Symmetrically and Asymmetrically Hard Cryptography

Alex Biryukov[1](✉) and Léo Perrin[2,3]

[1] University of Luxembourg, Belval, Luxembourg
alex.biryukov@uni.lu
[2] Inria, Paris, France
leo.perrin@inria.fr
[3] SnT, University of Luxembourg, Belval, Luxembourg

Abstract. The main efficiency metrics for a cryptographic primitive are its speed, its code size and its memory complexity. For a variety of reasons, many algorithms have been proposed that, instead of optimizing, try to increase one of these hardness forms.

We present for the first time a unified framework for describing the hardness of a primitive along any of these three axes: code-hardness, time-hardness and memory-hardness. This unified view allows us to present modular block cipher and sponge constructions which can have any of the three forms of hardness and can be used to build any higher level symmetric primitive: hash function, PRNG, etc.

We also formalize a new concept: *asymmetric hardness*. It creates two classes of users: common users have to compute a function with a certain hardness while users knowing a secret can compute the same function in a far cheaper way. Functions with such an asymmetric hardness can be directly used in both our modular structures, thus constructing any symmetric primitive with an asymmetric hardness. We also propose the first asymmetrically memory-hard function, Diodon.

As illustrations of our framework, we introduce Whale and Skipper. Whale is a code-hard hash function which could be used as a key derivation function and Skipper is the first asymmetrically time-hard block cipher.

Keywords: White-box cryptography · Memory hardness · Big-key encryption · Skipper · Whale · Diodon

1 Introduction

The design of cryptographic algorithms is usually a trade-off between security and efficiency. Broadly speaking, the efficiency of an algorithm is defined along three axes: time, memory and code size. Yet in some scenarios it is desirable to design primitives that are purposefully inefficient for one or several of these

The work of Léo Perrin was supported by the CORE project ACRYPT (ID C12-15-4009992) funded by the Fonds National de la Recherche, Luxembourg.

metrics. This can be done to slow down the attackers, provide different levels of service to privileged and non-privileged users, adjust cost of operation in proof-of-work schemes, etc. Primitives with different forms of computational hardness have been scattered in time and in several seemingly unrelated research/application areas. In this paper we propose a simple unifying framework which allows us to build new provably hard modes of operation and primitives.

The simplest illustration of functions designed to be time consuming to compute is that of key derivation functions (KDF). A KDF is typically built by iterating a one-way function (for example a cryptographic hash function), multiple times. Such functions are intended to prevent an adversary from brute-forcing a small set of keys (corresponding to, say 12 letter strings) by making each attempt very costly.

Time, however, is not the only form of hardness for which an artificial increase can be beneficial. Memory-hardness was one of the design goals of the winner of the Password Hashing Competition, Argon2 [12], the aim being to prevent hardware optimization of the primitive. As another example, one research direction in white-box cryptography is nowadays focusing on designing block ciphers such that the code implementing them is very large in order to prevent duplication and distribution of their functionality [7,11,16,17,22]. In this case, the aim could be to implement some form of Digital Right Management or to prevent the exfiltration of a block cipher key by malware.

Since hardness is an inherently expensive property, there are cases where a trap-door could be welcome. This is the case for the most recent weak white-box block ciphers [11,16,22]: while the white-box implementation requires a significant code size, there exists a functionally equivalent implementation which is much smaller but cannot be obtained unless a secret is known. That way, two classes of users are created: those who know the secret and can evaluate the block cipher efficiently and those who do not and thus are forced to use the code-hard implementation.

The different forms of hardness, their applications and typical instances from the literature are summarized in Table 1.

Table 1. Six types of hardness and their applications.

	Time	Memory	Code size
Applications	KDF, time-lock	Password hashing, egalitarian computing	White-box crypto, big-key encryption
Symmetrically hard functions	PBKDF2 [25]	Argon2 [12], Balloon [18]	XKEY2 [7], Whale (Sect. 5.2)
Asymmetrically hard functions	RSA-lock [30], Skipper (Sect. 5.1)	Diodon (Sect. 2.4)	White-box block ciphers [11,16,17,22]

Our Contribution. Regardless of the form of hardness, the aim of the designer of a hard primitive is to prevent an attacker from by-passing this complexity, even if the attacker is allowed significant precomputation time. Informally, a user cannot decrease the hardness of the computation below a certain threshold. Inspired by the formal definitions of hardness used in white-box cryptography, we provide for the first time a unified framework to study and design cryptographic algorithms with all forms of hardness. Our framework is easily extended to encompass *asymmetric hardness*, a form of complexity which can be bypassed provided that a secret is known.

Our approach consists in combining simple functions called *plugs* having the desired form of hardness with secure cryptographic primitives. Algorithms are then built in such a way as to retain the cryptographic security of the latter while forcing users to pay for the full hardness of the former. In fact, we provide a theoretical framework based on random oracles which reduces the hardness of the algorithms we design to that of their plugs.

Furthermore, we introduce the first asymmetrically memory-hard function, Diodon. It is a function based on `scrypt` [28] modified in such a way as to allow users knowing an RSA private key to evaluate it using a constant amount of memory. It can of course be used as a plug within our framework.

Finally, we used this approach to build a code-hard hash function called Whale and an asymmetrically time-hard block cipher called Skipper. It is impossible to design a functionally equivalent implementation of the Whale hash function which is much smaller than the basic one. On the other hand, encrypting a block with Skipper is time consuming but this time can be significantly decreased if an RSA private key is known.

Outline. First, Sect. 2 provides more details about the different forms of hardness and their current usage for both symmetric and asymmetric hardness. We also introduce the first asymmetrically memory-hard function, Diodon, in Sect. 2.4. Then, Sect. 3 presents our generic approach for dealing with all forms of hardness at once. To this end, plugs achieving all forms of hardness are introduced in the same section. We deduce practical modes of operation for building hard block ciphers and hard sponges which are described in Sect. 4. Our concrete proposals, called Skipper and Whale, are introduced in Sect. 5.

2 Enforcing Hardness

In this section, we argue that many recent ideas in symmetric cryptography can be interpreted as particular cases of a single general concept. The aim of several a priori different research areas can be seen as imposing the use of important resources for performing basic operations or in other words, *bind an operation to a specific form of hardness*. We restrict ourselves to the basic case of a well-defined function mapping each input to a unique output. It means in particular that protocols needing several rounds of communication or randomized algorithms which may return any of the many valid solution to a given problem such as HashCash (see below) are out of our scope. We also tie each function to an algorithm evaluating it: further below, functionally equivalent functions evaluated using different algorithms are considered like different functions. Thus, the "hardness" discussed is the hardness of the algorithm tied to the function.

The three main metrics for assessing the efficiency of an algorithm are its time complexity, its RAM usage and its code size. As we explain below, different lines of research in symmetric cryptography can be interpreted as investigating the design of algorithms such that one of these metrics is abnormally high and cannot be reduced while limiting the impact on the other two as much as possible.

Time-hardness is discussed in Sect. 2.1, *memory-hardness* in Sect. 2.2 and *code-hardness* in Sect. 2.3. Finally, in Sect. 2.4, we present the general notion of *asymmetric hardness.*

It is also worth mentioning that the three forms of hardness are not completely independent from one another. For example, due to the number of memory access needed in order for a function to be memory-hard, a function with this property cannot be arbitrarily fast.

2.1 Time Hardness

While the time efficiency of cryptographic primitives is usually one of the main design criteria, there are cases where the opposite is needed. That is, algorithms which can be made arbitrarily slow in a controlled fashion.

One of the most simple approaches is the one used for instance by the key derivation function PBKDF2 [25]. This function derives a cryptographic key from a salt and a password by iterating a hash function multiple times, the aim being to frustrate brute-force attacks. Indeed, while the password may be from a space small enough to be brute-forced, evaluating the key derivation function for each possible password is made infeasible by its time-hardness.

Somewhat similarly, proofs-of-work such as HashCash (used by the cryptocurrency Bitcoin [26]) consist in finding at least one of many solutions to a given problem. The hardness in this case comes from luck. Miners must find a value such that the hash of this value and the previous block satisfies the difficulty constraint. However, the subset of such valid values is sparse and thus miners have to try many random ones. Two different miners may find two different but equally valid values. Because of this randomness, such puzzles are out of our scope. In this paper, we only consider functions which are equally hard to evaluate on all possible inputs, not puzzles for which finding a solution is hard *on average.*

Furthermore, in order to mitigate the impact of adversaries with vast amount of processors at their disposal, we consider sequential time-hardness. Using a parallel computer should not help an attacker in evaluating the function much quicker. Formalizing parallel time hardness the way we do it for sequential time-hardness is left as a future work.

Overall, the goal of time-hardness is to prevent an adversary from computing a function in a time significantly smaller than the one intended. In other words, it must be impossible to compress the amount of time needed to evaluate the function on a random input.

2.2 Memory Hardness

Informally, a function is memory-hard if even an optimized implementation requires a significant amount of memory. For each evaluation, a large amount of information is written and queried throughout the computation. As a consequence, a memory-hard function cannot be extremely fast.

A function requiring large amounts of RAM for its computation prevents attacker from building ASICs filled with huge number of cores for parallel computations. This implies that memory-hard functions make good password hashing functions and proofs-of-work. One of the first to leverage this insight was the hash function `scrypt` [28] which was recently formally proved to be memory-hard in [3]. More recently several other memory-hard algorithms have been designed, such as the password hashing competition winner Argon2 [12] as well as the more recent Balloon Hashing [18] and Equihash [14]. Those can be used as building blocks to create memory-hard proofs-of-work which can offset the advantage of cryptocurrency miners using dedicated ASICs.

The idea of using memory-hard functions for general purpose computations was further explored in the context of *egalitarian computing* [13]. Similarly, *proofs of space* [5,21] are protocols which cannot be run by users if they are not able to both read and write a large amount of data. However, those are interactive protocols and not functions.

The recent research investigating memory-hardness has lead to several advances in our understanding of this property. For example, the difference between *amortized* and *peak* memory hardness was highlighted in [4].

2.3 Code Hardness

First of all, let us clarify the distinction we make between *memory* and *code*-hardness. With code-hardness, we want to increase the space needed to store information that is needed to evaluate a function on all possible inputs. However, the information itself does not depend on said input. During evaluation of the function, it is only necessary to *read* the memory in which the code is stored. In contrast, memory-hardness deals with the case where we need to store a large amount of information which depends on the function input and which is thus different during each evaluation of the function. In this case, one must be able to both read *and write* to the memory. Furthermore, in a typical code-hard function, only a small fraction of the whole information stored in the implementation is read during each call to the function. On the other hand, if a memory-hard function uses M bytes of memory, then all of those bytes will be written and read at least once.

Code-hardness is very close to what was first defined as *memory-hard white-box implementation (or weak white-box)* in the paper introducing the ASASA crypto-system [11], and later formalized under different names as (M, z)-space hardness [16,17] or as incompressibility in [22] following the more general definition from [20]. In all cases, the aim is the same: the block cipher implementation must be such that it is impossible to write a functionally equivalent implementation with a smaller code. This stands in contrast to *strong* white-box cryptography (as defined in [11]) where inverting a function given its white-box implementation should be impossible. We do not consider this case in this paper.

As was pointed out in [22], what we call code-hardness is also the goal of so-called *big-key encryption.* For instance, the XKEY2 scheme introduced in [7] achieves this goal: it uses a huge table and a nonce to derive a key of regular size (say, 128 bits) to be used in a standard encryption algorithm, e.g. a stream cipher. Bellare et al. show that even if an attacker manages to obtain half of the huge table, i.e. half of the code needed to implement the scheme, then they are still unable to compute the actual encryption key with non-negligible probability. Using our terminology, XKEY2 can be seen as a code-hard key derivation function. A more detailed analysis of the literature on code-hardness is provided in the full version of this paper [15].

The concept of *proof of storage* can also be interpreted as a particular type of code-hard protocol. Indeed, in such algorithms, challengers must prove that they have stored a given file.

2.4 Asymmetric Hardness

In this section, we discuss the concept of *asymmetric hardness* which introduces two classes of users. *Common users* evaluate a hard function but *privileged users*, who know a secret key, can evaluate a functionally equivalent function which is *not* hard. We also introduce Diodon, the first asymmetrically memory-hard function.

Asymmetric Code-Hardness. The most recent white-box block ciphers such as SPACE [16], the PuppyCipher [22] and SPNbox [17] can be seen as providing asymmetric code-hardness. Indeed, while the first aim of these algorithms is to provide regular code-hardness, referred to as "space-hardness" for the former and "incompressibility" for the latter, they both allow the construction of far more code-efficient implementation. For both SPACE and the PuppyCipher, the idea is to compute a large table containing the encryptions of the first 2^t integers with AES-128 for t in $\{8, 16, 24, 32\}$. These tables are then used as the code-hard part of the encryption which cannot be compressed because doing so would require a break of the AES. However, a user knowing the 128-bit AES key can get rid of these tables and merely recompute the entries needed on the fly, thus drastically decreasing the code-hardness of the implementation.

In fact, both constructions can be seen as structures intended to turn an asymmetrically code-hard function into an asymmetrically code-hard block cipher. In both cases, the asymmetrically code-hard function consists in the evaluation of the AES on a small input using either the secret key, in which case the implementation is not code-hard, or using only the public partial codebook which, because of its size, is code-hard.

Time-Hardness. While the asymmetry of its hardness was not insisted upon, there is a known asymmetrically time-hard function, which we call RSA-lock. It was proposed as a time-lock in [30], that is, a function whose output cannot be known before a certain date.

It is based on the RSA cryptosystem [29]. It consists in iterating squarings in an RSA modular ring: a user who does not know the prime factor decomposition of the modulus N must perform t squarings while a user who knows that $N = pq$ can first compute $e = 2^t \mod (p-1)(q-1)$ and then raise the input to the power e. If t is large enough, the second approach is much faster.

Memory-Hardness. We are not aware of any asymmetrically memory-hard function in the existing literature. Thus, we propose the first such function which we call Diodon. It is based on the ROMix function used to build `scrypt` [28] and relies on the RSA crypto-system to provide the asymmetry, much like in RSA-lock.

Diodon maps an element x of $\{0,1\}^t$ to a an element y of $\{0,1\}^u$ using RSA computations and a hash function H. It is parametrized by an RSA modulus N of size n_p, a hash function H and its input and output sizes t and u. Only privileged users know the prime factor decomposition of the RSA modulus. Finally, a parameter M is used to tune its memory-hardness while parameters L and η decide its time complexity. The computation is described in Algorithm 1 and in Fig. 1. In both, T_u denotes the function truncating a bitstring to its u bits of lowest weight.

Algorithm 1. Diodon Asymmetrically memory-hard function
Inputs: t-bit block x; RSA modulus N of n_p bits; M, L;
Output: u-bit output y

```
$V_0 = x$
for all $i \in \{1, ..., M-1\}$ do
    $V_i = V_{i-1}^{2^\eta} \mod N$
end for
$S = V_{M-1}$
for all $i \in \{0, ..., L-1\}$ do
    $j = S \mod M$
    $S = H(S, V_j)$
end for
return $T_u(S)$
```

A user without knowledge of the factorization of N must use Algorithm 1 to evaluate Diodon(x). However, if a user knows the factorization of $N = qq'$, she does not need to store the vector V. Indeed, she can simply evaluate $V_i = x^{2^{i \times \eta}} \mod N$ in constant time by first reducing $2^{i \times \eta}$ modulo $(q-1)(q'-1)$ and then raising x to the corresponding exponent. One may call her ability to access an arbitrary element of V in constant time an *RSA RAM*. Her evaluation strategy

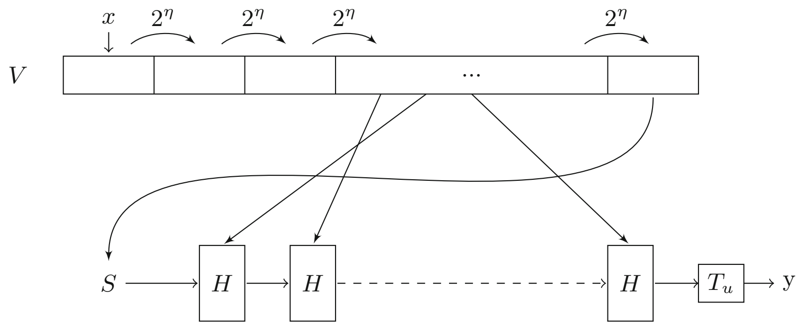

Fig. 1. The evaluation of $y = \textsc{Diodon}(x)$.

is summarized in Algorithm 2. Basic users need to perform $\eta \times M$ RSA squarings while privileged ones need to perform roughly $(L + 1) \times n_p$ of those as e and each exponent e_j is a priori of length n_p.

Algorithm 2. Diodon for privileged users
Inputs: t-bit block x; RSA factors q, q'; η; M, T;
Output: u-bit output y

```
$e = 2^{(M-1)\times\eta} \mod (q-1)(q'-1)$
$S = x^e \mod (qq')$
for all $i \in \{0, ..., L-1\}$ do
    $j = S \mod M$
    $e_j = 2^{j\times\eta} \mod (q-1)(q'-1)$
    $S = H\big(S, (x^{e_j} \mod (qq'))\big)$
end for
return $T_u(S)$
```

Let us first consider the simplest parameters, that is $\eta = n_p$ and $L = M$. In this case, the time complexity for both users is comparable as each class of user needs to perform M RSA encryptions.[1] Furthermore, without the knowledge of the secret key, the computation is essentially `scrypt` ROMix function which was shown to be optimally linearly memory hard [3], keeping time-memory product constant. It means that it is either necessary to store all the values V_i for the computation or any algorithm that saves a factor f in memory will have to pay the same factor in time.

The choice of the parameters η, M, L and n_p has a significant impact on the time efficiency of Diodon. It is investigated in the full version of this paper [15] where we also propose some concrete instances.

[1] The privileged user still has a small advantage in this case since knowing the prime factor decomposition of the modulus allows the use of the Chinese Remainder Theorem (DRT) to speed the exponentiation.

3 A Generic Framework

As we have seen, all the techniques presented in the sections above are intended to enforce some form of computational hardness. In this section, we present a unified framework for building any symmetric algorithm with some form of guaranteed hardness. We describe our aim in Sect. 3.1 and our design strategy with the generic hardness definition it is based on in Sect. 3.2. Our constructions need small functions with the intended hardness type to be bootstrapped. We provide examples of those in Sect. 3.3.

3.1 Design Strategy

Our aim is to design a general approach to build any cryptographic primitive with any form of hardness. To achieve this, we will build modes of operations allowing us to combine secure cryptographic primitives, such as the AES block cipher [19] or the Keccak sponge [9], with small functions called *plugs*. These plugs are simple functions with the desired form of hardness.

Our modes of operations, which are presented in Sect. 4, are all based on the same principle: ensuring that enough plug calls with an unpredictable input are performed so as to guarantee that, with overwhelming probability, an adversary cannot bypass the hardness of all plug evaluations. This ensures that the full complexity of a plug evaluation is paid at least once.

Indeed, regardless of the hardness form considered, the strategy of a generic adversary will always be the same. Provided that the plugs are indeed hard to evaluate, the only strategy allowing an adversary to bypass their hardness consists in storing a (feasibly) large number of plug outputs in a database and then querying those. If 2^p outputs of a plug $P : \{0,1\}^t \rightarrow \{0,1\}^v$ have been stored, the plug can be evaluated successfully *without paying for its hardness* with probability 2^{p-t}.

An alternative strategy using the same space consists in storing $2^p(v/d)$ partial outputs of length d. In this case, the success probability becomes $2^{p-t}(v/d) \times 2^{d-v}$: the input is partially known with a higher probability $2^{p-t}(v/d)$ but $(v-d)$ bits of the output remain to be guessed. This method is $(v/d) \times 2^{d-v}$ more efficient than the basic one but, for $1 \leq d < v$, this quantity is always smaller than one. The strategy consisting in storing full outputs is therefore the best.

However, if the output size of the plug is small enough, it might be more efficient for the adversary to directly guess the whole output. The probability that an adversary merely guessing the output of the plug gets it right[2] is 2^{-v}.

Our aim is therefore to guard our constructions from the adversary defined below. Protecting our structure against those is sufficient to reduce their hardness to that of the plug they use. Recall that we tie each function to a specific algorithm evaluating it. Thus, the hardness is actually that of the corresponding algorithm.

[2] This is only true under the assumption that the output of the plug is uniformly distributed. As we will see later, the plug can still be used if it is not the case.

Definition 1 (2^p-adversary). *Let f be a time-, memory- or code-hard function. A 2^p-adversary is an adversary trying to generate a function f' which does not have the hardness of f but which does not have access to more memory than needed to store 2^p outputs of f.*

A 2^p-adversary can perform more than 2^p calls to the function it is trying to approximate when generating f', although f' itself cannot have access to more than 2^p of those. Still, f' can perform additional computations using the information stored during its generation.

However, the computational power of this adversary is not unbounded. More precisely we consider only 2^p-adversaries which cannot perform more than 2^{100} operations. This means for example that recovering a 128-bit AES key is out of their reach. We are not interested in guarding against unrealistic, computationally unbounded adversaries.

Similarly, the complexity of the plug approximation along the other axes cannot be arbitrarily high. Consider a code-hard function f which uses a table s of M bits and an approximation f' which works as follow:

1. Evaluate and store several values of $f(i)$ for random inputs i using far fewer than M bits of storage.
2. When given a random input x:
   (a) brute-force all possible values of s, i.e. 2^M values using M bits of RAM to store each candidate table s';
   (b) try each candidate s' to see if they yield $f'(i) = f(i)$ using the values of $f(i)$ previously stored;
   (c) once the right s has been found, use it to evaluate $f'(x)$, return it and then erase s'.

Such an approximation f' is not code-hard, it merely needs enough space to store $f(i)$ for several i. However, it is memory-hard since it needs to store the same amount of information as is stored in the implementation of f, although the storage is in RAM rather than code. Much more importantly, its time-hardness is astronomical: M will typically be in the millions, if not the billions, meaning that enumerating all 2^M candidates s' is utterly impossible.

We explicitly do not consider such extreme trade-offs: in our cases, the hardness of f remains practical—though expensive—, meaning that a 2^p-adversary trying to bypass its hardness is trying to optimize an already usable implementation. It does not make sense for her to trade code-hardness for a wildly impractical time-hardness.

3.2 Theoretical Framework

Generic Symmetric Hardness. We are now ready to formally define hardness. We use a generalization to all forms of hardness of the incompressibility notion from [22].

Definition 2 (R**-hardness).** *We say that a function* $f : \mathcal{X} \rightarrow \mathcal{Y}$ *is* R-hard against 2^p-adversaries *for some tuple* $\mathsf{R} = (\rho, u, \epsilon(p))$ *with* $\rho \in \{\mathit{Time}, \mathit{Code}, \mathit{RAM}\}$ *if evaluating the function* f *using less than* u *units of the resource* ρ *and at most* 2^p *units of storage is possible only with probability* $\epsilon(p)$*. More formally, the probability for a* 2^p*-adversary to win the* efficient approximation game*, which is described below, must be upper-bounded by* $\epsilon(p)$*.*

1. *The challenger chooses a function* f *from a predefined set of functions requiring more than* u *units of* ρ *to be evaluated.*
2. *The challenger sends* f *to the adversary.*
3. *The adversary computes an approximation* f' *of* f *which, unlike* f*, can be computed using less than* u *units of the resource* ρ*.*
4. *The challenger picks an input* x *of* $\mathcal{X}$ *uniformly at random and sends it to the adversary.*
5. *The adversary wins if* $f'(x) = f(x)$*.*

This game is also represented in Fig. 2. The approximation f' *computed by the adversary must be evaluated using significantly less than* u *units of the resource* ρ*, although the precomputation may have been more expensive.*

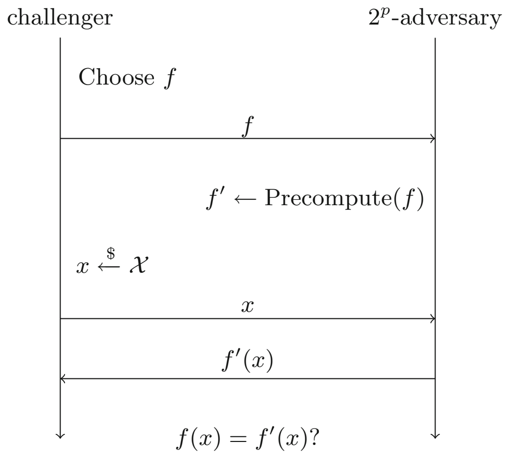
challenger
2^p-adversary
Choose f
f
$f' \leftarrow \text{Precompute}(f)$
$x \xleftarrow{\$} \mathcal{X}$
x
$f'(x)$
$f(x) = f'(x)?$

Fig. 2. The game corresponding to the definition of $(\rho, u, \epsilon(p))$-hardness against 2^p-adversaries.

In order for this definition to be relevant, the power of the adversary must be estimated. For example, preventing attacks from 2^{512}-adversaries would most definitely be over engineering and, conversely, preventing attacks from 2^{20}-adversaries would be useless since such precomputation is always feasible.

Our definition is not the strongest in the sense that it does not encompass e.g. "strong space-hardness" [16]. This definition of code-hardness aims at preventing the attacker from encrypting a plaintext *of their choosing*, a far stronger requirement than preventing the encryption of a *random* plaintext.

In the efficient approximation game described above, f' must be less hard than f along the appropriate axis. For example, if f is code-hard then the code implementing f' must be significantly smaller than that implementing f, meaning that the game corresponding to code-hardness when f is an encryption algorithm is essentially the same as the one used in the definition of encryption incompressibility [22]. Indeed, the computation of f' and its use by the adversary corresponds in this case to the computation of the leakage function on the secret large table and its use by the adversary to approximate the original table.

In the case of code-hardness the maximum code size of the implementation of f' must coincide with the power of the 2^p-adversary. Indeed, the implementation of the approximation f' needs at least enough space to store 2^p outputs of the plug.

For time-hardness, the time is measured in number of simple operations. For example, if a function requires evaluating a hash function t times, the unit of time is a hash computation. Memory- and code-hardness are measured in bytes. In our examples, code-hardness is achieved not by using programs with a complex logic but by forcing them to include large and incompressible tables. Much like in most recent white-box block ciphers, the size of the logic of the program is considered to be negligible and the size of the implementation is reduced to that of its incompressible tables.

A function which is easy to compute on a subset of its domain or whose output is not uniformly distributed in its range can still be (ρ, u, ϵ)-hard as such limitations can be taken into account by modifying the ϵ factor. For example, if a function is easy to evaluate on a fraction f of its domain then this limitation is simply captured by the fact that $\epsilon \geq 1/f$.

Generic Asymmetric Hardness. This generic definition is easily generalized to encompass asymmetric hardness.

Definition 3 (Asymmetric R-hardness). *We say that a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ is* asymmetrically R-hard against 2^p-adversaries *for some tuple* $\mathsf{R} = (\rho, u, \epsilon(p))$ *with* $\rho \in \{\mathit{Time}, \mathit{Code}, \mathit{RAM}\}$ *if it is impossible for a 2^p-adversary to win the approximation game of Definition 2 with probability higher than $\epsilon(p)$, unless a secret K is known.*

If this secret is known then it is possible to evaluate another function f_K which is is functionally equivalent to f but does not have its hardness.

An immediate consequence of this definition is that extracting the secret key K from the description of f must be computationally infeasible. Otherwise, the adversary could simply recover K during the precomputation step, use f_K as their approximation and then win the approximation game with probability 1. This observation is reminiscent of the *unbreakability* notion presented in [20].

White-box block ciphers are simple example of asymmetrically code-hard functions. This concept can also be linked to the *proof of work or knowledge* presented in [6]. It is a proof of work where a solution can be found in a more efficient way if a secret is known.

Asymmetric hardness is a different notion from public key encryption. Indeed, in the latter case, the whole decryption functionality is secret. In our case, the functionality is public. What is secret is a method to evaluate it efficiently.

A Counter-Example. The inversion of a one-way function may seem like a natural example of a time-hard function. However, as described below, it may not satisfy the requirements of our definition of $(\mathsf{Time}, u, \epsilon(p))$-hardness.

Let $h : \{0,1\}^{50} \rightarrow \{0,1\}^{50}$ be the function mapping a 50-bit string to the first 50 bits of their SHA-256 digest and let f be the inverse of h. In other words, f returns a preimage of a given digest. This function may seem time-hard at first glance as SHA-256 is preimage resistant. More specifically, it might be expected to be about $(\mathsf{Time}, 2^{50}, 2^{-20})$-hard against a 2^{30}-adversary. However, as is well known, such constructions can be attacked using Hellman's tradeoff [24] in the form of rainbow-tables allowing an attacker to recover a preimage in far less time at the cost of significant but practical pre-computation and storage. If M is the size of this table and T is the time complexity of an inversion using this table then it must hold that $MT^2 = N^2$ where $N = 2^{50}$ in our case. The failure of f to be time-hard in the sense of Definition 2 can be seen in the following strategy to win the approximation game.

1. The challenger chooses a secure hash function (SHA-256) and sends its description to the adversary.
2. The 2^{30}-adversary precomputes Hellman-type rainbow tables with in total 2^{30} entries using about 2^{50} calls to h. This adversary chooses $M = 2^{30}$ and $T = N/\sqrt{M} = 2^{35}$.
3. The challenger chooses a random value $x \in \{0,1\}^{50}$ and sends it to the adversary.
4. With high probability, the adversary computes $f(x)$ using their precomputed table in time $T = 2^{35}$ which is 2^{15} times smaller than the time needed for brute-force.

Thus, such a function is not time-hard in the sense of Definition 2.

3.3 Examples of Plugs

As our modes rely on smaller R-hard function to achieve their goal, we describe an array of such components, one for each hardness goal. A summary of all the plugs we describe, along with what we consider to be their hardness against 2^p-adversaries, is given in Table 2.

While we provide an intuition on why we assume these plugs to have the hardnesses we claim, we do not prove that it is the case.

If the output it is too large to be used in a higher level construction then it is possible to truncate it to v bits. If we denote T_v the function discarding all but the first v bits of its input and if P is a plug with a t-bit input which is $(\rho, u, \epsilon(p))$-hard against 2^p-adversaries, then $x \mapsto T_m(P(x))$ is $(\rho, u, \max(\epsilon(p), 2^{p-t}))$-hard against 2^p-adversaries. Overall, the probability of success of an approximation

Table 2. Possible plugs, i.e. sub-components for our constructions which we assume to be R-hard against 2^p-adversaries.

Hardness	Symmetric	Asymmetric
Time	IterHash^t_η $(\mathsf{Time}, \eta, 2^{p-t})$	RSAlock^t_η $(\mathsf{Time}, \eta, 2^{p-t})$
Memory	Argon2 $(\mathsf{RAM}, M/5, 2^{p-t})$	Diodon $(\mathsf{RAM}, M/10, 2^{p-t})$
Code	BigLUT^t_v $(\mathsf{Code}, 2^p, 2^{p-t})$	BcCounter^t_v $(\mathsf{Code}, 2^p, 2^{p-t})$

made by a 2^p-adversary of a plug mapping t to v bits is lower-bounded by $\max(2^{-v}, 2^{p-t})$ if its output is uniformly distributed in its range.

Time-Hard Plug. This hardness has been considered in previous works for instance in the context of key stretching and key derivation or for time-lock encryption. In fact, the constructions proposed for each use case can be used to provide time-hardness and asymmetric time-hardness respectively.

Symmetric Hardness. IterHash^t_η iterates a t-bit hash function on a t-bit input block η times where η must be much smaller than $2^{t/2}$ to avoid issues related to the presence of cycles in the functional graph of the hash function. If we denote by H the hash function used, then $\text{IterHash}^t_\eta(x) = H^\eta(x)$. Evaluating this function requires at least η hash function calls and, provided that the hash function iterated is cryptographically secure, it is impossible for an adversary to guess what the output is after η iterations with probability higher than $2^{-t/2}$.

We consider that this function is $(\mathsf{Time}, \eta, 2^{p-t})$-hard against 2^p-adversaries, as long as $p \ll t/2$.

Asymmetric Hardness. RSAlock^t_η is a function performing η squaring in a RSA modular ring of size $N = qq' \approx 2^t$, where q and q' are secret primes. Using these notations, $\text{RSAlock}^t_\eta(x) = x^{2^\eta} \mod N$. The common user therefore needs to perform η squarings in the modular ring.

However, a user who knows the prime decomposition of the RSA modulo can first compute $e = 2^\eta \mod (q-1)(q'-1)$ and thus compute $\text{RSAlock}^t_\eta(x) = x^e \mod N$. Furthermore, such a user can also use the Chinese remainder theorem to further speed up the computation which increases their advantage over common users. Thus, as long as $t > n$, the privileged user has an advantage over the common. We consider that RSAlock^t_η is asymmetrically $(\mathsf{Time}, \eta, 2^{p-t})$-hard against 2^p-adversaries.

Code-Hard Plug. As explained in Sect. 2.3, the main goals of code-hardness are white-box and big-key encryption. The structures used for both purposes rely on the same building block, namely a large look-up table where the entries are chosen uniformly at random or as the encryption of small integers. The former, BigLUT^t_v, is code-hard. The latter, BcCounter^t_v, is asymmetrically code-hard.

Furthermore, an identical heuristic can be applied to both of them to increase the input size of the plug while retaining a practical code size. It is described at the end of this section.

Symmetric Hardness. BigLUT_v^t uses a table K consisting in 2^t entries, each being a v-bit integer picked uniformly at random. Evaluating BigLUT_v^t then consists simply in querying this table: BigLUT_v^t is the function mapping a t-bit integer x to the v-bit integer $K[x]$.

This function is $(\mathsf{Code}, 2^p, 2^{p-t})$-hard against 2^p-adversaries. Indeed, an adversary who has access to 2^p outputs of the function cannot evaluate it efficiently on a random input with probability more than 2^{p-t}. Simply guessing the output succeeds with probability 2^{-v} which is usually much smaller than 2^{p-t}. Thus, we consider that BigLUT_v^t is $(\mathsf{Code}, 2^p, 2^{p-t})$-hard against 2^p-adversaries.

Asymmetric Hardness. BcCounter_v^t is the function mapping a t-bit integer x to the v-bit block $E_k(0^{v-t}||x)$, where E_k is a v-bit block cipher with a secret key k of length at least v. A common user would be given the codebook of this function as a table of 2^t integers while a privileged user would use the secret key k to evaluate this function.

The hardness of BcCounter_v^t is the same as that of BigLUT_v^t for a common user. The contrary would imply the existence of a distinguisher for the block cipher, which we assume does not exist. However, a privileged user with knowledge of the secret key used to build the table can bypass this complexity.

Furthermore, as the key size is at least as big as the block size in modern block ciphers, an adversary guessing the key is not more efficient than one who merely guesses the output of the cipher. Thus, we consider that BigLUT_v^t is asymmetrically $(\mathsf{Code}, 2^p, 2^{p-t})$-hard.

Increasing the Input Size. Both BigLUT_v^t and BcCounter_v^t have a low input size and leave a fairly high success probability for an attacker trying to win the efficient approximation game without using a lot of resource. An easy way to work around this limitation is to use $\ell > 1$ distinct instances of a given function in parallel and XOR their outputs. For example, $x \mapsto f(x)$ where

$$f(x_0||...||x_{\ell-1}) \;=\; \oplus_{i=0}^{\ell-1} E_k(\text{byte}(i)||0^{n-t-8}||x_i)$$

and where $\text{byte}(i)$ denotes the 8-bit representation of the integer i combines ℓ different instances of BcCounter_v^t. We consider that, against 2^p-adversaries, it is asymmetrically $\big(\mathsf{Code}, 2^p, \max(2^{p-v}, (2^{p-t}/\ell)^\ell)\big)$-hard.

Indeed, an attacker could store $2^p/\ell$ entries of each of the ℓ distinct tables, in which case they can evaluate the whole function if and only if all the table entries they need are among those they know. This happens with probability $(2^{p-t}/\ell)^\ell$. Alternatively, they could store the output of the whole function for about 2^p values of the complete input. In that case, they can evaluate the function if and only if the whole input is one that was precomputed, which happens with probability 2^{p-v}. We assume that there is not better attack for a 2^p-adversary than the ones we just described, hence the hardness we claimed.

It is also possible to use a white-box block cipher as an asymmetrically code-hard function as this complexity is precisely the one they are designed to achieve.

Memory-Hard Plug. Definition 2, when applied to the case of memory hardness, corresponds to functions for which a reduction in the memory usage is either impossible or extremely costly in terms of code or time complexity. We are not aware of any function satisfying the first requirement but, on the other hand, there are many functions for which a decrease in memory is bound to cause a quantifiable increase in time complexity. These are the functions we consider here.

Symmetric Hardness. Several recent functions are intended to provide memory-hardness. The main motivation was the Password Hashing Competition (PHC) which favored candidates enforcing memory-hardness to thwart the attacks of adversaries using ASICs to speed up password cracking.

The winner of the PHC competition, Argon2 [12], uses M bytes of memory to hash a password, where M can be chosen by the user. It was designed so that an adversary trying to use less than $M/5$ bytes of memory would have to pay a significant increase in time-hardness. Using our definition, if t is the size of a digest (this quantity can also be chosen by the user) and v is the size of the input, then Argon2 is about $(\mathsf{RAM}, M/5, 2^{p-t})$-hard against 2^p-adversaries as long as enough passes are used to prevent ranking and sandwich attacks [1,2] and as long as $2^{p-t} > 2^{-v}$.

The construction of memory-hard functions is a very recent topic. Only a few such functions are known, which is why Argon2 and Diodon are far more complex than the other plugs proposed in this section. It is an interesting research problem to build a simpler memory hard function with the relaxed constraint that it might be cheap to compute on a part of its domain, a flaw which would easily be factored into the $\epsilon(p)$ probability.

Asymmetric Hardness. The only asymmetrically memory-hard function we are aware of is the one we introduced in Sect. 2.4, Diodon. Because of its proximity with `scrypt`, we claim that it is $(\mathsf{RAM}, M/10, 2^{-u})$-hard for basic users [3]. In other words, we consider that Diodon is asymmetrically $(\mathsf{RAM}, M/10, 2^{p-t})$-hard — under the assumption, as for Argon2, that $2^{p-t} > 2^{-u}$. Note however that [3] guarantees only linear penalties if attacker trades memory for time, while modern memory-hard functions like Argon2 provide superpolynomial penalties. This leads us to the following open problem.

Open Problem 1. *Is it possible to build an asymmetrically memory-hard function for which the time-memory tradeoff is superpolynomial, i.e. such that dividing its memory usage by a factor α must mutiply its execution time by α^d for some $d > 1$?*

4 Modes of Operations for Building Hard Primitives

As said above, our strategy is to combine hard plugs with secure cryptographic primitives in such a way that the input of the plugs are randomized and that enough such calls are performed to ensure that at least one plug evaluation was hard with a high enough probability. The method we use is nicknamed *plug-then-randomize*. It is formalized in Sect. 4.1. Then, the block cipher and the sponge mode of operation based on it are introduced respectively in Sects. 4.2 and 4.3.

Unfortunately, our security arguments are not as formal as those used in the area of provable security. Our main issue is that we are not trying to prevent the adversary from recovering a secret: in fact, there is none in the case of symmetric hardness! Furthermore, since an (inefficient) implementation is public, we cannot try to prevent the attacker from distinguishing the function from an ideal one. It is our hope that researchers from the provable security community will suggest directions for new and more formal arguments.

4.1 Plug-Then-Randomize

Definition 4 (Plugged Function). *Let $P : \{0,1\}^t \rightarrow \{0,1\}^v$ be a plug and let $F : \{0,1\}^n \rightarrow \{0,1\}^n$ be a function, where $t + v \leq n$. The* plugged function *$(F \cdot P) : \{0,1\}^n \rightarrow \{0,1\}^n$ maps $x = x_t||x_v||x'$ with $|x_t| = t$, $|x_v| = v$ and $|x'| = m - t - v$ to y defined by:*

$$(F \cdot P)(x_t \mid\mid x_v \mid\mid x') = y = F\left(x_t \mid\mid x_v \oplus P(x_t) \mid\mid x'\right).$$

This computation is summarized in Fig. 3.

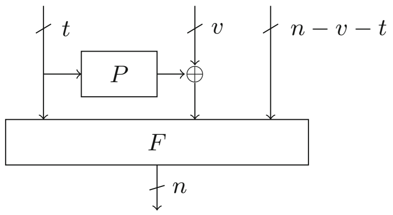

Fig. 3. Evaluating the plugged function $(F \cdot P)$.

Lemma 1 (Plugged Function Hardness). *If $P : \{0,1\}^t \rightarrow \{0,1\}^v$ is a plug $(\rho, u, \epsilon(p))$-hard against 2^p-adversaries and if $F : \{0,1\}^n \rightarrow \{0,1\}^n$ is a public random (permutation) oracle then the plugged function $(F \cdot P)$ is $(\rho, u, \epsilon(p))$-hard.*

Proof. First, the adversary could try and store 2^p outputs of $(F \cdot P)$. However, such an approximation would work only with probability $2^{p-n} < 2^{p-v} \leq \epsilon$, so that it is less successful than an approximation based on an approximation of the plug.

Without knowledge of the full input of F, it is impossible to predict its output because F is a random (permutation) oracle. Therefore, we simply need to show that the function $\mathcal{F}_P$ mapping (x, y, z) of $\{0,1\}^t \times \{0,1\}^v \times \{0,1\}^{n-t-v}$ to $(x, y \oplus P(x), z)$ is as hard as P itself.

By contradiction, suppose that there is an adversary $\mathcal{A}$ capable of winning the approximation game for $\mathcal{F}_P$. That is, $\mathcal{A}$ can compute an approximation $\mathcal{F}'_P$ of $\mathcal{F}_P$ using less than u units of the resource ρ which works with probability strictly higher than $\epsilon(p)$. Then $\mathcal{A}$ can win the approximation game for P itself as follows. When given P, $\mathcal{A}$ computes the approximation $\mathcal{F}'_P$ of the corresponding function $\mathcal{F}_P$. Then, when given a random input x of P of length t, $\mathcal{A}$ concatenates it with random bitstrings y and z of length v and $n-t-v$ respectively. The output of P is then approximated as the v center bits of $\mathcal{F}'_P(x||y||z) \oplus (x||y||z) = 0^t||P(x)||0^{n-t-v}$. Thus, $\mathcal{A}$ can violate the $(\rho, u, \epsilon(p))$-hardness of P.

We deduce that if P is $(\rho, u, \epsilon(p))$-hard, then so is $\mathcal{F}_P$ and thus $(F \cdot P)$. $\square$

Using this lemma, we can prove the following theorem which will play a key role in justifying the R-hardness of our later constructions.

Theorem 1 (Iterated Plugged Function Hardness). *Let F_i, $i < r$ be a family of r independent random oracles (or random permutation oracles) mapping n bits to n. Let $P : \{0,1\}^t \to \{0,1\}^v$ with $t+v \leq n$ be a plug $(\rho, u, \epsilon(p))$-hard against 2^p-adversaries. Then the function $f : \{0,1\}^n \to \{0,1\}^n$ defined by*

$$f : x \mapsto \big((F_{r-1} \cdot P) \circ ... \circ (F_0 \cdot P)\big)(x)$$

is $(\rho, u, \max(\epsilon(p)^r, 2^{p-n}))$-hard against 2^p-adversaries.

Proof. We denote f_i the function defined by $f : x \mapsto \big((F_{i-1} \cdot P) \circ ... \circ (F_0 \cdot P)\big)(x)$, so that $f = f_r$. We proceed by induction on the number of rounds i, our induction hypothesis being that the theorem holds for $r \leq i$.

Initialization. If $i = 1$ i.e. for $f_1 = (F \cdot P)$, Lemma 1 tells us that this function is (ρ, u, ϵ)-hard. As $\epsilon \geq 2^{p-v} > 2^{p-n}$, the induction holds for $i = 1$.

Inductive Step. Suppose that the theorem holds for i rounds. The attack based on pre-querying 2^p outputs of f_{i+1} and then approximating f_{i+1} using the content of this table would still work. Thus, if $\epsilon^{i+1} \leq 2^{n-p}$ then this strategy is the optimal one. Suppose now that $\epsilon^{i+1} > 2^{n-p}$, which also implies that $\epsilon^i > 2^{n-p}$.

As F_{i+1} is a random (permutation) oracle, the only way to evaluate the output of f_{i+1} is to first evaluate f_i and then to evaluate $(F_{i+1} \cdot P)$. The existence of another efficient computation method would violate the assumption that F_{i+1} is a random oracle.

Thus, the adversary needs first to evaluate f_i and then $(F_{i+1} \cdot P)$. Let f'_j be an approximation of the function f_j computed by a 2^p-adversary and let g_j be an approximation of $(F_j \cdot F)$ computed by the same adversary. The probability of the successful evaluation of f'_{i+1} is:

$$P\left[f'_{i+1}(x) = f_{i+1}(x),\ x \xleftarrow{\$} \{0,1\}^n\right]$$
$$= P\left[g_{i+1}(y) = (F_{i+1} \cdot P)(y) \mid y = f_i(x)\right]$$
$$\times P\left[f'_i(x) = f_i(x), x \xleftarrow{\$} \{0,1\}^n\right].$$

On the other hand, the first term is equal to

$$\begin{aligned} P\left[g_{i+1}(y) = (F_{i+1} \cdot P)(y) \mid y = f_i(x)\right] \\ = P\left[g_{i+1}(y) = (F_{i+1} \cdot P)(y), y \xleftarrow{\$} \{0,1\}^n\right] \end{aligned} \quad (1)$$

which, because of Lemma 1, is at most equal to ϵ.

Equation (1) is true. Were it not the case, then F_{i+1} would not be behaving like a random oracle. Indeed, $y = f_i(x)$ is the output of a sequence of random oracle calls sandwiched with simple bijections consisting in the plug calls that are independent from said oracle. Therefore, since x is picked uniformly at random, y must take any value with equal probability. Furthermore, the events $f_i(x) = y$ and $g_{i+1}(y) = (F_{i+1} \cdot P)(y)$ are independent: the latter depends only on the last random (permutation) oracle F_{i+1} while the former depends on all other random (permutation) oracles. As a consequence, the probability that $f'_{i+1}(x) = f_i(x)$ for x picked uniformly at random and for any approximation f'_{i+1} obtained by a 2^p-adversary is upper-bounded by ϵ^{i+1}. $\square$

4.2 Hard Block Cipher Mode (HBC)

Let E_k be a block cipher operating on n-bit blocks using a key of length $\kappa \geq n$. Let P be a plug $(\rho, u, \epsilon(p))$-hard against 2^p-adversaries. The HBC mode of operation iterates these two elements to create an n-bit block cipher with a κ-bit secret key which is $(\rho, u, \max(\epsilon(p)^r, 2^{p-n}))$-hard against 2^p-adversaries. This construction, when keyed by the κ-bit key k, is the permutation HBC$[E_k, P, r]$ which transforms an n-bit input x as described in Algorithm 3. This process is also summarized in Fig. 4. Below, we describe the hardness (Theorem 2) such an HBC instance achieves. We also reiterate that if an asymmetrically hard plug is used then the block cipher thus built is also asymmetrically hard.

Our proof is in the ideal cipher model, a rather heavy handed assumption. We leave as future work to prove the hardness of this mode of operation in simpler settings.

The role of the round counter XOR in the key is merely to make the block cipher calls independent from one another. If the block cipher had a tweak, these counter additions could be replaced by the use of the counter as a tweak with a fixed key. It is possible to use block ciphers which are not secure in the related-key setting and still retain the properties of HBC by replacing the keys $k \oplus i$ by the outputs of a key derivation function.

Theorem 2 (Hardness of HBC). *If the block cipher E_k used as a component of HBC$[E_k, P, r]$ is an ideal block cipher and if the plug P is $(\rho, u, \epsilon(p))$-hard against 2^p-adversaries, then the block cipher HBC$[E_k, P, r]$ is*

Algorithm 3. HBC$[E_k, P, r]$ encryption
Inputs: n-bit plaintext x; κ-bit key k
Output: n-bit ciphertext y

```
$y \leftarrow E_k(x)$
**for all** $i \in \{1, ..., r\}$ **do**
    $y_t \mid\mid y_{n-t} \leftarrow y$, where $|y_t| = t$ and $|y_{n-t}| = n - t$
    $y_{n-t} \leftarrow y_{n-t} \oplus P(y_t)$
    $y \leftarrow E_{k \oplus i}(y_t || y_{n-t})$
**end for**
**return** $y$
```

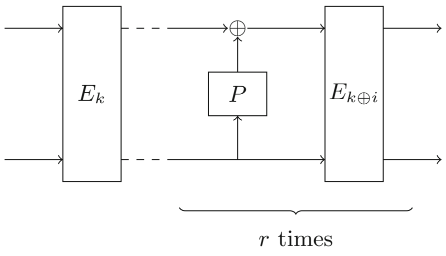

Fig. 4. The HBC block cipher mode.

$$\big(\rho, u, \max(\epsilon(p)^r, 2^{p-n})\big) \text{-}hard\ against\ 2^p\text{-}adversaries.$$

Proof. As E_k is an ideal cipher, E_k and $E_{k\oplus i}$ act like two independent random permutation oracles. As a consequence, Theorem 1 immediately gives us the theorem. □

We used the HBC structure to build an asymmetrically time-hard block cipher, SKIPPER, which we describe in Sect. 5.1.

4.3 Hard Sponge Mode (HSp)

The sponge construction was introduced by Bertoni et al. as a possible method to build a hash function [8]. They used it to design Keccak [9] which later won the SHA-3 competition. It is a versatile structure which can be used to implement hash functions, stream ciphers, message authentication codes (MAC), authenticated ciphers as described in [10], pseudo-random number generators (PRNG) and key derivation functions (KDF) as explained for example in [23]. In this section, we first provide a brief reminder on the sponge construction Then, we show how plugs can be combined with secure sponge transformation to build R-hard sponges, thus providing R-hard hash function, MAC, etc.

The Sponge Construction. A sponge construction uses an n-bit public permutation g and is parametrized by its capacity c and its rate r which are such

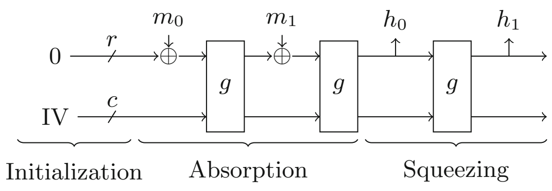

Fig. 5. A sponge-based hash function.

that $r + c = n$. This information is sufficient to build a hash function, as illustrated in Fig. 5. The two higher level operations provided by a sponge object parametrized by the function g, the rate r and the capacity c are listed below.

- **Absorption.** The r-bit block m_i of the padded message m is xored into the first r bits of the internal state of the sponge and the function g is applied.
- **Squeezing.** The first r bits of the internal state are output and the function g is applied on the internal state.

The internal state of the sponge obviously needs to be initialized. It can be set to a fixed string to create a hash function. However, if the initial value is a secret key, we obtain a MAC. Similarly, if the initial value is a secret key/initialization pair, we can generate a pseudo-random keystream by iterating the squeezing operation.

As explained in [10], this structure can be modified to allow single-pass authenticated encryption. This is achieved by using the sponge object to generate a stream with the added modification that, between the generation of the r-bit keystream block and the application of g to the internal state, the r-bit block of padded message is xor-ed into the internal state, just like during the absorption phase. In this case, there is no distinction between the absorption and squeezing phase. Once the whole padded message has been absorbed and encrypted using the keystream, the sponge object is squeezed to obtain the tag.

Finally, a sponge object can also be used to build a KDF or a PRNG using a similar strategy in both cases, as proposed in [23]. The general principle is to absorb the output of the low-entropy source and follow the absorption of each block by many iterations of $x \mapsto 0^r || T_c(g(x))$ on the internal state, where $T_c(x)$ is equal to the last c bits of x. Setting the first r bits of the internal state to zero prevents the inversion of the update function.

Sponge construction are known to be secure as long as the public update function g has no *structural distinguishers* such as high probability differentials or linear approximation with a high bias.

The main advantages of the sponge structure are its simplicity and its versatility. It is simple because it only needs a public permutation and it is versatile because all symmetric primitives except block ciphers can be built from it with very little overhead. As we will show below, the fact that its internal state is larger than that of a usual block cipher also means that attacks based on precomputations are far weaker.

4.4 The HSp Mode and Its Hardness

Given that a sponge object is fully defined by its rate r, capacity c and public update function g, we intuitively expect that building a R-hard sponge object can be reduced to building a R-hard update function. As stated in the theorem below, this intuition is correct provided that the family of functions $g_k : \{0,1\}^c \rightarrow \{0,1\}^c$ indexed by $k \in \{0,1\}^r$ and defined as the capacity bits of $g(x||k)$ is assumed to be a family of independent random oracles.

We call HSp the mode of operation described in this section. It is superficially similar to a round of the HBC block cipher mode.

An update function g can be made R-hard using the R-hardness of a plug $P : \{0,1\}^t \rightarrow \{0,1\}^v$ to obtain a new update function $(g \cdot P)$ as described in Algorithm 4.

Algorithm 4. $(g \cdot P)$ sponge transformation
Inputs: n-bit block x;
Output: n-bit block y

```
$x_t \mid\mid x_v \mid\mid x' \leftarrow x$, where $|x_t| = t, |x_v| = v, |x'| = n - t - v$
$x_v \leftarrow x_v \oplus P(x_t)$
$y \leftarrow g(x_t \mid\mid x_v \mid\mid x')$
return $y$
```

This process is summarized in Fig. 6. In order to prevent the adversary from reaching either the input or the output of P, which could make some attacks possible, we impose that $t + v \leq c$ so that the whole plug input and output are located in the capacity.

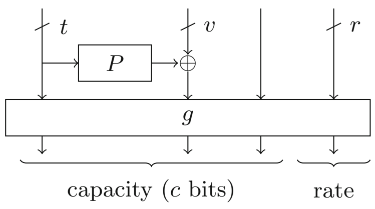

Fig. 6. The hard sponge transformation $(g \cdot P)$.

Theorem 3 (HSp absorption hardness). *Consider a sponge defined by the n-bit transformation $(g \cdot P)$, a rate r and a capacity c so that $r + c = n$ and $r > c$. Let $(g \cdot P)$ be defined as in Algorithm 4, where $P : \{0,1\}^t \rightarrow \{0,1\}^v$ is a plug $(\rho, u, \epsilon(p))$-hard against 2^p-adversaries.*

Let $\mathsf{Absorb} : \{0,1\}^{\ell \times r} \rightarrow \{0,1\}^c$ *be the function mapping an un-padded message m of ℓ r-bit blocks to the capacity bits of the internal state of the sponge after it absorbed m.*

Furthermore, suppose that the n-bit transformation g is such that the family of functions $g_k : \{0,1\}^c \rightarrow \{0,1\}^c$ indexed by $k \in \{0,1\}^r$ and defined as $g_k(x) = T_c\left((g(x||k)\right)$ can be modeled as a family of random oracles.

Then Absorb *is* $\big(\rho, u, \max(\epsilon(p)^{\ell-1}, 2^{p-c})\big)$*-hard against* 2^p*-adversaries.*

This theorem deals with un-padded messages. The padding of such a message imposes the creation of a new block with a particular shape which cannot be considered to be random.

Proof. Let $g_k : \{0,1\}^c \rightarrow \{0,1\}^c$ be as defined in the theorem. Let the message m be picked uniformly at random.

The first call to $(g \cdot P)$ is not $(\rho, u, \epsilon(p))$-hard. Indeed, the content of the message has not affected the content of the capacity yet. However, the capacity bits of the internal state after this first call to $(g \cdot P)$ are uniformly distributed as they are the image of a constant by the function indexed by m_0 from a family of 2^r different random oracles.

Let $m'_i = m_i \oplus z_i$, where m_i is the message block with index $i > 1$ and where z_i is the first r bits of the content of the sponge after the absorption of $m_0, ..., m_{i-1}$. That is, z_i is the content of the rate juste before the call to $(g \cdot P)$ following the addition of the message block m_i. We can therefore represent the absorption function as described in Fig. 7.

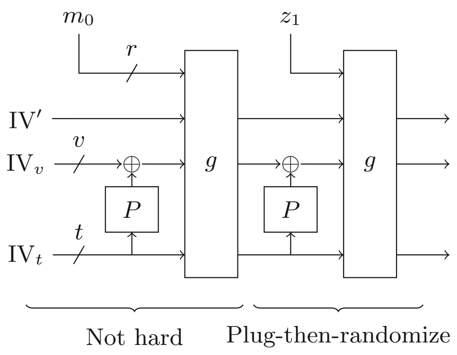

Fig. 7. An alternative representation of the absorption procedure.

Since the message blocks m_i have been picked uniformly at random, so are the values z_i. We can therefore apply Theorem 1, where the independent random oracles are g_{z_i}, the plug is P, the random message is $(g_{m_0} \cdot P)(0||IV)$, the block size is c and the number of rounds is $\ell - 1$. □

As c is typically much larger than a usual block cipher size of 128 bits, the probability of success of a 2^p adversary can be made much smaller when a sponge is built rather than a block cipher.

Note that if a sponge is used to provide e.g. authenticated encryption, the same bound should be used as the message is absorbed into the state in the same fashion in this case.

The following claim describes the hardness of the squeezing operation.

Claim 1 (HSp squeezing hardness). *Consider a sponge defined by the n-bit transformation $(g \cdot P)$, a rate r and a capacity c so that $r + c = n$ and $r > c$. Let $(g \cdot P)$ be defined as in Algorithm 4, where $P : \{0,1\}^t \rightarrow \{0,1\}^v$ is a plug $(\rho, u, \epsilon(p))$-hard against 2^p-adversaries.*

Let $\mathsf{Squeeze}^{\ell} : \{0,1\}^n \rightarrow \{0,1\}^{\ell \times r}$ be the function mapping an internal state of n bits to a stream of ℓ r-bit blocks obtained by iterating ℓ times the $\mathsf{Squeeze}$ operation.

Then $\mathsf{Squeeze}^{\ell}$ is $\left(\rho, u, \max(\epsilon(p)^{\ell}, 2^{p-(c+r)})\right)$-hard against 2^p-adversaries.

We cannot prove this hardness using Theorem 1 because the transformations called in each round are all identical. In particular, they cannot be independent. This situation can however be interpreted as a variant of the one in the proof of Theorem 3 where z_i is not formally picked uniformly at random as there is no message absorption but can be interpreted as such because it is the output of the sponge function.

The claimed probability of success bound comes from the hardness of approximating all ℓ calls to the plug composed with the sponge transformation $(\epsilon(p)^{\ell})$ and the hardness of using a precomputation of the image of 2^p possible internal states $(2^{p-(c+r)})$).

If the sponge is used to provide a simple stream cipher, the bound of Claim 1 should be used. Indeed, since there is no message absorption in this case, Theorem 3 cannot be used.

5 Practical Instances: Skipper and Whale

We illustrate the versatility and simplicity of the modes of operation described in the previous section by presenting an instance of each. The first is an asymmetrically time-hard block cipher called Skipper and the second is a code-hard sponge called Whale.

5.1 The Skipper Block Cipher

One possible application for *egalitarian computing* which is mentioned but not explored in [13] is *obfuscation*. The idea is to modify a program in such a way that a memory-hard function must be computed in parallel to the execution of the program. Using this very general approach, any program or function could be made memory-hard, not just cryptographic ones. However, a shortcoming in this case is the fact that the compiler returning the obfuscated code must also pay the full price of running this parallel memory-hard function.

Solving this issue requires a primitive with asymmetric memory hardness such as Diodon. However, this primitive is rather slow even for privileged users.

Therefore, we build instead an asymmetrically time-hard block cipher using the HBC mode to combine the AES and the RSA-lock plug. The result is the asymmetrically time-hard block cipher Skipper. It could be used to create an efficient obfuscator. The obfuscator would use the fast implementation of the plug to create an obfuscated program which forces common users to evaluate its slow version to run the program. That way, the computational hardness is only paid by the users of the program and not by the compiler. While this cost might be offset through the use of dedicated hardware for the computation of RSA-lock, we note that this function cannot be parallelized.

Our proposal Skipper is $\text{HBC}[AES - 128, \text{RSAlock}_{\eta}^{n_p}, 2]$, that is, a 128-bit block cipher using a 128-bit secret key k, an $\text{RSAlock}_{\eta}^{n_p}$ instance truncated to 40 bits as a plug and 3 calls to AES-128 sandwiching 2 calls to the plug. The plug operates modulo $N \geq 2^{n_p}$. The Skipper encryption procedure is described in Algorithm 5.

Algorithm 5. Skipper encryption
Inputs: n-bit plaintext x; k-bit key k; RSA modulus N
Output: n-bit ciphertext y

```
$y \leftarrow \text{AES}_k(x)$
**for all** $i \in \{1, 2\}$ **do**
    $y_1 \mid\mid y_2 \leftarrow y$, where $|y_1| = 88$ and $|y_2| = 40$
    $y_2 \leftarrow y_2 \oplus T_{40}(y_1^{2^\eta} \mod N)$
    $y \leftarrow \text{AES}_{k \oplus i}(y_1 || y_2)$
**end for**
**return** $y$
```

The RSA-based plug we use is asymmetrically $\big(\mathsf{Time}, \eta, \max(2^{p-88}, 2^{-40})\big)$-hard. As said before in Sect. 3.3, we assume that no adversary can evaluate $x^{2^\eta} \mod N$ without performing η squarings in the modular ring. However, a 2^p-adversary can either guess all 40 bits of the output, which succeeds with probability 2^{-40}, or store 2^p out of the 2^{88} possible outputs, in which case a successful evaluation is possible with probability 2^{p-88}.

Merely guessing is the best strategy unless the adversary has access to at least $40 \times 2^{88-40} \approx 2^{53.3}$ bits of storage, i.e. more than a thousand terabytes. Furthermore, the cost of such a pre-computation could only be amortized if more than $2^{48}/2 = 2^{47}$ blocks are encrypted using the same plug, i.e. 2^{54} bits (more than a thousand Tb). Otherwise, the time taken by the precomputation would be superior to the time needed to evaluate the slow function. We therefore consider 2^{48}-adversaries, that is, adversaries capable of pre-computing 2^{48} values of $\text{RSAlock}_{\eta}^{n_p}(x)$. Such an adversary is already quite powerful as it has significant computing power and storage in addition to knowing the secret key k. Providing maximum security against more powerful adversaries would probably be over-engineering. Thus, in our setting, the plug is asymmetrically $(\mathsf{Time}, \eta, 2^{-40})$-hard.

Claim 2 (Properties of Skipper). *The block cipher* Skipper *is asymmetrically* $(\mathsf{Time}, \eta, 2^{-80})$*-hard and cannot be distinguished from a pseudo-random permutation using less than* 2^{128} *operations.*

Skipper is $\text{HBC}[\text{AES} - 128, \text{RSAlock}_{\eta}^{n_p}, 2]$ and its plug is asymmetrically $(\mathsf{Time}, \eta, 2^{-40})$-hard. Thus, by applying Theorem 2 we obtain that Skipper is asymmetrically $\left(\mathsf{Time}, \eta, \max\left(2^{48-128}, (2^{-40})^2\right)\right)$-hard.

As there is to the best of our knowledge no related-key attack against full-round AES-128 in the case where the related keys are linked by a simple XOR, we claim that Skipper cannot be distinguished from a random permutation using much less than 2^{128} operations. Should such distinguishers be found, an alternative key schedule such as the one from [27] could be used.

We implemented Skipper on a regular desktop PC. The corresponding benchmarks are provided in the full version of this paper [15].

5.2 The Whale Hash Function

Preventing the leakage of encryption keys is a necessity in order for a system to be secure. A possible method for preventing this was informally proposed by Shamir in a talk at RSA'2013 and then formalized by Bellare et al. in their CRYPTO'16 paper. As the throughput of the exfiltration method used by the attacker is limited, using a huge key would make their task all the harder. To use our terminology, an encryption algorithm with significant code-hardness would effectively be bound to the physical device storing it: since the code cannot be compressed, an attacker would have to duplicate the whole implementation to be able to decrypt the communications. Even a partial leakage would be of little use.

The proposal of Bellare et al., XKEY2, is effectively a code-hard key derivation algorithm which takes as input a random initialization vector and outputs a secret key. Since it is code-hard, an attacker cannot evaluate this function without full knowledge of the source code of the function and cannot extract a smaller (and thus easier to leak) implementation.

We propose the code-hard hash function Whale as an alternative to XKEY2. It can indeed be used to derive a key by hashing a nonce, a process which cannot be approximated by an attacker unless they duplicate the entirety of the implementation of Whale. Whale is a simple sponge-based hash function which uses the XOR of $\lceil 128/t \rceil$ instances of BigLUT_t^{128} as a plug. Different choices of t lead to different levels of code-hardness. It is only parametrized by the input length of the tables t.

It is based on SHA-3-256: it uses the $\text{Keccak} - f[1600]$ permutation, the same padding scheme, the same rate $r = 1088$, the same capacity $c = 512$ and the same digest size of 256 bits. There are only two differences:

- the permutation $\text{Keccak} - f[1600]$ is augmented with the code-hard plug consisting in the XOR of $\ell = \lceil 128/t \rceil$ distinct instances of BigLUT_t^{128}, and
- t blank calls to the transformation are performed between absorption and squeezing.

These parameters were chosen so as to prevent an adversary with access to at most half of the implementation of WHALE to compute the digest of a message with probability higher than 2^{-128}.

Claim 3 (Code-hardness of Whale). *The* WHALE *hash function using tables with t-bit inputs is $(\mathsf{Code}, 2^{t+13}/t, 2^{-128})$-hard against an adversary trying to use only half of the code-space used to implement* WHALE.

WHALE uses $\lceil 128/t \rceil$ tables of 2^t 128-bit entries. Thus, about $2^t \times 128 \times \lceil 128/t \rceil \approx 2^{t+14}/t$ bits are needed to store the implementation of its plug. An adversary trying to compress it and divide its size by 2 therefore has access to $2^{t+13}/t$ bits. Note however that, since the entries in each instance of BigLUT_t^{128} have been picked uniformly at random, it is impossible to actually compress them. The best an attacker can do is therefore to store as many entries as they can.

When hashing a message, at least t calls to the plug are performed during the blank calls to the transformation between the absorption and the squeezing. Therefore, the adversary needs to successfully compute $t \times \lceil 128/t \rceil \geq 128$ entries of the big tables. If they only have half of them stored, then they succeed in computing the digest of a message with probability at most 2^{-128}.

6 Conclusion

We have presented for the first time a unified framework to study all three forms of hardness (time, memory and code) as well as their asymmetric variants. We have proposed DIODON, the first asymmetrically memory-hard function. We have also presented the first general approach for building a cryptographic primitive with *any* type of hardness and illustrated it with two fully specified proposals. The first is the asymmetrically time-hard block cipher SKIPPER which can be made arbitrarily slow for some users while retaining its efficiency for those knowing a secret key. The second is the code-hard hash function WHALE whose implementation cannot be compressed.

Acknowledgements. We thank anonymous reviewers from S&P, USENIX and ASIACRYPT'17 for their helpful comments. The work of Léo Perrin was supported by the CORE project ACRYPT (ID C12-15-4009992) funded by the Fonds National de la Recherche, Luxembourg.

References

1. Alwen, J., Blocki, J.: Efficiently computing data-independent memory-hard functions. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9815, pp. 241–271. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53008-5_9
2. Alwen, J., Blocki, J.: Towards practical attacks on Argon2i and balloon hashing. Cryptology ePrint Archive, Report 2016/759 (2016). http://eprint.iacr.org/2016/759

3. Alwen, J., Chen, B., Pietrzak, K., Reyzin, L., Tessaro, S.: Scrypt is maximally memory-hard. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 33–62. Springer, Cham (2017). doi:10.1007/978-3-319-56617-7_2
4. Alwen, J., Serbinenko, V.: High parallel complexity graphs and memory-hard functions. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, pp. 595–603. ACM (2015)
5. Ateniese, G., Bonacina, I., Faonio, A., Galesi, N.: Proofs of space: when space is of the essence. In: Abdalla, M., De Prisco, R. (eds.) SCN 2014. LNCS, vol. 8642, pp. 538–557. Springer, Cham (2014). doi:10.1007/978-3-319-10879-7_31
6. Baldimtsi, F., Kiayias, A., Zacharias, T., Zhang, B.: Indistinguishable proofs of work or knowledge. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10032, pp. 902–933. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53890-6_30
7. Bellare, M., Kane, D., Rogaway, P.: Big-key symmetric encryption: resisting key exfiltration. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9814, pp. 373–402. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53018-4_14
8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Sponge functions. In: ECRYPT hash workshop, vol. 2007. Citeseer (2007)
9. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak specifications. Submission to NIST (Round 2) (2009)
10. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the sponge: single-pass authenticated encryption and other applications. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 320–337. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28496-0_19
11. Biryukov, A., Bouillaguet, C., Khovratovich, D.: Cryptographic schemes based on the ASASA structure: black-box, white-box, and public-key (extended abstract). In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 63–84. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45611-8_4
12. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: new generation of memory-hard functions for password hashing and other applications. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 292–302. IEEE (2016)
13. Biryukov, A., Khovratovich, D.: Egalitarian computing. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, 10–12 August 2016, pp. 315–326. USENIX Association (2016). https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/biryukov
14. Biryukov, A., Khovratovich, D.: Equihash: asymmetric proof-of-work based on the generalized birthday problem. In: Proceedings of NDSS 2016, 21–24 February 2016, San Diego, CA, USA. ISBN 1-891562-41-X (2016)
15. Biryukov, A., Perrin, L.: Symmetrically and asymmetrically hard cryptography (full version). Cryptology ePrint Archive, Report 2017/414 (2017). http://eprint.iacr.org/2017/414
16. Bogdanov, A., Isobe, T.: White-box cryptography revisited: Space-hard ciphers. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 1058–1069. ACM (2015)
17. Bogdanov, A., Isobe, T., Tischhauser, E.: Towards practical whitebox cryptography: optimizing efficiency and space hardness. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 126–158. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53887-6_5

18. Boneh, D., Corrigan-Gibbs, H., Schechter, S.: Balloon hashing: a memory-hard function providing provable protection against sequential attacks. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 220–248. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53887-6_8
19. Daemen, J., Rijmen, V.: The Design of Rijndael: AES-the Advanced Encryption Standard. Springer, Heidelberg (2002). doi:10.1007/978-3-662-04722-4
20. Delerablée, C., Lepoint, T., Paillier, P., Rivain, M.: White-box security notions for symmetric encryption schemes. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 247–264. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43414-7_13
21. Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 585–605. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48000-7_29
22. Fouque, P.-A., Karpman, P., Kirchner, P., Minaud, B.: Efficient and provable white-box primitives. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 159–188. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53887-6_6
23. Gaži, P., Tessaro, S.: Provably robust sponge-based PRNGs and KDFs. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 87–116. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49890-3_4
24. Hellman, M.: A cryptanalytic time-memory trade-off. IEEE Trans. Inf. Theory **26**(4), 401–406 (1980)
25. Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational) (Sep 2000). http://www.ietf.org/rfc/rfc2898.txt
26. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
27. Nikolić, I.: Tweaking AES. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 198–210. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19574-7_14
28. Percival, C.: Stronger key derivation via sequential memory-hard functions. Self-published, pp. 1–16 (2009)
29. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). http://doi.acm.org/10.1145/359340.359342
30. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1996)