] N. Durgin, P. Lincoln, J. Mitchell and A. Scedrov, ‘Undecidability of bounded security protocols’, in *Workshop on Formal Methods and Security Protocols*, IEEE Computer Society, 1999.

[43] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah and E. H. Spafford, ‘Ersatzpasswords: Ending password cracking and detecting password leakage’, in *Proceedings of the 31st Annual Computer Security Applications Conference*, ACM, 2015, pp. 311–320.

A MITB Formalization

We base the MITB’s definition on a curried function `MITB_FUN`, which specifies the behaviour abstractly. `MITB_FUN` uses ML-style pattern matching and there are separate equations for the various combinations of values of `cntl` and the input i. In what follows we try to briefly explain some of these equations. The first case is when `Skip` appears in the input. In this case, the state stays the same:

$$\begin{aligned}&\mathsf{MITB\_FUN}\ f\ \ (cntl, pmem, vmem)\ \ \mathsf{Skip}\\ &\quad=\ (cntl, pmem, vmem)\end{aligned}$$

Starting in the ready state (i.e. `cntl` = `Ready`), if the input is set to `Input` *key len* then the permanent memory `pmem` is set to $f\quad(\mathsf{ZERO@@}key)$, the volatile memory `vmem` is set to ZERO, and the state remains ready. However, 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:

$$\begin{aligned}&\mathsf{MITB\_FUN}\ f\ \ (\mathsf{Ready}, pmem, vmem)\ \ (\mathsf{Input}\ \ key\ \ len)\\ &\quad=\ (\mathsf{Ready}, f\ \ (\mathsf{ZERO}\ \ @@\ \ key), \mathsf{ZERO})\end{aligned}$$

$$\begin{aligned}&\mathsf{MITB\_FUN}\ f\ \ (\mathsf{Ready}, pmem, vmem)\ \ \mathsf{Move}\\ &\quad=\ (\mathsf{Absorbing}, pmem, pmem)\end{aligned}$$

Furthermore, following cases specify what happens if `Move` is input whilst absorbing. In this case, the volatile memory is reset to zeros and the device returns to the ready state:

$$\begin{aligned}&\mathsf{MITB\_FUN}\, f\ \ (\mathsf{Absorbing}, pmem, vmem)\ \mathsf{Move}\\ &\quad=\ (\mathsf{Ready}, pmem, \mathsf{ZERO})\end{aligned}$$

$$\begin{aligned}&\mathsf{MITB\_FUN}\, f\ (\mathsf{AbsorbEnd}(\mathsf{0S1}\ |\mathsf{10S1}\ |\ \mathsf{110S1}),\\ &\quad pmem, vmem)\ \mathsf{Move} = (\mathsf{Ready}, pmem, \mathsf{ZERO})\end{aligned}$$

The most complex part of `MITB_FUN` specifies the state transition corresponding to absorbing a block. What happens depends on the input length. The complexity here is due to the padding applied by the devices, as described in Section 3. If the block length is zero or it is less than equal to $r - 4$ the device applies padding and enters to the state of ready. The only difference is that when the block length is zero full-padding needs to be applied (cf. Section 3). If the last block is one bit short of being a full block of length r ($len = r - 1$) then one bit is added and the device enters the state of absorbing with `cntl` = `AbsorbEnd0S1`, and on the next cycle the remaining padding (i.e., $r - 1$ zeros and a final T) is added and the permutation f will be applied before transitioning back to the ready state:

$$(\mathsf{Ready}, pmem, f\ \ (vmem\ \ \oplus\ \ \mathsf{ZERO}\ \ @@\ \ \mathsf{INT\_MINw}))$$

Where INT_MINw expresses the 0*1 block. Similar steps also taken if the input block length is equal to $r - 2$ or $r - 3$. However, when the block size is exactly r, the device start absorption of a non-final block, which is done by: (i) appending zero to it, (ii) XOR-ing the result with the current value of the volatile memory `vmem`, (iii) applying the Keccak permutation f to the result of the XOR-ing, and finally (iv) updating the volatile memory with the result of applying the function f. Also, note that if the condition of the last conditional expression does not hold then the state stays the same (i.e. behave like `Skip`):

```
MITB_FUN $f$ (Absorbing, $pmem$, $vmem$) (Input $blk$  $len$) =
 (let $apply$  $x$  =  $f$  ($vmem$  $\oplus$  ZERO  @@  $x$) ;
      $g$  $len$  =  ($len$  $-$  1 >< 0)  $blk$
               && SHA3_APPEND_ZERO_WORD  $len$
               || SHA3_APPEND_ONE_WORD  $len$
               || PAD_WORD ($len$  +  2) ;
   $r$  =  dimindex (: $\varsigma$)
  in
     if $len$  =  0  then
       (Ready, $pmem$,
        $apply$
         (SHA3_APPEND_ZERO_WORD  $len$
          && SHA3_APPEND_ONE_WORD  $len$
          || PAD_WORD ($len$  +  2)))
     else if $len$  $\leq$  $r$  $-$  4
        then (Ready, $pmem$, $apply$  ($g$  $len$))
     else if $len$  =  $r$  $-$  3
        then (AbsorbEnd0S1, $pmem$, $apply$  ($g$  $len$))
     else if $len$  =  $r$  $-$  2
        then (AbsorbEnd10S1, $pmem$, $apply$  ($g$  $len$))
     else if $len$  =  $r$  $-$  1
        then
       (AbsorbEnd110S1, $pmem$,
        $apply$
          (($len$  $-$  1 >< 0)  $blk$
            && SHA3_APPEND_ZERO_WORD $len$))
     else if $len$  =  $r$  then (Absorbing, $pmem$, $apply$  $blk$)
     else (Absorbing, $pmem$, $vmem$))
```

The following auxiliary functions are used in the definition of `MITB_FUN`: (i) SHA3_APPEND_ZERO_WORD, which returns a word that is one except at the position given as parameter, (ii) SHA3_APPEND_ ONE_WORD, which returns a word that is zero expect at the position given as parameter plus 1, and (iii) the padding function PAD_WORD. Moreover, $(h \mathrel{>\!<} l)\ w$ represents the HOL4 *bit extraction* function for input word w, and h and l are the upper and lower bound respectively for the number of extracted bits.

Based on the `MITB_FUN`'s definition, we then define the function MITB which is similar to `MITB_FUN` except that it decodes the inputs into abstract commands `Skip`, `Move` and `Input` *bk len*.

```
MITB $f$ $(cntl, pmem, vmem)$
        $(skip, move, block, size)$  $=$
  MITB_FUN $f$ $(cntl, pmem, vmem)$
   (if $skip \iff$ T then Skip
     else if $move \iff$ T then Move
     else if $size \le r$ then
        Input $block\ size$
     else Skip)
```

Then we proceed to define a *step function* which yields the next state of the system. The step function behaves like the MITB, but defines the output, too. It takes a permutation f, current state of the MITB denoted as s, and the input i = (`skip_inp`, `move_inp`, `block_inp`, `size_inp`) and returns the next state of the MITB together whit an output. It's output depends on the state that the device enters in the next cycle.

```
MITB_STEP $f\ s\ i$ $=$
 let $(cntl', pmem', vmem')$ $=$ MITB $f\ s\ i$;
     $digest$ $=$
          (if $cntl' \iff$ Ready
           then $(\mathsf{T}, (r - 1 \mathrel{>\!<} 0)\ vmem')$
           else (F, ZERO))
 in  $((cntl', pmem', vmem'), digest)$
```

B Details on the real-word protocol definition

For messages from the environment, PROTO_def models the protocol for MAC computation we defined in Section 3.

```
PROTO $mitbf$ $(s, \mathsf{F})$ (EnvtoP (Mac $m$)) =
  (let
     $(s_0, rdy_0, dig_0)$ =
       RunMITB $mitbf$ $s$ $[(\mathsf{T}, \mathsf{F}, \mathsf{ZERO}, 0)]$ ;
     $(sr, rdyr, digr)$ =
       if $rdy_0$ $\iff$ F then
         RunMITB $mitbf$ $s_0$ $[(\mathsf{F}, \mathsf{T}, \mathsf{ZERO}, 0)]$
       else $(s_0, rdy_0, dig_0)$ ;
     $(ss, rdys, digest)$ =
       RunMITB $mitbf$ $sr$
         $((\mathsf{F}, \mathsf{T}, \mathsf{ZERO}, 0)$ ::
               PROCESS_MESSAGE_LIST
                 (Split (dimindex $(: \varsigma)$) $m$)) ;
     $(sq, rdyq, digq)$ =
       RunMITB $mitbf$ $ss$
         $[(\mathsf{F}, \mathsf{T}, \mathsf{ZERO}, 0); \ (\mathsf{F}, \mathsf{T}, \mathsf{ZERO}, 0)]$
   in
     $((sq, \mathsf{F})$, Proto_toEnv $digest$))
```

Similarly, to insert a new key, the MITB is first brought into `Ready` state, and then the key is input.

```
PROTO $mitbf$ $(s, \mathsf{F})$ (EnvtoP (SetKey $k$)) =
  (let
     $(s_1, rdy_1, dig_1)$ = $mitbf$ $(s, \mathsf{T}, \mathsf{F}, \mathsf{ZERO}, 0)$
   in
     if $rdy_1$ $\iff$ F then
       (let
          $(s_2, rdy_2, dig_2)$ = $mitbf(s_1, \mathsf{F}, \mathsf{T}, \mathsf{ZERO}, 0)$ ;
          $(s_3, rdy_3, dig_3)$ =
            $mitbf$ $(s_2, \mathsf{F}, \mathsf{F}, k$, dimindex $(: \varsigma))$
        in
          $((s_3, \mathsf{F})$, Proto_toEnv $0w$))
     else
       (let
          $(s_2, rdy_2, dig_2)$ =
            $mitbf$ $(s_1, \mathsf{F}, \mathsf{F}, k$, dimindex $(: \varsigma))$
        in
          $((s_2, \mathsf{F})$, Proto_toEnv $0w$)))
```

Besides the state of the MITB, the protocol stores a boolean corruption state. The environment can signal that the host computer was corrupted, to switch its state from `F` to `T`.

$$\begin{array}{l}\mathsf{PROTO}\ \mathit{mitbf}\ (s,\mathsf{F})\ (\mathsf{EnvtoP\ Corrupt})\ =\\ \quad((s,\mathsf{T}),\mathsf{Proto\_toEnv}\ 0w)\end{array}$$

From now on, all adversarial inputs are forwarded to the MITB.

$$\begin{array}{l}\mathsf{PROTO}\ \ \mathit{mitbf}\ \ (s,\mathsf{T})\ \ (\mathsf{AtoP}\ \ i)\ \ =\\ \quad(\mathbf{let}\\ \qquad(s\_next,\mathit{rdy},\mathit{dig})\ \ =\ \ \mathit{mitbf}\ \ (s,i)\\ \quad\ \mathbf{in}\\ \qquad((s\_next,\mathsf{T}),\mathsf{Proto\_toA}\ \ (\mathit{rdy},\mathit{dig})))\end{array}$$

All other queries are ignored, e.g., honest queries after corruption or adversarial queries before corruption.

C Simulator Formalization

In what follows we present SIM's formal definition. The simulator SIM ignores queries until the variable *corrupted* is set. Afterwards, it parses each message m sent by the environment into an input $(\texttt{skip\_inp}, \texttt{move\_inp}, \texttt{block\_inp}, \texttt{size\_inp}) \in \mathbb{B} \times \mathbb{B} \times \{0,1\}_r \times \mathbb{N}_r$. The purpose of the simulator is to imitate the MITB, for this SIM uses the oracle $\mathcal{H}(k\|m)$. The output of SIM is a new state, and the simulated output of the MITB, that is $(\texttt{ready\_out}, \texttt{digest\_out}) \in \mathbb{B} \times \{0,1\}^n$.

When `Skip` is set to true, if the simulator is in the ready state the content of volatile memory is sent out and in all the other cases the simulator's state remains unchanged. If `Move` is set to true and when the simulator is in the ready state it will move to absorbing state. However, if the control register is set to other control-states the simulator outputs zero and moves to ready state.

$$\begin{array}{l}\mathsf{SIM}\ \ \mathit{mitbf}\ \ f\ \ (\mathsf{T},\mathsf{Ready},\mathit{vm},m,\mathsf{F},c)\\ \quad(\mathsf{EnvtoA}\ \ (\mathsf{T},v_0,v_1,v_2))\ \ =\\ \quad((\mathsf{T},\mathsf{Ready},\mathit{vm},m,\mathsf{F},c),\mathsf{Adv\_toEnv}\ \ (\mathsf{T},\mathit{vm}))\\ \mathsf{SIM}\ \ \mathit{mitbf}\ \ f\ \ (\mathsf{T},\mathsf{Absorbing},\mathit{vm},m,\mathsf{F},c)\\ \quad(\mathsf{EnvtoA}\ \ (\mathsf{T},v_3,v_4,v_5))\ \ =\\ \quad((\mathsf{T},\mathsf{Absorbing},\mathit{vm},m,\mathsf{F},c),\mathsf{Adv\_toEnv}\ \ (\mathsf{F},\mathsf{ZERO}))\\ \mathsf{SIM}\ \ \mathit{mitbf}\ \ f\ \ (\mathsf{T},\mathsf{Ready},\mathit{vm},m,\mathsf{F},c)\\ \quad(\mathsf{EnvtoA}\ \ (\mathsf{F},\mathsf{T},v_{15},v_{16}))\ \ =\\ \quad((\mathsf{T},\mathsf{Absorbing},\mathit{vm},[],\mathsf{F},c),\mathsf{Adv\_toEnv}\ \ (\mathsf{F},\mathsf{ZERO}))\\ \mathsf{SIM}\ \ \mathit{mitbf}\ \ f\ \ (\mathsf{T},\mathsf{Absorbing},\mathit{vm},m,\mathsf{F},c)\\ \quad(\mathsf{EnvtoA}\ \ (\mathsf{F},\mathsf{T},v_{17},v_{18}))\ \ =\\ \quad((\mathsf{T},\mathsf{Ready},\mathsf{ZERO},m,\mathsf{F},c),\mathsf{Adv\_toEnv}\ \ (\mathsf{T},\mathsf{ZERO}))\end{array}$$

More interesting part of the SIM's definition is when the simulator is in the absorbing state and receives some input.

```
$\mathsf{SIM}\ \mathit{mitbf}\ f\ (\mathsf{Absorbing}, \mathit{vmem}, m, \mathsf{F}, c)\ (\mathsf{EnvtoA}\ (\mathsf{F}, \mathsf{F}, \mathit{inp}, \mathit{inp\_size}))\ =$
 **if** $\mathit{inp\_size} > r$
 **then** $((\mathsf{Absorbing}, \mathit{vmem}, m, \mathsf{F}, c), \mathsf{AtoEnv}\ (\mathsf{F}, \mathsf{ZERO}))$
 **else** **if** $\mathit{inp\_size} = r$
 **then** $((\mathsf{Absorbing}, \mathsf{ZERO}, m \mathbin{||} \mathsf{w2b}\ \mathit{inp}, \mathsf{F}, c),\ \mathsf{AtoEnv}\ (\mathsf{F}, \mathsf{ZERO}))$
 **else** **if** $\mathit{inp\_size} \le r - 4$
 **then** **if** $\mathit{inp\_size} = 0$
 **then**$((\mathsf{Absorbing}, \mathit{vmem}, [\,], \mathsf{F}, c), \mathsf{AtoP}(\mathsf{OracleQuery}\ m))$
 **else** $((\mathsf{Absorbing}, \mathit{vmem}, [\,], \mathsf{F}, c),$
  $\mathsf{AtoP}\ (\mathsf{OracleQuery}(m \mathbin{||} \mathsf{TAKE}^{10} \mathit{inp\_size}(\mathsf{w2b}^{11}((\mathit{inp\_size} - 1 >< 0)\ \mathit{inp})))))$
 **else**
  $(\textbf{let}\ \mathit{state} = \textbf{if}\ \mathit{inp\_size} = r - 1\ \textbf{then}\ \mathsf{AbsorbEnd110S1}$
           $\textbf{else}\ \textbf{if}\ \mathit{inp\_size} = r - 2\ \textbf{then}\ \mathsf{AbsorbEnd10S1}$
           $\textbf{else}\ \mathsf{AbsorbEnd0S1}$
  **in**
   $((\mathit{state}, \mathsf{ZERO},\ m \mathbin{||} \mathsf{TAKE}\ \mathit{inp\_size}\ ($
       $\mathsf{w2b}\ ((\mathit{inp\_size} - 1 >< 0)\ \mathit{inp})), \mathsf{F}, c),\ \mathsf{AtoEnv}\ (\mathsf{F}, \mathsf{ZERO}))))$
```

What happens in this case depends on the input size. When the size of the input message is greater than $r - 4$ the simulator behaves like it is in the `Skip` state; that is, simulator's state remains unchanged. When the input size is equal to r the simulator adds the full block to its buffer m, and if the input size is 0 or less than equal to $r - 4$ the simulator queries the oracle and proceeds only when it receives the FMAC's response:

$$\begin{array}{l} \mathsf{SIM}\ \mathit{mitbf}\ f\ (-, \mathit{vmem}, m, \mathsf{F}, c) \\ \quad (\mathsf{PtoA}\ (\mathsf{OracleResponse}\ \mathit{hashValue}))\ = \\ \quad ((\mathsf{Ready}, \mathit{hashValue}, [\,], \mathsf{F}, c), \mathsf{AtoEnv}\ (\mathsf{T}, \mathit{hashValue})) \end{array}$$

Finally, if the input size is greater than $r-4$ and less than equal to $r-1$ part of the input is taken and the simulator enters the corresponding **`AbsorbEnd`**(`0S1`|`10S1`|`110S1`) state, in which depending on the size of the input either the simulator queries the oracle or it behaves like it is in the `Skip` state.

In the **`AbsorbEnd`**(`0S1`|`10S1`|`110S1`) state depending on the size of the input either the simulator queries the oracle or it behaves like it is in the `Skip` state. For example the following snippet shows what happens when the simulator is the `AbsorbEnd0S1` state. Similar things also happens in states `AbsorbEnd10S1` and `AbsorbEnd110S1`.

```
⊢ SIM mitbf f (T, AbsorbEnd0S1, vm, m, F, c)
      (EnvtoA (F, F, inp, inp_size)) =
    if inp_size ≤ dimindex (: ς) then
      ((T, AbsorbEnd0S1, vm, [], F, c),
       Adv_toP (OracleQuery m))
    else ((T, AbsorbEnd0S1, vm, m, F, c),
       Adv_toEnv (F, ZERO))
```

Note that in the definition above, dimindex$(: \varsigma)$ is the HOL4 function that returns the cardinality of the type $(: \varsigma)$.

If FMAC is corrupted, the adversary is able to overwrite the key. In this case the simulator can skip querying the oracle and directly use the MITB for simulation. After the key was overwritten, the MITB can be simulated using MITB_STEP.

```
SIM mitbf f (T, Ready, vm, m, F, c)
   (EnvtoA (F, F, inp, inp_size)) =
   if inp_size ≤ dimindex (: ς) then
     (let
        s = (Ready, f (ZERO @@ inp), ZERO)
      in
        ((T, Ready, vm, m, T, s), Adv_toEnv (T, ZERO)))
   else ((T, Ready, vm, m, F, c), Adv_toEnv (T, vm))
SIM mitbf f (T, Ready, vm, m, T, s) (EnvtoA i) =
   (let
     (s_next, rdy, dig) = mitbf f (s, i)
    in
     ((T, Ready, vm, m, T, s_next), Adv_toEnv (rdy, dig)))
```

Where '@@' is the word concatenation function.

D Applications

Due to the simplicity of its design and the absence of a random number generator, the MITB is incredibly cheap to produce. Its main use is secure password storage, but it is amenable to many applications. For each of the applications below, we formulate improved resilience against host-compromise in terms of application-specific properties like password security against offline attacks or perfect forward security.

All properties we mention have been verified using off-the-shelf protocol verification tools. These tools operate in the symbolic model, where cryptographic outputs are abstracted using a term algebra. E.g., in the following examples, a MAC is a term of form $mac(k, m)$, where mac is a function symbol, and k, m themselves are terms. In the symbolic model, the adversary's deductive capabilities are modelled as a set of deduction rules or, equivalently, as a set of equations that induces an equivalence relation among terms. As usual, the properties of a

MAC are modelled by the *absence* of a deduction rule that deconstructs, i.e., allows the deduction of subterms of, terms of form $mac(k, m)$. This reflects that (a) that MACs do not expose the MACed message or key, (b) MACs are deterministic and may leak whether two equivalent messages were MACed with the same key (as there is no random value represented in this term, except for the key k) and (c) if the key is known, MACs can be verified by recomputing the term (given k and m, it is always possible for compute $mac(k, m)$, as for any other function symbol). Due to the guarantees provides by FMAC, these models can keep the key k secret, even if the attacker gains control over its host system. The relation between FMAC and the symbolic model we used remains informal. Proving the correspondence is the subject of computational soundness **rogaway+abadi**. As of now, no computational soundness result has been machine-checked. The same holds for the correctness of the verification procedures we used. (There are machine-checked formalisations of the symbolic model **scyther-proof**, [40], but they provide a much lower degree of automation.)

We used three different verification tools. Each employs a slightly different formalism to express the protocol or the security property. Hence, we refer to their respective formalisation for details (DEEPSEC [35], tamarin [37] and SAPIC [38]), and try to keep the presentation intuitive.

Secure password storage: All businesses that store password data need to secure these password databases for the case where they get stolen. To name but a few-high profile cases of stolen password databases: the Playstation Network (77M entries) in 2011, LinkedIn (6,5M entries) and Last.fm (8M entries) in 2012, Adobe (152M entries) in 2013, Ashley Madison (37M entries) in 2015, and MyHeritage (92M entries) in 2018. Some of these breaches were discovered years after they occurred, when the leaked databases started circulating on the black market. Hence it is likely, that there are more recent events that are not known yet.

Besides popular (yet not widely enough adopted) best practices like salting or key-derivation, security experts advise the introduction of a 'local parameter' to the computation, i.e. a high-entropy value used in hash computation that is stored on a separate device which provides only limited access [41]. An attacker performing brute-force attacks on a stolen password database would then be required to guess the local parameter along with the password. Among various options, including FPGAs and GPUs, hardware tokens seem to be the most promising candidate [41], however, the ability to audit these devices is a key requirement. For many commercial tokens, only a black-box audit is possible. The MITB provides the required functionality, while also providing a formally verified design, which goes beyond typical audits.

To store passwords securely, the MITB is initialised with a fresh key during set-up, and then used to compute MACs on the hashed and salted password. These MACs are stored in the password file upon registration. To validate a password, the MAC is recomputed. The hash is used to maintain compatibility with schemes where only the hashed password is submitted over the wire. The salt is used to avoid leaking whether two users have the same password, as the MAC is deterministic.

We used the recent DEEPSEC decision procedure [35] to show strong secrecy in the Dolev-Yao model. Strong secrecy is formulated as the ability to distinguish between two systems. In the first, the adversary obtains, for each secret password pw in the database, the salt s, the MITB's output on the hashed and salted secret $h(pw, s)$ and pw in the clear. Like mac, h is a function symbol, and there is no equation for obtaining a subterm of $h(pw, s)$. In the second system, the adversary again obtains s and the MITB's output on $h(pw, s)$, but a completely different pw' in the clear. The equivalence of these systems means that, even if the adversary can make a guess on pw, she cannot verify whether she was right (first system) or wrong (second system). This renders offline attacks impossible. Hence, for a password database with k entries, we show equivalence between

$$\nu k; !^k(\nu pw; \nu s; (\mathsf{out}(mac(k, h(pw, s))); \mathsf{out}(s) \mid \mathsf{out}(pw)))$$

and

$$\nu k; !^k(\nu pw; \nu s; (\mathsf{out}(mac(k, h(pw, s))); \mathsf{out}(s) \mid \nu pw'; \mathsf{out}(pw')))$$

Here, $\nu k; P$ chooses a fresh name k, akin to drawing a random key. $P \mid Q$ denotes parallel composition, i.e., processes P and Q running in parallel. $!^k P$ is a shortcut for k copies of P composed in parallel. Finally, $\mathsf{out}(t)$ outputs the term t on the public channel.

As even weak secrecy is undecidable in general [42], DEEPSEC operates in the bounded model. We thus have to set a limit for the number of passwords at the adversary's disposal, i.e., the value of k. For a database of six passwords, i.e., $k = 4$, the decision procedure gives a positive results after 90s[12].

The MITB applies even to the more elaborate password storage scheme by Almeshekah et. al, where 'ersatzpasswords' are computed from a user's password [43]. To detect database leaks, ersatzpasswords mimic the appearance of user generated passwords and raise an alarm if they are used for login. Their implementation uses the YubiHSM, which costs about \$ 650 and could be replaced by the MITB.[13]

Establishing a secure channel: Secure channels are usually established by performing key-exchange and then encrypting the communication with the resulting key. Key-exchange requires some method of authentication to ensure the key is exchanged with the intended communication partner, otherwise it would be vulnerable to a man-in-the-middle attack. The signed Diffie-Hellman key-exchange protocol is such an example, and used, e.g., in TLS and IPsec. Communication partners that share a key on the MITB can use MACs instead of

[12] Computed on a MacBook Pro with 3,1GHz Intel i7 and 16GB RAM.

[13] Pessl and Hutter managed to implement SHA-3 on an RFID token [36], citing cost estimates of about \$ 0.05. As the MITB's state machine and key storage does not fundamentally add to that, the production cost of the MITB will likely be dominated by the bus technology connecting it to its host, e.g., USB.

signatures to ensure the integrity of the key-exchange, and thus establish a channel that is confidential against active attacker. More so, confidentiality holds even if the MITB is lost after the communication session (perfect forward security). We demonstrate this for a Diffie-Hellman key exchange.

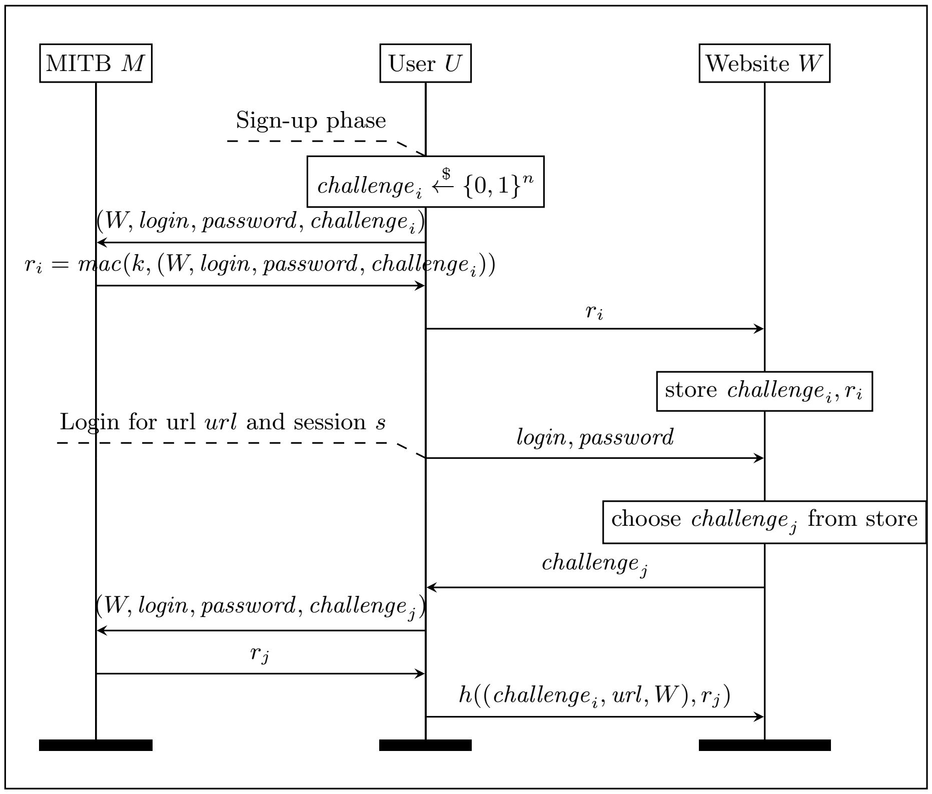
MITB M
User U
Website W
Sign-up phase
$challenge_i \xleftarrow{\$} \{0,1\}^n$
$(W, login, password, challenge_i)$
$r_i = mac(k, (W, login, password, challenge_i))$
r_i
store $challenge_i, r_i$
Login for url url and session s
$login, password$
choose $challenge_j$ from store
$challenge_j$
$(W, login, password, challenge_j)$
r_j
$h((challenge_i, url, W), r_j)$

Figure 7. U2F protocol for user U using MITB M to sign up and login on website W, simplified.

For G a suitably chosen group and g a public generator for G, two parties randomly pick non-zero elements of G, x and y, respectively. They exchange g^x and g^y. Now both parties can compute the same key $(g^x)^y = g^{xy} = (g^y)^x$. As the decisional Diffie-Hellman problem, i.e., distinguishing $g^{x,y}$ from a random group element, is widely believed to be impossible to solve, this protocol is secure against passive adversaries. To provide security against active adversaries, instead of signing the message, two parties that have managed to synchronise their keys prior to communication can also use the MITB to set up a secure channel, MACing the messages instead of signing them. Once a channel is set up, it provides very strong guarantees that go beyond the secrecy of the key:

– *perfect forward secrecy*: even if the adversary gains control over one of the MITBs, all sessions keys established prior to this event remain secret.
– *post-compromise security*: even if the adversary temporarily gains control over one of the MITBs, once the participants come together and set up a new key, future session keys will again remain secure.

We used the SAPIC/tamarin [37], [38] toolchain for protocol verification to establish both properties for an unbounded number of sessions. The proof is automated and terminates in 1516s.[12] The model is listed in Appendix E.

This scheme is limited to the communication between two communication partners, both need to agree on a key during key-setup. It thus applies to high-security scenarios only. For end-user communication, the following protocol is more realistic.

Two-factor authentication: We demonstrate that the MITB is compatible with the FIDO standard for universal 2nd factor authentication [39]. The goal is that a user can log in to a webserver with a username and password, and a so-called *second factor*, typically a piece of hardware she owns. The standard is tailored to devices that create digital signatures, however, we managed to integrate the MITB by using a method based on one-time passwords. Upon sign-up, the user generates random challenges $c_1, \ldots, c_n$ and pre-computes the responses

$$r_i = mac(k, \langle W, login, password, c_i \rangle),$$

where W is the domain name of the web server, *login* and *password* the user's password and login and $\langle \cdot \rangle$ a suitable encoding for lists. These responses can only be computed using the MITB. The users sends these pairs c_i, r_i to the server, who stores them.

The responses can be thought of as one-time passwords. To login, the server chooses an arbitrary challenge, and the user recomputes the response to this challenge. Hence, the user does not need to keep additional state.

We again used SAPIC/tamarin to establish perfect forward security and post-compromise security for authentication, i.e., the property that any successful login on the webserver was initiated by the user. We model a single server with an unbounded number of users, each running an unbounded number of sessions:

$$!P_{\mathsf{Server}} \mid !(\nu k; !P_{\mathsf{User}}).$$

The user process choses a fresh login and password. Afterwards, it can perform setup and authentication, repeatedly and in any order.

$$!P_{\mathsf{User}} := \nu\, login, password; (P_{\mathsf{usetup}} \mid P_{\mathsf{uauth}} \mid P_{\mathsf{ucorrupt}}).$$

P_{usetup} is defined as follows:

```
lock 'setup';
    in(handle);ν challenge;
    event FreshSetup(challenge);
    insert ⟨'Server', login, handle⟩,
P_Server :=

in(url); in( tls_session );
in(⟨ login ,password⟩);

in(handle); // adversary decides which
    challenge to use
lookup ⟨'F_Server', login ,handle⟩ as
    entry in
   out( fst (entry));
   in (xsignature );
   if  xsignature = h(fst( entry), url ,
       tls_session ,snd(entry)) then
     delete ⟨'F_Server', login ,handle⟩
       ;
     event Access( login , url ,
         tls_session , fst (entry))
```

```
P_uauth :=

in(url); in( tls_session );
out(⟨ login ,password⟩);
event Request( login , url, tls_session );


in (challenge);
out(h(⟨ challenge, url, tls_session ⟩ ,mac(k,
      ⟨'Server', login ,password, challenge⟩ )
      ))
```

Figure 8. Authentication process for the server (left) and the client (right).

```
        ⟨ challenge , mac(k,⟨'Server', login ,password, challenge⟩ )⟩ ;
unlock 'setup'
```

Setup is protected by a Dijkstra-style semaphore to ensure that it can never run in parallel with an authentication process. It stores a randomly chosen challenge and the corresponding response on the server's database. Both lock-/unlock and insert/lookup are SAPIC constructs extending the applied-π protocol to deal with stateful protocols such as this one. The construct insert x, y stores term y at a position x in a global, dictionary-like store. The construction lookup x as y in P else Q retrieves the value at position x at a later point in time. It reduces to P with the last value inserted at x in place of y. If no such value exists, it reduces to Q.

Figure 8 is a straight-forward modelling of the server and user authentication protocol described in Figure 7. The adversary gets to choose the url and TLS session, as well as the order, in which the server sends the challenge. The server sends the pre-recorded challenge and, in place of a signature, a hash on the challenge, the url, the TLS session and the response. Recall that the response is a MAC computed by the MITB.

If the client's host is compromised, the adversary can compute MACs, but thanks to the MITB, she does not obtain the key k itself:

```
P_ucorrupt := lock 'setup'; event Corrupt();
                  in(x); out(mac(k,x));
              unlock 'setup'
```

We show the following property:

Theorem 2 (Post-compromise and forward security). *For all traces of the process just described above, it holds that any successful login, i.e., event*

Access(*login*, *url*, *tlssession*, *challenge*), *implies* either *a genuine login request, i.e.,* Request(*login*, *url*, *tlssession*), or *that the client's host was compromised, i.e., the event* Corrupt() *occured, however, if that is the case, then this event must have occured* after *the challenge was set up, i.e., event* FreshSetup(*challenge*), *and* before *the aforementioned* Access-*event.*

Observe that the constraints on the Corrupt-events imply that (a) any challenge that is set up after the last corruption event recovers the authentication property (*post-compromise security*), and (b) if a challenge has been used for authentication, it cannot be used for future authentication, even if the user's host is compromised afterwards (*forward security*). SAPIC/tamarin proves this theorem without any user intervention or helping lemmas within 9s.[12]

E Listing for Section 8

```
theory MACedDH_PCS
begin
section{* Diffie-Hellman with MAC-in-the-Box *}

 builtins : diffie-hellman
functions : mac/2

let Initiator =
    let m1 = <'1', $I,$R, 'g' ^ ~F_ekI>
        m1m = mac(k,m1)
        m2 = <'2',$I, Y>
    in
    lock 'setup';
    lookup 'mitb_init' as k in
    ν ~F_ekI;
    out(<m1,m1m>);
    in (<m2,m2m>);
    if m2m = mac(k,m2) then
        event SessionKey($I,$R, Y^~F_ekI);
        unlock 'setup'

let Responder =
    let m1 = <'1', $I,$R, X>
        m2 = <'2',$I,'g' ^ ~F_ekR>
        m2m = mac(k,m2)
    in
    lock 'setup';
    lookup 'mitb_resp' as k in
    ν ~F_ekR;
    in(<m1,m1m>);
    if  m1m = mac(k,m1) then
        out(⟨m2,m2m⟩);
        event  SessionKey($I,$R,X^^~F_ekR);
        unlock  'setup'

let  Setup = (lock 'setup';new k;insert 'mitb_init',k;insert 'mitb_resp',k;event
     FreshSetup(); unlock 'setup')——(lock  'setup'; lookup  'mitb_init'  as  k
     in  event  Corrupt();  in(x);  out(mac(k,x));  unlock  'setup')
             |(lock 'setup'; lookup 'mitb_resp' as k in event Corrupt(); in(x);
                 out(mac(k,x)); unlock 'setup')(Setup |  Initiator  | Responder )

// Post−compromise and Perfect−Foward−Secrecy
lemma PC_PF_Secrecy:
  "∀ I R sessKey i k.
      SessionKey(I,R,sessKey) @ i ∧
      K(sessKey) @ k
       ⟹ (
      ∃ c  .  Corrupt()@c ∧c<i
      ∧ (∀ f.  FreshSetup()@f  ⟹ f <  c ∨ i <  f)
      ) "
end
```