Memory-Tight Reductions

Benedikt Auerbach[1(✉)], David Cash[2], Manuel Fersch[1], and Eike Kiltz[1]

[1] Horst Görtz Institute for IT Security, Ruhr University Bochum, Bochum, Germany
{benedikt.auerbach,manuel.fersch,eike.kiltz}@rub.de
[2] Rutgers University, New Brunswick, NJ, USA
david.cash@cs.rutgers.edu

Abstract. Cryptographic reductions typically aim to be *tight* by transforming an adversary A into an algorithm that uses essentially the same resources as A. In this work we initiate the study of *memory efficiency* in reductions. We argue that the amount of working memory used (relative to the initial adversary) is a relevant parameter in reductions, and that reductions that are inefficient with memory will sometimes yield less meaningful security guarantees. We then point to several common techniques in reductions that are memory-inefficient and give a toolbox for reducing memory usage. We review common cryptographic assumptions and their sensitivity to memory usage. Finally, we prove an impossibility result showing that reductions between some assumptions must *unavoidably* be either memory- or time-inefficient. This last result follows from a connection to data streaming algorithms for which unconditional memory lower bounds are known.

Keywords: Memory · Tightness · Provable security · Black box reduction

1 Introduction

Cryptographic reductions support the security of a cryptographic scheme S by showing that any attack against S can be transformed into an algorithm for solving a problem P. The *tightness* of a reduction is in general some measure of how closely the reduction relates the resources of attacks against S to the resources of the algorithm for P. A tighter reduction gives a better algorithm for P, ruling out a larger class of attacks against S. Typically one considers resources like runtime, success probability, and sometimes the number of queries (to oracles defined in P) of the resultant algorithm when evaluating the tightness of a reduction.

This work revisits how we measure the resources of the algorithm produced by a reduction. We observe that *memory usage* is an often important but overlooked metric in evaluating cryptographic reductions. Consider typical "tight" reductions from the literature, which start with an attack against a scheme S that uses (say) time t_S to achieve success probability ε_S, and transform the

J. Katz and H. Shacham (Eds.): CRYPTO 2017, Part I, LNCS 10401, pp. 101–132, 2017.
DOI: 10.1007/978-3-319-63688-7_4

attack into an algorithm for problem P running in time $t_P \approx t_S$ and succeeding with probability $\varepsilon_P \approx \varepsilon_S$. We observe that reductions tight in this sense are sometimes highly *memory-loose*: If the attack against S used m_S bits of working memory, the reduction may produce an algorithm using $m_P \gg m_S$ bits of memory to solve P. Depending on P, this changes the conclusions we can draw about the security of the scheme.

In this paper we investigate memory-efficiency in cryptographic reductions in various settings. We show that some standard decisions in security definitions have a bearing on memory efficiency of possible reductions. We give several simple techniques for improving memory efficiency of certain classes of reductions, and finally turn to a connection between streaming algorithms and memory/time-efficient reductions.

Tightness, memory-tightness, and security. Reductions between a problem P and a cryptographic scheme S that approximately preserve runtime and success probability are usually called *tight* (c.f. [6,8,17]). Tight reductions are preferred because they provide stronger assurance for the security of S. Specifically, let us call an algorithm running in time t and succeeding with probability ε a (t, ε)-algorithm (for a given problem, or to attack a given scheme). Suppose that a reduction converts a (t_S, ε_S)-adversary against scheme S into a (t_P, ε_P)-algorithm for P where (t_P, ε_P) are functions of the first two. If it is believed that no (t_P, ε_P) algorithm should exist for P, then one concludes that no (t_S, ε_S) adversary can exist against S.

If a reduction is not tight, then in order to conclude that scheme S is secure against (t_S, ε_S)-adversaries one must adjust the parameters of the instance of P on which S is built, leading to a less efficient construction. In some extreme cases, obtaining a reasonable security level for a scheme with a non-tight reduction leads to an impractical construction. Addressing this issue has become an active area of research in the last two decades (e.g. [4–6,8,11,12,18]).

In this work we keep track of the amount of memory used in reductions. To see when memory usage becomes relevant, let a (t, m, ε)-algorithm use t time steps, m bits of memory, and succeed with probability ε. A tight reduction from S to P transforms $(t_S, m_S, \varepsilon_S)$-adversaries into $(t_P, m_P, \varepsilon_P)$-algorithms, where "tight" guarantees $t_S \approx t_P$ and $\varepsilon_S \approx \varepsilon_P$, but permits $m_P \gg m_S$, up to the worst-case $m_P \approx t_P$.

Now, suppose concretely that we want S to be secure against $(2^{256}, 2^{128}, O(1))$-adversaries, based on very conservative estimates of the resources available to a powerful government. Consider two possible "tight" reductions: One that is additionally "memory-tight" and transforms a $(2^{256}, 2^{128}, O(1))$-adversary A against S into a $(2^{256}, 2^{128}, O(1))$-algorithm B_{mt} for P, and one that is "memory-loose" and instead only yields a $(2^{256}, 2^{256}, O(1))$-algorithm $\mathsf{B}_{\mathrm{nmt}}$ for P.

The crucial point is that some problems P can be solved faster when larger amounts of memory are used. In our example above, it may be that P is impossible to solve with 2^{256} time and 2^{128} memory for some specific security parameter λ. But with both time and memory up to 2^{256} bits, the best algorithm may be able to solve instances of P with security parameter λ, and with even

larger parameters up to some $\lambda' > \lambda$. The memory-looseness of the reduction now bites, because to achieve the original security goal for S we must use the larger parameter λ' for P, resulting in a slower instantiation of the scheme. When P is a problem involving a symmetric primitive where the "security parameter" cannot be changed the issue is more difficult to address.

We now address two points in turn: If P is easier to solve when large memory is available, what does this mean for memory-tight reductions? And when are reductions "memory-loose"?

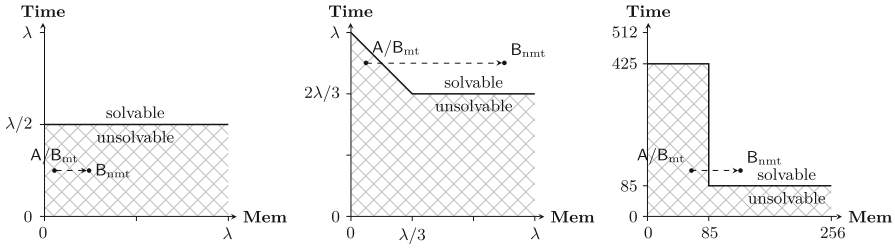

Fig. 1. Time/memory trade-off plots for collision-resistance (CR_2, left), triple collision-resistance (CR_3, middle) and LPN with dimension 1024 and error rate 1/4 (right). All plots are log-log and the axes on the right plot are not to scale.

Memory-sensitive problems and memory-tightness. Many, but not all, problems P relevant to cryptography can be solved more quickly with large memory than with small. In the public-key realm these include factoring, discrete-logarithm in prime fields, Learning Parities with Noise (LPN), Learning With Errors (LWE), approximate Shortest Vector Problem, and Short Integer Solution (SIS). In symmetric-key cryptography such problems include key-recovery against multiple-encryption, finding multi-collisions in hash functions, and computation of memory-hard functions. We refer to problems like these as *memory-sensitive*. (We refer to Sect. 6 for more discussion.)

On the other hand, problems P exist where the best known algorithm also uses small memory: Discrete-logarithm in elliptic curve groups over prime-fields [16], finding (single) collisions in hash functions [23], finding a preimage in hash functions (exhaustive search), and key recovery against block-ciphers (also exhaustive search).

Let us consider some specific examples to illustrate the impact of a memory-loose reduction to a non-memory-sensitive versus a memory-sensitive problem. Let CR_k be the problem of finding a k-way collision in a hash function H with λ output bits, that is, finding k distinct domain points $x_1, \ldots, x_k$ such that $\mathsf{H}(x_1) = \mathsf{H}(x_2) = \cdots = \mathsf{H}(x_k)$ for some fixed $k \geq 2$.

First suppose we reduce the security of a scheme S to CR_2, which is standard collision-resistance. The problem CR_2 is not memory-sensitive, and the best known attack is a $(2^{\lambda/2}, O(1), O(1))$-algorithm. In the left plot of Fig. 1 we visualize the "feasible" region for CR_2, where the shaded region is unsolvable. Now

we consider two possible reductions. One is a memory-tight reduction which maps an adversary A (with some time and memory complexity, with possibly much less memory than time) to an algorithm B_{mt} for CR_2 with the same time and memory. The other reduction is memory-loose (but time-tight) and maps A to an adversary B_{nmt} that uses time and memory approximately equal to the time of A. We plot the effect of these reductions in the left part of the figure. A tight reduction leaves the point essentially unchanged, while a memory-loose reduction moves the point horizontally to the right. Both reductions will produce a B_{nmt} in the region not known to be solvable, thus giving a meaningful security statement about A that amounts to ruling out the shaded region of adversaries. We do note that there is a possible quantitative difference in the guarantees of the reductions, since it is only harder to produce an algorithm with smaller memory, but this benefit is difficult to measure.

Now suppose instead that we reduce the security of a scheme S to CR_3. The best known attack against CR_3 is a $(2^{(1-\alpha)\lambda}, 2^{\alpha\lambda}, O(1))$-algorithm due to Joux and Lucks [20], for any $\alpha \leq 1/3$. We visualize this time-memory trade-off in the middle plot of Fig. 1, and again any adversary with time and memory in the shaded region would be a cryptanalytic advance. We again consider a memory-tight versus a memory-loose reduction. The memory-tight reduction preserves the point for the adversary A in the plot and thus rules out $(t_S, m_S, O(1))$ adversaries for any t_S, m_S in the shaded region. A memory-loose (but time-tight) reduction mapping A to B_{nmt} for CR_3 that blows up memory usage up to time usage will move the point horizontally to the right. We can see that there are drastic consequences when the original adversary A lies in the triangular region with time $>2\lambda/3$ and memory $<\lambda/3$, because the reduction produces an adversary B_{nmt} using resources for which CR_3 is known to be broken. In summary, the reduction only rules out adversaries A below the horizontal line with time $= 2\lambda/3$.

Finally we consider an example instantiation of parameters for the *learning parities with noise* (LPN) problem, which is memory-sensitive, where a memory-loose reduction would diminish security guarantees. In Sect. 6 we recall this problem and the best attacks, and in the right plot of Fig. 1 the shaded region represents the infeasible region for the problem in dimension 1024 and error rate 1/4. (For simplicity, all hidden constants are ignored in the plot.) In this problem the effect of memory-looseness is more stark. Despite using a large dimension, a memory-loose reduction can only rule out attacks running in time $<2^{85}$. A memory-tight reduction, however, gives a much stronger guarantee for adversaries with memory less than 2^{85}.

Memory-loose reductions. Reductions are often memory-loose, and small decisions in definitions can lead to memory usage being artificially high. We start with an illustrative example.

Suppose we have a tight security reduction (in the traditional sense) in the random oracle model [7] between a problem P and some cryptographic scheme S. More concretely, suppose a reduction transforms a $(t_S, m_S, \varepsilon_S)$-adversary A_S in the random-oracle model into a $(t_P, m_P, \varepsilon_P)$-algorithm A_P for P. A typical

reduction has A_P simulate a security game for A_S, including the random oracle, usually via a table that stores responses to queries issued by A_S. Naively removing the table from storage usually is not an option for various reasons: For example, if A_S queries the oracle on the same input twice, then it expects to see the same output twice, or perhaps the reduction needs to "program" the random oracle with responses that must be remembered.

Storing a table for the random oracle may dramatically increase memory usage of the algorithm A_P. If adversary A_S makes q_H queries to the random oracle, then A_P will store $\Omega(q_H)$ bits of memory, plus the internal memory m_S of A_S during the simulation, which gives

$$m_P = m_S + \Omega(q_H).$$

In the worst case, A_S could run in constant memory and make one random oracle query per time unit, meaning that A_P requires as much memory as its running time. Thus the reduction may be "tight" in the traditional sense with $t_P \approx t_S, \varepsilon_P \approx \varepsilon_S$, but also have

$$m_P = m_S + t_S. \tag{1}$$

Thus A_P may use an enormous amount of memory m_P even if A_S satisfied $m_S = O(1)$.

This example is only the start. Memory-looseness is sometimes, but not always, easily fixed, and seems to occur because it was not measured in reductions. Below we will furnish examples of other reductions that are (sometimes implicitly) memory-loose. We will also discuss some decisions in definitions and modeling that dramatically effect memory usage but are not usually stressed.

1.1 Our Results

Even though there exists an extensive literature on tightness of cryptographic security reductions (e.g. [5,8,11,12]), memory has, to the best of our knowledge, not been considered in the context of security reductions. In this paper we first identify the problems related to non-memory-tight security reductions. To overcome the problems, we initiate a systematic study on how to make known security reductions memory-tight. Concretely, we provide several techniques to obtain memory-efficient reductions and give examples where they can be applied. Our techniques can be used to make many security reductions memory-tight, but not all of them. Furthermore, we show that this is inherent, i.e., that there exist natural cryptographic problems that do not have a fully tight security reduction. Finally, we examine various memory-sensitive problems such as the learning parity with noise (LPN) problem, the factoring problem, and the discrete logarithm problem over finite fields.

The Random Oracle technique. Recall that a classical simulation of the random oracle using the lazy sampling technique requires the reduction to store $O(q_H)$ values. The idea is to replace the responses $H(x)$ to a random oracle

query x by $\mathsf{PRF}(k, x)$, where PRF is a pseudo-random function and k is its key. The limitation of this technique is that it can only be applied to very restricted cases of a programmable random oracle.

The Rewinding Technique. The idea of the rewinding technique is to use the adversary as a "memory device." Concretely, whenever the reduction would like to access values previously output by the adversary that it did not store in its memory, it simply rewinds the adversary which is executed with the same random coins and with the same input. This way the reduction's running time doubles, but (unlike previous applications of the rewinding technique in cryptography, e.g., [22]) the overall success probability does not decrease. The rewinding technique can be applied multiple times providing a trade-off between memory efficiency and running time of the reduction. To exemplify the techniques, we show a memory-tight security reduction to the RSA full-domain hash signature scheme in the appendix.

A Lower Bound. Some reductions appear (to us at least) to inherently require increased memory. We take a first step towards formalizing this intuition by proving a lower bound on the memory usage of a class of black-box reductions in two scenarios.

First, we revisit a reduction implicitly used to justify the standard unforgeability notion for digital signatures, which reduces a game with several chances to produce a valid forgery to the standard game with only one chance. One can take this as a possible indication that signatures with memory-tight reductions in the more permissive model may be preferred. Second, we prove a similar lower bound on the memory usage of a class of reductions between a "multi-challenge" variant of collision resistance and standard collision resistance.

Interestingly, our lower bound follows from a result on *streaming algorithms*, which are designed to use small space while working with sequential access to a large stream of data.

Open problems. This work initiates the study of memory-tight reductions in cryptography. We give a number of techniques to obtain such reductions, but many open problems remain. There are likely other reductions in the literature that we have not covered, and to which our techniques do not apply. It is even unclear how one should consider basic definitions, like unforgeability for signatures, since the generic reductions from more complicated (but more realistic) definitions may be tight but not memory-tight.

One reduction we did consider, but could not improve, is the IND-CCA security proof for Hash ElGamal in the random oracle model [1] under the gap Diffie-Hellman assumption. This reduction (and some others that use "gap" assumptions) use their random oracle table in a way that our techniques cannot address. We conjecture that a memory-tight reduction does not exist in this case, and leave it as an open problem to (dis)prove our conjecture.

2 Complexity Measures

We denote random sampling from a finite set A according to the uniform distribution with $a \xleftarrow{\$} A$. By $\mathrm{Ber}(\alpha)$ we denote the Bernoulli distribution for parameter α, i.e., the distribution of a random variable that takes value 1 with probability α and value 0 with probability $1-\alpha$; by $\mathbb{P}_\ell$ the set of primes of bit size ℓ and by log the logarithm with base 2.

2.1 Computational Model

Computational model. All *algorithms* in this paper are taken to be RAMs. These programs have access to memory with words of size λ, along with a constant number of registers that each hold one word. In this paper λ will always be the security parameter of a construction or a problem under consideration.

We define *probabilistic algorithms* to be RAMs with a special instruction that fills a distinguished register with random bits (independent of other calls to the special instruction). We note that this instruction does not allow for rewinding of the random bits, so if the algorithm wants to access previously used random bits then it must store them. *Running* an algorithm A means executing a RAM machine with input written in its memory (starting at address 0). If A is randomized, we write $y \xleftarrow{\$} \mathsf{A}(I)$ to denote the random variable y that is obtained by running A on input I (which may consist of a tuple $I = (I_1, \ldots, I_n)$). If A is deterministic, we write $\leftarrow$ instead of $\xleftarrow{\$}$. We sometimes give an algorithm A access to *stateful oracles* $\mathsf{O}_1, \mathsf{O}_2, \ldots, \mathsf{O}_n$. Each O_i is defined by a RAM M_i. We also define an associated string st_O called the *oracle state* that is stored in a protected region of the memory of A that can only be read by the oracles. Initially st_O is defined to be empty. An algorithm A *calls an oracle* O_i via a special instruction, which runs the corresponding RAM on input from a fixed region of memory of A along with the oracle state st_O. The RAM M_i uses its own protected working memory, and finally its output is written into a fixed region of memory for A, the updated state is written to st_O, and control is transferred back to A.

Games. Most of our security definitions and proofs use *code-based games* [9]. A game G consists of a RAM defining an Init oracle, zero or more stateful oracles $\mathsf{O}_1, \ldots, \mathsf{O}_n$, and a Fin RAM oracle. An adversary A is said to play game G if its first instruction calls Init (handing over its own input) and its last instruction calls Fin, and in between these calls it only invokes $\mathsf{O}_1, \ldots, \mathsf{O}_n$ and performs local computation. We further require that A outputs whatever Fin outputs.

Executing game G *with* A is formally just running A with input λ, the security parameter. Keeping with convention, we denote the random variable induced by executing G with A as G^A (where the sample space is the randomness of A and the associated oracles). By $\mathsf{G}^\mathsf{A} \Rightarrow \mathtt{out}$ we denote the event that G executed with A outputs $\mathtt{out}$. In our games we sometimes denote a "Stop" command that takes an argument. When Stop is invoked, its argument is considered the output of the game (and the execution of the adversary is halted). If a game description omits

the Fin procedure, it means that when A calls Fin on some input x, Fin simply invokes Stop with argument x. By default, integer variables are initialized to 0, set variables to $\emptyset$, strings to the empty string and arrays to the empty array.

2.2 Complexity Measures

This work is concerned with measuring the resource consumption of an adversary in a way that allows for meaningful conclusions about security. Success probabilities and time are widely used in the cryptographic literature with general agreement on the details, which we recall first. Memory consumption of reductions is however new, so we next discuss the possible options in measuring memory and the implications.

Success Probability. We define the *success probability of* A *playing game* G *as* $\mathbf{Succ}(\mathsf{G}^{\mathsf{A}}) := \Pr[\mathsf{G}^{\mathsf{A}} \Rightarrow 1]$.

Runtime. Let A be an algorithm (RAM) with no oracles. The runtime of A, denoted $\mathbf{Time}(\mathsf{A})$, is the worst-case number of computation steps of A over all inputs of bit-length λ and all possible random choices. Now let G be a game and A be an adversary that plays game G. The runtime of executing G with A is usually taken to be the number of computation steps of A plus the number of computation steps of each RAM used to respond to oracle queries: We denote this as $\mathbf{TotalTime}(\mathsf{G}^{\mathsf{A}})$ or $\mathbf{TotalTime}(\mathsf{A})$. One may prefer not to include the time used by the oracles, and in this case we denote $\mathbf{LocalTime}(\mathsf{G}^{\mathsf{A}})$ or $\mathbf{LocalTime}(\mathsf{A})$ to be the number of steps of A only.

Memory. We define the memory consumption of a RAM program A without oracles, denoted $\mathbf{Mem}(\mathsf{A})$, to be size (in words of length λ) of the code of A plus the worst-case number of registers used in memory at any step in computation, over all inputs of bit-length λ and all random choices. Now let G be a game and A be an adversary that plays game G. The memory required to execute game G with A includes the memory needed to input and output to A, as well as input and output to each oracle, along with the working memory and state of each oracle. We denote this as $\mathbf{TotalMem}(\mathsf{G}^{\mathsf{A}})$ or $\mathbf{TotalMem}(\mathsf{A})$. Alternatively, one may measure only the code and memory consumed by A, but not its oracles. We denote this measure by $\mathbf{LocalMem}(\mathsf{A})$.

One advantage of the **LocalMem** measure is that it can avoid small details of security definitions drastically changing the meaning of memory-tightness in reductions.

Sometimes it will be convenient to measure the memory consumption in bits, in which case we use $\mathbf{Mem}_2(\mathsf{A})$, $\mathbf{LocalMem}_2(\mathsf{A})$, and $\mathbf{TotalMem}_2(\mathsf{A})$.

2.3 Case Study I: Unforgeability of Digital Signatures

Let $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ be a digital signature scheme (see Sect. 5 for the exact syntax of signatures, which is standard). On the left side of Fig. 2 we recall the game UFCMA that defines the standard notion of (existential) unforgeability under chosen-message attacks. The advantage of an adversary A is

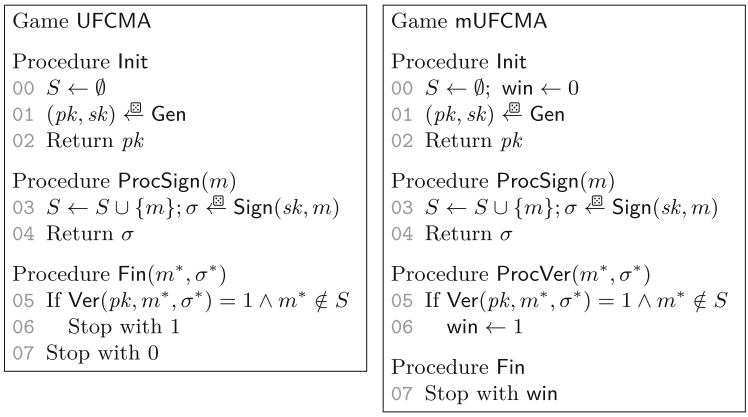

Fig. 2. Games UFCMA, mUFCMA.

defined by $\mathbf{Adv}(\mathsf{UFCMA}^{\mathsf{A}}) = \mathbf{Succ}(\mathsf{UFCMA}^{\mathsf{A}})$, and a signature scheme where $\mathbf{Adv}(\mathsf{UFCMA}^{\mathsf{A}})$ is "small" for some class of adversaries is usually defined to be "secure". In order for the definition to be meaningful, the game UFCMA checks that the signature σ^* on m^* is valid, and also that m^* was not queried to the signing oracle. In our version of the definition, the signing oracle maintains a set S of messages that were queried, and the game uses S to check if m^* was queried.

The UFCMA game is an example where we prefer **LocalMem** to **TotalMem**. Any adversary A playing UFCMA will always have $\mathbf{TotalMem}(\mathsf{A}) = \Omega(q_S)$, where q_S is the number of signature queries it issues, while it may have $\mathbf{LocalMem}(\mathsf{A})$ much smaller. Restricting the number of signing queries q_S is an option but weakens the definition.

An alternative style of definition for unforgeability is to limit the class of adversaries A considered to those that are "well behaved" in that they never submit an m^* that was previously queried. The game no longer needs to track which messages were queried to the signing oracle in order to be meaningful. This definition is equivalent up to a small increase in (local) running time, but it is not clear if the same is true for memory. To convert *any* adversary to be well behaved, natural approaches mimic our version of the game, storing a set S and checking the final forgery locally before submitting.

We contend that there is good reason to prefer our definition over the version that only quantifies over well-behaved adversaries. In principle, it is possible that a signature construction is secure against a class of well-behaved adversaries (say, running in a bounded amount of time and memory) but not against general adversaries running with the same time/memory. Counter-intuitively, such a general adversary might produce a forgery without knowing itself if the forgery is fresh and thus wins the game. Since we cannot rule this out, we prefer our stronger definition.

Stronger unforgeability. Games in many crypto-definitions are chosen to be simple and compact but also general. The game UFCMA only allows a single attempt at a forgery in order to shorten proofs, but the definition also tightly implies (up to a small increase in runtime) a version of unforgeability where the attacker gets many attempts, which more closely models usages where an attacker will have many chances to produce a forgery.

It is less clear how UFCMA relates to more general definitions when memory tightness is taken into account. To make this more concrete, consider the game mUFCMA (for "many UFCMA") on the right side of Fig. 2. In this game the adversary has an additional verification oracle. If it ever submits a fresh forgery to this oracle, it wins the game. It is easy to give a tight, but non-memory-tight, reduction converting any (t, m, ε)-adversary playing mUFCMA into a (t', m', ε)-adversary playing UFCMA for $t' \approx t$ but $m' \gg m$. Other trade-offs are also possible but achieving tightness in all three parameters seems difficult.

For the reasons described in the introduction, a memory-tight reduction from winning mUFCMA to winning UFCMA is desirable. In Sect. 4, we show that a certain class of black-box reductions for these problems in fact cannot be simultaneously tight in runtime, memory, and success probability. We conclude that signatures with dedicated memory-tight proofs against adversaries in the mUFCMA may provide stronger security assurance, especially when security is reduced to a memory-sensitive problem like RSA.

We remark that the common reduction from multi-challenge to single-challenge IND-CPA/IND-CCA security for public-key encryption is memory tight (but not tight in terms of the success probability).

2.4 Case Study II: Collision-Resistance Definitions

Collision-resistance, and multi-collision-resistance of hash functions, is used for security reductions in many contexts. Let H be a keyed hash function (with κ-bit keys), with standard syntax. On the left side of Fig. 3 we recall the game CR_t used to define t-collision resistance. The game provides no extra oracles, and A wins if it can find t domain points that are mapped to the same point by H.

As we will see in later sections, it is sometimes feasible to fix typical memory-tight reductions to CR_t. We however now consider using collision-resistance (for $t = 2$) for *domain extension of pseudorandom functions*. Let $\mathsf{F} : \{0,1\}^\kappa \times \{0,1\}^\delta \to \{0,1\}^\rho$ be a keyed function with input-length δ which should have random looking input/output behavior to some class of adversaries (see Sect. 3.1 for a formal definition of PRFs). We can define a new keyed function F^* that takes arbitrary-length inputs by

$$\mathsf{F}^* : \{0,1\}^{2\kappa} \times \{0,1\}^* \to \{0,1\}^\rho,$$
$$\mathsf{F}^*((k, k_h),\ x) = \mathsf{F}(k,\ \mathsf{H}(k_h, x)).$$

The proof that F^* is a PRF is an easy hybrid argument. One first bounds the probability that an adversary submits two inputs that collide in H. Once this

probability is known to be small, the memory-tight reduction to the pseudorandomness of F is immediate.

Naive attempts at the reduction to collision-resistance are however not memory-tight. One can run the adversary attacking F^* and record its queries, checking for any collisions, but this increases memory usage.

To model what such a proof is trying to do, we formulate a new game for t-collision resistance called mCR_t in the right side of Fig. 3. In the game, the adversary has an oracle $\mathsf{ProcInput}$ that takes a message and adds it to a set S. At the end of the game, the adversary wins if S contains any t inputs that are mapped to the same point. The game implements this check using counters stored in a dictionary.

```
Game CR_t

Procedure Init
08 k ←$ {0,1}^κ
09 Return k

Procedure Fin(m_1, ..., m_t)
10 If |{m_1, ..., m_t}| < t
11    Stop with 0
12 If ∀i : H(k, m_1) = H(k, m_i)
13    Stop with 1
14 Stop with 0
```

```
Game mCR_t

Procedure Init
15 k ←$ {0,1}^κ; S ← ∅
16 Return k

Procedure ProcInput(m)
17 S ← S ∪ {m}

Procedure Fin
18 Initialize dictionary D
19 For m ∈ S:
20    Increment D[H(k, m)]
21    If D[H(k, m)] ≥ t
22       Stop with 1
23 Stop with 0
```

Fig. 3. Games $\mathsf{CR}_t, \mathsf{mCR}_t$.

Returning to the proof for F^*, one can easily construct an adversary to play mCR_2 using any PRF adversary. The resulting reduction will be memory-tight. Thus it would be desirable to have a memory-tight reduction from mCR_2 to CR_2 to complete the proof. This however seems difficult or even impossible, and in Sect. 4 we show that a class of black-box reductions cannot be memory-tight. As discussed in the introduction, t-collision-resistance is not memory sensitive for $t = 2$, and thus the meaning of a memory-tight reduction is somewhat diminished (i.e. it does not justify more aggressive parameter settings). For $t > 2$ the effect of memory-tightness is more significant.

3 Techniques to Obtain Memory Efficiency

In this section we describe four techniques to obtain memory-efficient reductions. In Sect. 5 we show how to apply those techniques to memory-tightly prove the security of the RSA Full Domain Hash signature scheme [7]. Using this example we also point to technical challenges that may arise when applying multiple techniques in the same proof.

3.1 Pseudorandom Functions

First, we formally define pseudorandom functions. They are the main tool used in this section to make reductions memory efficient.

Definition 1. *Let κ, δ and ρ be integers. Further let* $\mathsf{F}\colon \{0,1\}^\kappa \times \{0,1\}^\delta \to \{0,1\}^\rho$ *be a deterministic algorithm and let* A *be an adversary that is given access to an oracle and outputs a single bit. The* PRF advantage *of* A *is defined as* $\mathbf{Adv}(\mathsf{PRF}^{\mathsf{A}}) := |\mathbf{Succ}(\mathsf{Real}^{\mathsf{A}}) - \mathbf{Succ}(\mathsf{Random}^{\mathsf{A}})|$, *where* Real *and* Random *are the games depicted in Fig. 4.*

Game Real	Game Random	Game Random_α
Procedure Init 00 $k \xleftarrow{\$} \{0,1\}^\kappa$	Procedure Init	Procedure Init
Procedure $\mathsf{O}_\mathsf{F}(x)$ 01 Return $\mathsf{F}(k,x)$	Procedure $\mathsf{O}_\mathsf{F}(x)$ 01 If $R[x]$ undefined: 02 $R[x] \xleftarrow{\$} \{0,1\}^\rho$ 03 Return $R[x]$	Procedure $\mathsf{O}_\mathsf{F}(x)$ 01 If $R[x]$ undefined: 02 $R[x] \xleftarrow{\$} \mathrm{Ber}(\alpha)$ 03 Return $R[x]$

Fig. 4. Games defining PRF and α-PRF advantage.

If the range of F *is just a single bit* $\{0,1\}$*, we define the* α-PRF advantage *with bias* $0 \le \alpha \le 1$ *of* A *as* $\mathbf{Adv}(\mathsf{PRF}^{\mathsf{A}}_\alpha) := |\mathbf{Succ}(\mathsf{Real}^{\mathsf{A}}) - \mathbf{Succ}(\mathsf{Random}^{\mathsf{A}}_\alpha)|$, *where* Real *and* Random_α *are the games in Fig. 4.*

Note that a $2^{-\rho}$-PRF can be easily constructed from a standard PRF with range $\{0,1\}^\rho$ by mapping 1^ρ to 1 and all other values to 0. A $1/q$-PRF for arbitrary q can be constructed in a similar way from a standard PRF with sufficiently large image size ρ.

3.2 Generating (Pseudo)random Coins

Our first technique is the simplest, where we observe random coins used by adversaries can be replaced with pseudorandom coins, and that this substitution will save memory in certain reductions.

Consider a security game G and an adversary A. Both are probabilistic processes and therefore require randomness. When considering memory efficiency details on storing random coins could come to dominate memory usage. Specifically, some reductions run an adversary multiple times with the same random tape, which must be stored in between runs. One possibility to do this is by sampling all randomness required in game G^{A} (including the randomness used by A) in advance. More formally let $L \le 2^\lambda$ be an upper bound on the amount of executions of the instruction filling an register with random bits in G^{A}. Then the sampling of random coins can be replaced filling and storing L registers (memory units) with random bits at the beginning of Init and in the rest of the

game replacing the ith call to the instruction with a procedure Coins returning the contents of the ith register. This is formalized in game G_0 of Fig. 5.

The game can be simulated in a memory-efficient way by replacing the random bits used by G and A with pseudorandom bits generated by a PRF $\mathsf{F}\colon \{0,1\}^\kappa \times \{0,1\}^\delta \to \{0,1\}^\lambda$, as described in Game G_1 of Fig. 5. In this variant the game sets up the counter i in the usual way. Then a PRF key k is sampled from a key space $\{0,1\}^\kappa$ and calls to Coins are simulated by returning the pseudorandom bits $\mathsf{F}(k,i)$. We now compare the two ways of executing the game in terms of success probability, running time, and memory consumption.

G_0: **Standard Coin Generation**	G_1: **Memory-Efficient Coin Generation**
Procedure Init	Procedure Init
00 $r \xleftarrow{\$} (\{0,1\}^\lambda)^L$	00 $k \xleftarrow{\$} \{0,1\}^\kappa$
Procedure Coins	Procedure Coins
01 $i \leftarrow i+1$	01 $i \leftarrow i+1$
02 Return r_i	02 Return $\mathsf{F}(k,i)$

Fig. 5. Generating (pseudo)random coins in a memory-efficient way. By r_i we denote the i^{th} block of λ bits of the string r.

Success Probability. By a simple reduction to the security of the PRF, there exists an adversary B with $\mathbf{LocalTime}(\mathsf{B}) = \mathbf{LocalTime}(\mathsf{A})$, $\mathbf{LocalMem}(\mathsf{B}) = \mathbf{LocalMem}(\mathsf{A}) + 1$ such that

$$\left|\mathbf{Succ}(\mathsf{G}_0^{\mathsf{A}})] - \mathbf{Succ}(\mathsf{G}_1^{\mathsf{A}})\right| \leq \mathbf{Adv}(\mathsf{PRF}^{\mathsf{B}})$$

(see Definition 1). Observe that B perfectly simulates the Coins oracle as follows. For A's i^{th} query to Coins, it queries O_F of the PRF games on i and relays its response back to A. To do this, it needs to store a counter of $\log L$ bits. All other procedures are simulated as specified in G_1.

Running Time. Game G_1 needs to evaluate the PRF (via algorithm F) L times, hence we have $\mathbf{TotalTime}(\mathsf{G}_1^{\mathsf{A}}) \leq \mathbf{TotalTime}(\mathsf{G}_0^{\mathsf{A}}) + L \cdot \mathbf{Time}(\mathsf{F})$.

Memory. Both games have to store a counter i of size $\log L \leq \lambda$ bits, which equals one memory unit. But while game G_0 needs memory for storing L strings, the memory-efficient game G_1 only needs additional memory $\mathbf{Mem}(\mathsf{F})$. Note that the PRF key is included in the memory of F. So overall, we have

$$\mathbf{TotalMem}(\mathsf{G}_0^{\mathsf{A}}) = \mathbf{LocalMem}(\mathsf{A}) + 1 + L,$$
$$\mathbf{TotalMem}(\mathsf{G}_1^{\mathsf{A}}) = \mathbf{LocalMem}(\mathsf{A}) + 1 + \mathbf{Mem}(\mathsf{F}).$$

Note that when applying this (and the following) techniques in a larger environment, special care has to be taken to keep the entire game consistent with the components changed by the technique. In particular, all intermediate reductions in a sequence of games have to be memory efficient to yield an overall memory-efficient reduction.

3.3 Random Oracles

Suppose a security game G is defined in the random oracle model, that is one of the game's procedures models a random oracle

$$H\colon \{0,1\}^{\delta} \to \{0,1\}^{\lambda}.$$

The standard way of implementing this is via a technique called lazy sampling [9], meaning that when an adversary A queries H on some value x, the game has to check if $H(x)$ is already defined, and if not, it samples $H(x)$ from some distribution and stores the value in a list, see G_0 in Fig. 6. This means that in the worst case, it needs to store as many strings as the number of adversarial queries.

However, there are several settings where the random oracle can be implemented by a PRF $\mathsf{F}\colon \{0,1\}^{\kappa} \times \{0,1\}^{\delta} \to \{0,1\}^{\lambda}$ as described in G_1 of Fig. 6, thus making G more memory-efficient. Among these settings are the non-programmable random oracle model and certain random oracles, where only values obtained or computed during the Init procedure are used to program them.

G_0: **Standard Random Oracle**	G_1: **Memory-Efficient Random Oracle**
Procedure Init	Procedure Init 00 $k \xleftarrow{\$} \{0,1\}^{\kappa}$
Procedure $\mathsf{RO}(x_i)$ 01 If $H[x_i]$ undefined: 02 $\quad H[x_i] \xleftarrow{\$} \{0,1\}^{\lambda}$ 03 Return $H[x_i]$	Procedure $\mathsf{RO}(x_i)$ 01 Return $\mathsf{F}(k, x_i)$

Fig. 6. The Random Oracle technique to simulate RO in a memory-efficient way. Here x_i denotes the i^{th} query to RO. Note that the queries $x_1, \ldots, x_q$ are not necessarily distinct.

In the following paragraph we analyze how success probability, running time and memory consumption change if we apply this technique.

Success Probability. There exists an adversary B with $\mathbf{LocalTime}(\mathsf{A}) = \mathbf{LocalTime}(\mathsf{B})$ and $\mathbf{LocalMem}(\mathsf{A}) = \mathbf{LocalMem}(\mathsf{B})$ such that

$$\left|\mathbf{Succ}(\mathsf{G}_0^{\mathsf{A}}) - \mathbf{Succ}(\mathsf{G}_1^{\mathsf{A}})\right| \leq \mathbf{Adv}(\mathsf{PRF}^{\mathsf{B}}).$$

B perfectly simulates the RO by relaying all of A's queries to O_{F} of the PRF games and forwarding the responses back to A. All other procedures are simulated as specified in G_1. When B is run with respect to game Random of Definition 1 it provides A with a perfect simulation of G_0, if it is run with respect to game Real with a perfect simulation of game G_1.

Running Time. Let q_H be the number of random oracle queries posed by the adversary. Then game G_1 needs to evaluate the PRF q_H times, hence we have $\mathbf{TotalTime}(\mathsf{G}_1^{\mathsf{A}}) \leq \mathbf{TotalTime}(\mathsf{G}_0^{\mathsf{A}}) + q_H \cdot \mathbf{Time}(\mathsf{F})$.

Memory. Game G_0 needs to store an array H of size at least $q_H \cdot \lambda$ bits ($= q_H$ memory units), while the memory-efficient game only needs memory to execute the PRF via algorithm F. So overall, we have

$$\mathbf{TotalMem}(\mathsf{G}_0^{\mathsf{A}}) \geq \mathbf{LocalMem}(\mathsf{A}) + q_H,$$
$$\mathbf{TotalMem}(\mathsf{G}_1^{\mathsf{A}}) = \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{F}).$$

3.4 Random Oracle Index Guessing Technique

This technique is used when random oracle queries are answered in two different ways, e.g. in a reduction where challenge values, like a discrete logarithm challenge $X = g^x$, are embedded in the programmable random oracle. Usually this is done by guessing some index i^* between 1 and q_H in the beginning, where q_H is the number of random oracle queries posed by the adversary. During the simulation, the challenge value is then embedded in the reduction's response to the $i^{*\text{th}}$ random oracle query.

To do this, the game needs to keep a list of all queries and responses. Independently of the way the game answers all the other queries except for the $i^{*\text{th}}$ one, simply keeping a counter is not sufficient, since an adversary posing the same query all the time would then receive two different responses and the random oracle thus wouldn't be well defined anymore. An example of such a game using the index guessing technique is game G_0 of Fig. 7, where two deterministic procedures P_0 and P_1 are used to program H depending on i^*.

To make games of this kind memory-efficient, one can use a $1/q_H$-PRF (see Definition 1) $\mathsf{F}\colon \{0,1\}^\kappa \times \{0,1\}^\delta \to \{0,1\}$, associating to each value of the domain of the random oracle a bit 0 with probability $1 - 1/q_H$ or 1 with probability $1/q_H$ and then programming the random oracle accordingly as described in game G_1 of Fig. 7. This method of using a biased bit goes back to Coron [14].

G_0: Standard Index Guessing	G_1: Memory-Efficient Index Guessing
Procedure Init	Procedure Init
00 $i^* \xleftarrow{\$} \{1, \ldots, q_H\}$	00 $k \xleftarrow{\$} \{0,1\}^\kappa$
Procedure $\mathsf{RO}(x_i)$	Procedure $\mathsf{RO}(x_i)$
01 If $H[x_i]$ undefined:	01 If $\mathsf{F}(k, x_i) = 0$: Return $\mathsf{P}_0(x_i)$
02 If $i = i^*$: $H[x_i] \leftarrow \mathsf{P}_0(x_i)$	02 Else: Return $\mathsf{P}_1(x_i)$
03 Else: $H[x_i] \leftarrow \mathsf{P}_1(x_i)$	
04 Return $H[x_i]$	

Fig. 7. The random oracle index guessing technique. By x_i we denote the i^{th} query to RO. F is a $1/q_H$-PRF. Note that the queries to RO are not necessarily distinct.

We now compare the two games in terms of success probability, running time and memory efficiency.

Success Probability. Let A be an adversary that is executed in G_0. We define an intermediate game G'_0, as depicted in Fig. 8, in which the index guessing is replaced by tossing a biased coin for each query.

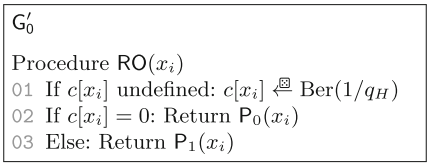

Fig. 8. Intermediate game for the transition to memory-efficient index guessing.

These games are identical if $c[x_{i^*}] = 0$ and $c[x_i] = 1$ for all $i \neq i^*$. Hence,

$$\mathbf{Succ}((\mathsf{G}'_0)^{\mathsf{A}}) \geq (1 - 1/q_H)^{q_H - 1} \cdot \mathbf{Succ}(\mathsf{G}_0^{\mathsf{A}}) \geq e^{-1} \cdot \mathbf{Succ}(\mathsf{G}_0^{\mathsf{A}}).$$

Now it is easy to construct an adversary B against F with $\mathbf{LocalTime}(\mathsf{B}) = \mathbf{LocalTime}(\mathsf{A})$ and $\mathbf{LocalMem}(\mathsf{B}) = \mathbf{LocalMem}(\mathsf{A})$ that provides A with a perfect simulation of $\mathsf{G}_{0'}$ when interacting with game Random_α of Fig. 4 or respectively with a perfect simulation of G_1 when interacting with Real. Hence $\left|\mathbf{Succ}((\mathsf{G}'_0)^{\mathsf{A}}) - \mathbf{Succ}(\mathsf{G}_1^{\mathsf{A}})\right| \leq \mathbf{Adv}(\mathsf{PRF}^{\mathsf{B}}_{1/q_H})$. So overall, we have

$$\mathbf{Succ}(\mathsf{G}_1^{\mathsf{A}}) \geq e^{-1} \cdot \mathbf{Succ}(\mathsf{G}_0^{\mathsf{A}}) - \mathbf{Adv}(\mathsf{PRF}^{\mathsf{B}}_{1/q_H}).$$

Running Time. Game G_1 needs to evaluate the $1/q_H$-PRF q_H times, hence we have $\mathbf{TotalTime}(\mathsf{G}_1^{\mathsf{A}}) = \mathbf{TotalTime}(\mathsf{G}_0^{\mathsf{A}}) + q_H \cdot \mathbf{Time}(\mathsf{F})$.

Memory. The standard game needs to store an array of size at least $q_H \cdot \lambda$ bits and the integer i^*, while the memory-efficient game only needs additional memory $\mathbf{Mem}(\mathsf{F})$. So overall, we have

$$\mathbf{TotalMem}(\mathsf{G}_0^{\mathsf{A}}) \geq \mathbf{LocalMem}(\mathsf{A}) + q_H + 1,$$
$$\mathbf{TotalMem}(\mathsf{G}_1^{\mathsf{A}}) = \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{F}).$$

Note that for simplicity we ignored the memory consumption and running time for procedures P_0 and P_1.

3.5 Single Rewinding Technique

This technique can be used for games containing a procedure Query, which can be called by an adversary A up to q times on inputs $x_1, \ldots, x_q$. When A terminates, it queries Fin on a value x^*. Procedure Fin then checks whether there exists $i \in \{1, \ldots, q\}$ such that $\mathsf{R}(x_i, x^*) = 1$, where R is an efficiently computable

relation specific to the game. If so, it invokes Stop with 1. If no such i exists it invokes Stop with 0. Note that we do not specify how queries to Query are answered since it is not relevant here. To be able to check whether there exists an i such that $\mathsf{R}(x_i, x^*) = 1$, the game usually stores the values $x_1, \ldots, x_q$ as described in G_0 in Fig. 9.

However it is possible to make the game memory efficient as described in G_1 of Fig. 9. In this variant the game no longer stores all the x_i's. Instead, it only stores the adversarial input x^* to Fin and then *rewinds* A to the start, i.e., it runs it a second time providing it with the *exact same input and random coins*, and responding to queries to Query with the *same values* as in the first run. This means that from the adversary's view, the second run is an exact replication of the first one. Whenever A calls Query on a value x_i, the game checks whether $\mathsf{R}(x^*, x_i) = 1$ and—if so—invokes Stop with 1. Note that it is necessary to store the random coins given to A as well as random coins potentially used to answer queries to Query to be able to rewind. This can be done memory-efficiently with the technique of Sect. 3.2.

Standard Game $\mathsf{G}_0^{\mathsf{A}}$	**Memory-efficient Game** $\mathsf{G}_1^{\mathsf{A}}$
Procedure $\mathsf{Query}(x_i)$	Procedure $\mathsf{Query}(x_i)$
00 $X_i \leftarrow x_i$	00 During rewinding:
01 ...	01 If $\mathsf{R}(X^*, x_i) = 1$: Stop with 1
	02 ...
Procedure $\mathsf{Fin}(x^*)$	Procedure $\mathsf{Fin}(x^*)$
02 For $i = 1$ to q	03 $X^* \leftarrow x^*$
03 If $\mathsf{R}(x^*, X_i) = 1$: Stop with 1	04 Rewind A to start
04 Stop with 0	05 Stop with 0

Fig. 9. The single rewinding technique.

Success Probability. Since after rewinding, G_1 provides A with the exact same input as in the first execution, all values x_i are the same in both executions of A, so

$$\mathbf{Succ}(\mathsf{G}_0^{\mathsf{A}}) = \mathbf{Succ}(\mathsf{G}_1^{\mathsf{A}}).$$

Running Time. G_0 runs A once, while G_1 runs A twice. Both games invoke the relation algorithm R a total number of q times, so overall we obtain

$$\mathbf{TotalTime}(\mathsf{G}_1^{\mathsf{A}}) \leq 2 \cdot \mathbf{TotalTime}(\mathsf{G}_0^{\mathsf{A}}).$$

Memory. $\mathsf{G}_0^{\mathsf{A}}$ stores all values $x_1, \ldots, x_q, x^*$ while $\mathsf{G}_1^{\mathsf{A}}$ only stores x^* and one of the $x_i, 1 \leq i \leq q$ at a time. Assuming each of the values $x_1, \ldots, x_q, x^*$ takes one memory unit, we obtain

$$\mathbf{TotalMem}(\mathsf{G}_0^{\mathsf{A}}) = \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{R}) + q + 1,$$
$$\mathbf{TotalMem}(\mathsf{G}_1^{\mathsf{A}}) = \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{R}) + 2.$$

We remark that the single rewinding technique can be extended to a multiple-rewinding technique, in which the reduction runs the adversary m times (on the same random coins and with the same input). For example, in Theorem 4 we consider a reduction between t-multi-collision-resistance and t-collision-resistance that rewinds the adversary several times.

4 Streaming Algorithms and Memory-Efficiency

In this section we prove two lower bounds on the memory usage of black-box reductions between certain problems. The first shows that any reduction from mUFCMA to UFCMA must either use more memory, run the adversary many times, or obey some tradeoff between the two options. The second gives a similar result for mCR_t to CR_t reductions. We start by recalling results from the data-stream model of computation which will provide the principle tools for our lower bounds.

In this section we also deal with bit-memory ($\mathbf{Mem}_2$) which measures the number of bits used, rather than $\mathbf{Mem}$ which measures the number of λ-bit words used.

4.1 The Data Stream Model

The *data stream model* is typically used to reason about algorithmic challenges where a very large input can only be accessed in discrete pieces in a given order, possibly over multiple passes. For instance, data from a high-rate network connection may often be too large to store and thus only accessed in sequence.

Streaming formalization. We adopt the following notation for a streaming problem: An input is a vector $\mathbf{y} \in U^n$ of dimension n over some finite universe U. We say that the number of elements in the stream is n. An algorithm B accesses $\mathbf{y}$ via a stateful oracle $\mathsf{O}_{\mathbf{y}}$ that works as follows: On the first call it saves an initial state $i \leftarrow 0$ and returns $\mathbf{y}[0]$. On future calls, $\mathsf{O}_{\mathbf{y}}$ sets $i \leftarrow (i+1 \mod n)$, and returns $\mathbf{y}[i]$. The oracle models accessing a stream of data, one entry at a time. When the counter i is set to 0 (either at the start or by wrapping modulo n), the algorithm B is said to be initiating a *pass* on the data. The *number of passes* during a computation $\mathsf{B}^{\mathsf{O}_{\mathbf{y}}}$ is thus defined as $p = \lceil q/n \rceil$, where q is the number of queries issued by B to its oracle.

A streaming lower bound. Below we will use a well-known result lower bounding the trade-off between the number of passes and memory required to determining the most frequent element in a stream. We will also use a lower bound on a related problem that can be proven by the same techniques.

For a vector $\mathbf{y} \in U^n$, define $F_\infty(\mathbf{y})$ as

$$F_\infty(\mathbf{y}) = \max_{s \in U} |\{i : \mathbf{y}[i] = s\}|.$$

That is, $F_\infty(\mathbf{y})$ is the number of appearances of the most frequent value in $\mathbf{y}$. Our results will use the following modified version of F_∞, denoted $F_{\infty,t}$ that

only checks if the most frequent value appears t times or not:

$$F_{\infty,t}(\mathbf{y}) = \begin{cases} 1 & \text{if } F_\infty(\mathbf{y}) \geq t \\ 0 & \text{otherwise} \end{cases}$$

We also define the function $G(\mathbf{y})$ as follows. It divides its input into two equal-length halves $\mathbf{y} = \mathbf{y}_1 \| \mathbf{y}_2$, each in $U^{n/2}$. We let

$$G(\mathbf{y}_1 \| \mathbf{y}_2) = \begin{cases} 1 & \text{if } \exists j \ \forall i : \mathbf{y}_2[j] \neq \mathbf{y}_1[i] \\ 0 & \text{otherwise} \end{cases}.$$

In words, G outputs 1 whenever $\mathbf{y}_2$ contains an entry that is not in $\mathbf{y}_1$.

Theorem 1 (Corollary of [21,24]). *Let t be a constant and B be a randomized algorithm such that for all $\mathbf{y} \in U^n$,*

$$\Pr[\mathsf{B}^{\mathsf{O}_\mathbf{y}}(|U|, n) = F_{\infty,t}(\mathbf{y}))] \geq c,$$

where $1/2 < c \leq 1$ is a constant. Then $\mathbf{LocalMem}_2(\mathsf{B}) = \Omega(\min\{n/p, |U|/p\})$, where p is the number of passes B makes in the worst case. The same statement holds if $F_{\infty,t}$ is replaced with G.

This theorem is actually a simple corollary of a celebrated result on the communication complexity of the disjointness problem, which has several other applications. See also the lecture notes by Roughgarden [25] that give an accessible theorem statement and discussion after Theorem 4.11 of that document.

The standard version of this theorem only states that computing F_∞ requires the stated space, so we sketch how to obtain our easy corollary. The full proof is omitted from this version due to the page limit. The proof for F_∞ works by showing that any p-pass streaming algorithm with local memory m can be used to construct a p-round two-party protocol to compute whether sets S_1, S_2 held by the parties are disjoint. One then proves a communication lower bound on any protocol to test for disjointness.

A simple modification of this argument shows that computing G also gives such a protocol: It easily allows two parties to compute if $S_1 \setminus S_2$ is empty, which is equivalent to computing if $\overline{S_1}$ and S_2 are disjoint. Thus one can reduce disjointness to this problem by having the first party take the compliment of its set.

The modification for $F_{\infty,t}$ is slightly more subtle. The essential idea is that one party can copy its set $t-1$ times when feeding it to the streaming algorithm. Then if the parties' sets are not disjoint, we will have $F_{\infty,t}$ equal to 1 and 0 otherwise. Since t is a constant this affects the lower bound by only a constant factor.

4.2 mUFCMA-to-UFCMA Lower Bound

Black-box reductions for mUFCMA to UFCMA. Let R be an algorithm playing the UFCMA game. Recall that R receives input pk and has access to

an oracle $\mathsf{ProcSign}$, and stops the game by querying $\mathsf{Fin}(m^*, \sigma^*)$. Below for an adversary A playing mUFCMA, we write R^{A} to mean that R has additionally "oracle access to A", which means an oracle $\mathsf{NxtQ}_{\mathsf{A}}$ that returns the "next query" of A after accepting a response to the previous query from R. When A halts (i.e. $\mathsf{NxtQ}_{\mathsf{A}}$ returns a query to Fin), the oracle resets itself to start again with the same random tape and input pk.

Definition 2. *A* restricted black-box reduction from mUFCMA to UFCMA *for signature scheme* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *is an oracle algorithm* R*, playing* UFCMA*, that respects the following restrictions for any* A*:*

1. R^{A} *starts by forwarding its initial input (consisting of the security parameter and public key) to* $\mathsf{NxtQ}_{\mathsf{A}}$*.*
2. *When the oracle* $\mathsf{NxtQ}_{\mathsf{A}}$ *emits a query for* $\mathsf{ProcSign}(m)$*,* R *forwards* m *to its own signing oracle* $\mathsf{ProcSign}$ *and returns the result to* $\mathsf{NxtQ}_{\mathsf{A}}$*, possibly after some computation.*
3. *When* $\mathsf{NxtQ}_{\mathsf{A}}$ *emits a query for* $\mathsf{ProcVer}(m^*, \sigma^*)$*,* R *performs some computation then returns an empty response to* $\mathsf{NxtQ}_{\mathsf{A}}$*.*
4. *When* R *queries* $\mathsf{Fin}(m^*, \sigma^*)$*, the value* (m^*, σ^*) *will be amongst the values that* $\mathsf{NxtQ}_{\mathsf{A}}$ *returned as a query to* $\mathsf{ProcVer}$*.*

Finally we say that R *is* advantage-preserving *if there exists an absolute constant* $1/2 < c \leq 1$ *such that for all adversaries* A *and all random tapes* r *for* A*,*

$$\mathbf{Succ}(\mathsf{UFCMA}^{\mathsf{R}^{\mathsf{A}}} \mid r) \geq c \cdot \mathbf{Succ}(\mathsf{mUFCMA}^{\mathsf{A}} \mid r), \tag{2}$$

where $\mathbf{Succ}(\cdot \mid r)$ *is exactly* $\mathbf{Succ}(\cdot)$ *conditioned on the tape of* A *being fixed to* r*.*

These restrictions force R to behave in a combinatorial manner that is amenable to a connection to streaming lower bounds. The final condition, requiring R to preserve the advantage of A for all random tapes, is especially restrictive. At the end of the section we discuss directions for considering more general R.

Theorem 2. *Let* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *be any signature scheme with message length* $\delta = \lambda$*. Let* R *be a restricted black-box reduction from* mUFCMA *to* UFCMA *that is advantage-preserving, and let* p *be the number of times* R *runs* A*. Then for any*[1] $q = q(\lambda)$ *there exists an adversary* A^* *making* q *signing queries, and using memory* $\mathbf{LocalMem}_2(\mathsf{A}^*) = O(\mathbf{LocalMem}_2(\mathsf{Ver}))$*, such that* $\mathbf{LocalMem}_2(\mathsf{R}^{\mathsf{A}^*}) =$

$$\Omega(\min\{\frac{q}{p+1}, \frac{2^\lambda}{p+1}\}) - O(\log q) - \max\{\mathbf{LocalMem}_2(\mathsf{Gen}), \mathbf{LocalMem}_2(\mathsf{Ver})\}.$$

Proof. Let R be a restricted black-box reduction for $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ that is advantage-preserving for some $c \geq 1/2$. We proceed fixing an adversary A^* and using $\mathsf{R}^{\mathsf{A}^*}$ to construct a streaming algorithm B, making $p+1$ passes on its stream, such that

$$\Pr[\mathsf{B}^{\mathsf{O}_{\mathbf{y}}}(2^\delta, n) = G(\mathbf{y})] \geq c \tag{3}$$

[1] We assume that q is linear-space constructible.

for all n and all $\mathbf{y} \in (\{0,1\}^\lambda)^n$. We will apply the streaming lower bound on computing G (Theorem 1) to B, and then relate the memory used by B to that of $\mathsf{R}^{\mathsf{A}^*}$ to obtain the theorem.

We start by fixing the adversary A^*. It takes as input the security parameter λ and public key pk. Then A^* selects q random messages $m_1, \ldots, m_q$, and queries them to $\mathsf{ProcSign}$, and ignores the outputs. Next A^* selects q more random messages $m'_1, \ldots, m'_q$, and for each m'_j it forges a signature σ'_j by brute force and queries (m'_j, σ'_j) to $\mathsf{ProcVer}$. After the verification queries, it halts.

We record two facts about A^*. Let $\mathbf{y} \in (\{0,1\}^\lambda)^{2q}$ the vector consisting of all of its queried messages, in order (the first q to $\mathsf{ProcSign}$, and the second q to $\mathsf{ProcVer}$ along with signatures). First, if $G(\mathbf{y}) = 0$, then $\mathbf{Succ}(\mathsf{mUFCMA}^{\mathsf{A}^*} \mid \mathbf{y}) = 0$ because A^* will not issue any queries with a fresh forgery. If however $G(\mathbf{y}) = 1$, then $\mathbf{Succ}(\mathsf{mUFCMA}^{\mathsf{A}^*} \mid \mathbf{y}) = 1$ because A^* will issue at least one fresh forgery to the verification oracle.

Algorithm $\mathsf{B}^{\mathsf{O}_\mathbf{y}}$ will run $\mathsf{R}^{\mathsf{A}^*}$, which expects input pk, oracles for $\mathsf{ProcSign}$, Fin (for the UFCMA game) and oracle $\mathsf{NxtQ}_{\mathsf{A}^*}$ for an adversary. $\mathsf{B}^{\mathsf{O}_\mathbf{y}}$ works as follows, on input $(2^\lambda, n := 2q)$:

- B starts by initializing a $\log n$-bit counter $i \leftarrow 0$, running $(pk, sk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$, and running R on input pk.
- B responds the oracle query $\mathsf{ProcSign}(m)$ from R by returning $\mathsf{Sign}(sk, m)$.
- When R queries $\mathsf{NxtQ}_{\mathsf{A}^*}$, B ignores the input and responds as follows:
  - If $i < n/2$, then B queries $\mathsf{O}_\mathbf{y}$, which returns $\mathbf{y}_1[i]$, and has $\mathsf{NxtQ}_{\mathsf{A}^*}$ return $\mathsf{ProcSign}(\mathbf{y}_1[i])$ as the next query.
  - If $i \geq n/2$, it queries $\mathsf{O}_\mathbf{y}$ to get $\mathbf{y}_2[j]$ (where $j = i - n/2$). Then B computes a valid signature σ_j by brute force, and increments i modulo n. It then has $\mathsf{NxtQ}_{\mathsf{A}^*}$ return $\mathsf{ProcVer}(\mathbf{y}_2[j], \sigma)$ as the next query.
- When R queries $\mathsf{Fin}(m^*, \sigma^*)$, B performs another pass on its stream and checks if m^* appears anywhere in $\mathbf{y}_1$. If it does, then it outputs 0 and otherwise it outputs 1.

We now verify (3). If $G(\mathbf{y}) = 0$ then $\mathsf{B}^{\mathsf{O}_\mathbf{y}}$ will output 0 with probability 1. This is because our restrictions on R, which restricts it to outputting a value m^* that was queried by A^* to $\mathsf{ProcVer}$. On the other hand, if $G(\mathbf{y}) = 1$ then $\mathsf{B}^{\mathsf{O}_\mathbf{y}}$ will output 1 with probability at least c. This is because A^* will have success probability 1 when such a $\mathbf{y}$ is fixed, so by (2) $\mathsf{R}^{\mathsf{A}^*}$ has success probability at least c, and B outputs 1 whenever R succeeds in the simulated mUFCMA game.

It is clear that B makes $p+1$ passes on its stream, where p is the number of times $\mathsf{R}^{\mathsf{A}^*}$ runs A^*. Applying Theorem 1 to B we have

$$\mathbf{LocalMem}_2(\mathsf{B}) = \Omega(\min\{n/(p+1), 2^\lambda/(p+1)\}).$$

On the other hand, by the construction of B we have that $\mathbf{LocalMem}_2(\mathsf{B})$

$$= O(\mathbf{LocalMem}_2(\mathsf{R}^{\mathsf{A}^*})) + \max\{\mathbf{LocalMem}_2(\mathsf{Gen}), \mathbf{LocalMem}_2(\mathsf{Ver})\})$$

Combining the two bounds on $\mathbf{LocalMem}_2(\mathsf{B})$, and noting that $q = \Theta(n)$, gives the theorem. □

4.3 mCR_t-to-CR_t Lower Bound

Black-box reductions for mCR_t to CR_t. Similar to the case with signatures, we formalize a class of reductions from mCR_t to CR_t for a hash function H. Let R be an oracle algorithm R^{A} that play the CR_t game (with the only oracle being Fin), and additionally has access to an oracle $\mathsf{NxtQ}_{\mathsf{A}}$ that returns the next query or some adversary playing the game mCR_t. The only oracles in mCR_t are $\mathsf{ProcInput}$ and Fin, so $\mathsf{NxtQ}_{\mathsf{A}}$ either returns a domain point m or halts A. As before, the oracle resets itself after the last query by A, with the same input and random tape.

Definition 3. *A* restricted black-box reduction from mCR_t to CR_t *for a hash function* H *is an oracle algorithm* R*, playing* CR_t*, that respects the following restrictions for any* A*:*

1. R^{A} *starts by forwarding its initial input (consisting of the security parameter and hashing key) to* $\mathsf{NxtQ}_{\mathsf{A}}$.
2. *When* R *queries* $\mathsf{Fin}(m_1, \ldots, m_t)$*, the values* $m_1, \ldots, m_t$ *will be amongst the values that* $\mathsf{NxtQ}_{\mathsf{A}}$ *returned as a query to* $\mathsf{ProcInput}$.

Finally we say that R *is* advantage-preserving *if there exists an absolute constant* $1/2 < c \leq 1$ *such that for all adversaries* A *and all random tapes* r *for* A*,*

$$\mathbf{Succ}(\mathsf{mCR}_t^{\mathsf{R}^{\mathsf{A}}} \mid r) \geq c \cdot \mathbf{Succ}(\mathsf{CR}_t^{\mathsf{A}} \mid r), \tag{4}$$

where $\mathbf{Succ}(\cdot \mid r)$ *is exactly* $\mathbf{Succ}(\cdot)$ *conditioned on the tape of* A *being fixed to* r*.*

Theorem 3. *Let* H *be the function (with empty hash key) that truncates the last* λ *bits of its input. Let* R *be a restricted black-box reduction from* mCR_t *to* CR_t *that is advantage-preserving and let* p *be the number of times* R *runs* A*. Then for any*[2] $q = q(\lambda) \leq 2^{\lambda}$ *there exists an adversary* A^* *making* q *signing queries, and using memory* $\mathbf{LocalMem}_2(\mathsf{A}^*) = O(\lambda)$*, such that*

$$\mathbf{LocalMem}_2(\mathsf{R}^{\mathsf{A}^*}) = \Omega(\min\{q/p, 2^{\lambda}/p\}).$$

Proof. We proceed similarly to the proof of Theorem 2, but we now construct a streaming algorithm $\mathsf{B}^{\mathsf{O}_{\mathbf{y}}}$ for $F_{\infty,t}$ instead of G. Let R be a restricted black-box reduction for H that is advantage-preserving for some $c \geq 1/2$. We will fix an adversary A^* and use $\mathsf{R}^{\mathsf{A}^*}$ to construct a streaming algorithm B, making p passes on its stream, such that

$$\Pr[\mathsf{B}^{\mathsf{O}_{\mathbf{y}}}(2^{\delta}, n) = F_{\infty,t}(\mathbf{y})] \geq c \tag{5}$$

for all n and all $\mathbf{y} \in (\{0,1\}^{\lambda})^n$.

[2] We again assume that q is linear-space constructible.

The adversary A^* works as follows: On input λ (and empty hash key), it chooses q random messages $m_1, \ldots, m_q$ and queries $m_i \| i$ to its $\mathsf{ProcInput}$ oracle, where i is encoded in λ bits. It then queries Fin and halts.

Let $\mathbf{y} \in (\{0,1\}^\lambda)^q$ be the vector consisting of all of messages queried to $\mathsf{ProcInput}$. If $F_{\infty,t}(\mathbf{y}) = 0$, then $\mathbf{Succ}(\mathsf{mCR}_t^{\mathsf{A}^*}|\mathbf{y}) = 0$ because there will be no t-collision in the queries of A^*. If however $F_{\infty,t}(\mathbf{y}) = 1$, then $\mathbf{Succ}(\mathsf{mUFCMA}^{\mathsf{A}^*}|\mathbf{y}) = 1$ because A^* there will be a t-collision, as the hash function H is defined to truncate the final λ bits of its inputs, which consist of the counter value.

The streaming algorithm $\mathsf{B}^{\mathsf{O}_\mathbf{y}}(2^\lambda, q)$ works as follows. It initializes a counter i to 0 and runs R. When R requests an input from $\mathsf{NxtQ}_{\mathsf{A}^*}$, $\mathsf{B}^{\mathsf{O}_\mathbf{y}}$ queries its oracle for $\mathbf{y}[i]$ and returns $\mathbf{y}[i] \| i$ to R. When R halts by calling $\mathsf{Fin}(m_1, \ldots, m_t)$, $\mathsf{B}^{\mathsf{O}_\mathbf{y}}$ simply checks if the messages are all of the form $y \| i$ for a fixed y and different values of i. If so, it outputs 1 and otherwise it outputs 0.

It is easy to verify that B satisfies (5) and that it makes p passes on its input stream. Therefore by Theorem 1 we have

$$\mathbf{LocalMem}_2(\mathsf{B}) = \Omega(\min\{q/p, 2^\lambda/p\}).$$

By construction we also have

$$\mathbf{LocalMem}_2(\mathsf{B}) = O(\mathbf{LocalMem}_2(\mathsf{R}^{\mathsf{A}^*})).$$

Combining these inequalities gives the theorem. □

Sharpness of the bounds. We observe that when one is not concerned with memory-tightness then it is trivial to reduce t-multi-collision-resistance to t-collision-resistance, by simply storing all inputs to $\mathsf{ProcInput}$ and checking for collisions. This will however be non-tight if the mCR_t adversary uses small memory but produces a large number of domain points (i.e. q is large). Memory tightness can be achieved via rewinding $O(q)$ times, but this increases the runtime of the reduction.

Theorem 4. *Let* $\mathsf{H}\colon \{0,1\}^\kappa \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ *be a hash function and let* t *be a fixed natural number. Then for all adversaries* A *in the* mCR_t *game with parameter* λ *making* q *queries to* $\mathsf{ProcInput}$ *and for all natural numbers* $1 \le c, p, m \le q < 2^\lambda$ *such that* $c \cdot p \cdot m = q$ *there exists an adversary* B *in the* CR_t *game such that*

$$\begin{aligned}
\mathbf{Succ}(\mathsf{CR}_t^{\mathsf{B}}) &\ge \frac{1}{2c} \cdot \mathbf{Succ}(\mathsf{mCR}_t^{\mathsf{A}}),\\
\mathbf{LocalTime}(\mathsf{B}) &\le (2p+1) \cdot \mathbf{LocalTime}(\mathsf{A}) + (mp(q+1)+q) \cdot \mathbf{Time}(\mathsf{H})\\
\mathbf{LocalMem}(\mathsf{B}) &= \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{H}) + 3m + t + 3.
\end{aligned}$$

If we choose $c = 1$ and $m = q/p$, this theorem proves that the lower bound from Theorem 3 is sharp.

```
Adversary B
00 k ←$ Init_{CR_t}
01 FOR ℓ = 1 to p:
02   sample distinct i_1, ..., i_m from {1, ..., q}
03   run A on input k
04   FOR j = 1 to m:
05     x_j ← H(k, A(i_j))
06   run A on input k
07   FOR i = 1 to q:
08     FOR j = 1 to m:
09       IF x_j = H(k, A(i)) ∧ i_j ≠ i:
10         c_j ← c_j + 1
11         IF c_j = t:
12           run A on input k
13           store all of A's t outputs y_1 ... y_t such that H(y_α) = x_j
14           Stop with y_1 ... y_t
```

Fig. 10. Adversary B in the CR_t game. By $\mathsf{A}(j)$ we denote the j-th out of q inputs of A to ProcInput.

Proof. By assumption $m = q/cp$. Let A be an adversary in the mCR_t game. For simplicity we assume that A is deterministic, otherwise we can apply the PRF coin fixing technique from Sect. 3.2.

Consider adversary B as defined in Fig. 10. First, B stores the hash values of m out of the q inputs of A to ProcInput. Note that A only needs to be run once to perform these operations in line 05, as the indices i_1 to i_m can be sorted. Then it rewinds A to the start and checks for collisions of the stored hash values with all of the hash values of A's inputs to ProcInput. Assume that at least t of A's inputs have the same hash value. Then in each execution of the loop starting in line 01 B succeeds in finding the colliding messages if it stored the corresponding hash value. The probability of this event is bounded from below by $m/q = 1/cp$. The loop is repeated p times with freshly sampled $i_1, \ldots, i_m$. Thus

$$\Pr[\mathsf{CR}_t^{\mathsf{B}} \Rightarrow 1 \mid \mathsf{mCR}_t^{\mathsf{A}} \Rightarrow 1] \geq 1 - (1 - 1/cp)^p \geq 1 - e^{-1/c} \geq 1/2c.$$

This implies $\mathbf{Succ}(\mathsf{CR}_t^B) \geq 1/2c \cdot \mathbf{Succ}(\mathsf{mCR}_t^A)$. When B finds a collision, it rewinds A one last time to obtain the preimages of the t colliding values.

So overall, B runs A at most $2p + 1$ times and the hash algorithm H at most $p(m + mn) + q$ times. It needs to store $2m + 3$ counters of size $\log q \leq \lambda$ (i.e. $2m + 3$ memory units), m values from H's range $\{0,1\}^\rho$ (i.e. m memory units) and the t elements from $\{0,1\}^\delta$ that collide under H (i.e. t memory units) and provide memory for A and H. □

Limitations, extensions, and open problems. Our notion of black-box reductions assumes that the reduction will only run the adversary A from beginning to end, each time with the same random tape. It would be interesting to generalize the reduction to allow for partial rewinding of A, and also for saving "snapshots" of the state of A that allow for rewinding.

Our restrictions on black-box reductions confine them to essentially work like combinatorial streaming algorithms. It seems likely that these restrictions can be greatly relaxed by using a different notion of black-box reduction and using pathological (unbounded) signature schemes and hash functions to enforce the combinatorial behavior of the reduction with high probability. We pursued our version of the results for simplicity.

5 Memory-Tight Reduction for RSA Full Domain Hash Signatures

This section gives an example of a memory-tight reduction obtained via the techniques of Sect. 3. We first recall the syntax of signature schemes and recall the RSA assumption. Then we show how the RSA Full Domain Hash signature scheme can be shown secure in the random oracle model using coin replacement, random oracle replacement, single rewinding, and the random oracle index guessing technique. For subtle reasons we implement all techniques using a single PRF to obtain a memory tight reduction.

Signature schemes. A *signature scheme* consists of algorithms Gen,Sign,Ver such that: algorithm Gen generates a verification key pk and a signing key sk; on input of a signing key sk and a message m algorithm Sign generates a signature σ or the failure indicator $\perp$; on input of a verification key pk, a message m, and a candidate signature σ, deterministic algorithm Ver outputs 0 or 1 to indicate rejection and acceptance, respectively. A signature scheme is correct if for all sk, pk, m, if $\mathsf{Sign}(sk, m)$ outputs a signature then Ver accepts it. Recall that the standard security notion of existential unforgeability against chosen message attacks is defined in Sect. 2.3 via the game of Fig. 2.

RSA assumption. Let GenRSA_λ be an algorithm that returns $(N = pq, e, d)$, where p and q are distinct primes of bit size $\lambda/2$ and e, d are such that $e = d^{-1} \bmod \Phi(N)$.

Definition 4 (RSA Assumption). *Game* RSA_λ *defining the hardness of RSA relative to* GenRSA_λ *is depicted in Fig. 11.*

Game RSA_λ

Procedure Init	Procedure $\mathsf{Fin}(x^*)$
00 $(N, e, d) \xleftarrow{\$} \mathsf{GenRSA}_\lambda$	04 If $x = x^*$:
01 $x \xleftarrow{\$} \mathbb{Z}_N$	05 Stop with 1
02 $y \leftarrow x^e \bmod N$	06 Stop with 0
03 Return (N, e, y)	

Fig. 11. The RSA_λ game relative to algorithm GenRSA_λ.

RSA-FDH. The RSA Full Domain Hash (RSA-FDH) signature scheme [7] is defined in Fig. 12. Its security can be reduced to the RSA assumption in the random oracle model (see [8,14]). In the usual proof the reduction interacting with an adversary against RSA-FDH's existential unforgeability making up to q_H hash queries and up to q_s signing queries simulates the random oracle using lazy sampling and therefore has to store up to $(q_H + q_s)$ messages making the reduction highly non-memory-tight. However, the proof can be made memory-efficient by using the coin replacement technique of Sect. 3.2, the random oracle technique of Sect. 3.3, the random oracle index guessing technique of Sect. 3.4, and the single rewinding technique of Sect. 3.5.

Gen	$\mathsf{Sign}(sk, m)$	$\mathsf{Ver}(pk, m, \sigma)$
00 $(N, e, d) \xleftarrow{\$} \mathsf{GenRSA}_\lambda$	04 $(N, d) \leftarrow sk$	07 $(N, e) \leftarrow pk$
01 $pk \leftarrow (N, e)$, $sk \leftarrow (N, d)$	05 $\sigma \leftarrow \mathsf{H}(m)^d \bmod N$	08 If $\sigma^e = \mathsf{H}(m) \bmod N$:
02 Pick RO $\mathsf{H}\colon \{0,1\}^\lambda \to \mathbb{Z}_N$	06 Return σ	09 Return 1
03 Return (pk, sk)		10 Return 0

Fig. 12. The RSA-FDH signature scheme for parameter λ.

Theorem 5. *Let* $\mathsf{F}\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^{2\lambda+1}$ *be a PRF. Then for every adversary* A *in the* UFCMA *game for RSA-FDH with parameter* λ *that poses* q_H *queries to the* Hash, q_s *queries to the* ProcSign *oracle, and samples at most* $L \leq 2^\lambda$ *memory units of randomness, in the random oracle model there exist an adversary* B_1 *against the* RSA_λ *game, an adversary* B_2 *against the* PRF *game such that*

$$\mathbf{Succ}(\mathsf{UFCMA}^{\mathsf{A}}) \leq e\, q_s\, \mathbf{Succ}(\mathsf{RSA}_\lambda^{\mathsf{B}_2}) + e\, q_s\, \mathbf{Adv}(\mathsf{PRF}^{\mathsf{B}_1}).$$

Further it holds that

$$\begin{aligned}
\mathbf{LocalMem}(\mathsf{B}_1) &= \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{GenRSA}_\lambda) + 6,\\
\mathbf{LocalMem}(\mathsf{B}_2) &= \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{F}) + 6,\\
\mathbf{LocalTime}(\mathsf{B}_1) &\approx 2\mathbf{LocalTime}(\mathsf{A}) + \mathbf{Time}(\mathsf{RSA}_\lambda),\\
\mathbf{LocalTime}(\mathsf{B}_2) &\approx \mathbf{LocalTime}(\mathsf{A}) + (q_H + q_s + L) \cdot \mathbf{Time}(\mathsf{F}).
\end{aligned}$$

Note that in the proof of Theorem 5 it is necessary to apply the random coins technique and the random oracle technique in the same step. Otherwise one obtains an intermediate reduction that is not memory-tight: the reduction either *has to simulate the random oracle by lazy sampling (in case the random coins technique is applied first)* or*, since rewinding is impossible, it has to store the messages asked to the signing oracle (if the random oracle technique is applied first).*

Proof. Consider the sequence of games of Fig. 13. For computations in $\mathbb{Z}_N$ we omit writing $\bmod N$ if it is clear from the context. We assume without loss of

G_0 / G_1	G_2	G_3
Procedure Init	Procedure Init	Procedure Init
00 $(N, e, d) \xleftarrow{\$} \mathsf{GenRSA}_\lambda$	00 $(N, e, d) \xleftarrow{\$} \mathsf{GenRSA}_\lambda$	00 $(N, e, d) \xleftarrow{\$} \mathsf{GenRSA}_\lambda$
01	01 $x \xleftarrow{\$} \mathbb{Z}_N$	01 $x \xleftarrow{\$} \mathbb{Z}_N$
02	02 $y \leftarrow x^e$	02 $y \leftarrow x^e$
03 $r \xleftarrow{\$} (\{0,1\}^\lambda)^L$	03 $r \xleftarrow{\$} (\{0,1\}^\lambda)^L$	03 $k \xleftarrow{\$} \{0,1\}^{\kappa(N)}$
04 Return (N, e)	04 Return (N, e)	04 Return (N, e)
Procedure $\mathsf{Hash}(m_i)$	Procedure $\mathsf{Hash}(m_i)$	Procedure $\mathsf{Hash}(m_i)$
05 If $H[m_i]$ undefined:	05 If $H[m_i]$ undefined:	05
06 $H[m_i] \xleftarrow{\$} \mathbb{Z}_N$	06 $H[m_i] \xleftarrow{\$} \mathbb{Z}_N$	06
07	07 $B[m_i] \xleftarrow{\$} \mathrm{Ber}(1/q_s)$	07
08	08 If $B[m_i] = 1$:	08 If $\mathsf{F}_2(k, m_i) = 1$:
09 Return $H[m_i]$ (G_0)	09 Return $H[m_i]^e y$	09 Return $\mathsf{F}_1(k, m_i)^e y$
10 Return $H[m_i]^e$ (G_1)	10 Else: Return $H[m_i]^e$	10 Return $\mathsf{F}_1(k, m_i)^e$
Procedure $\mathsf{ProcSign}(m_i)$	Procedure $\mathsf{ProcSign}(m_i)$	Procedure $\mathsf{ProcSign}(m_i)$
11 $M \leftarrow M \cup \{m_i\}$	11 $M \leftarrow M \cup \{m_i\}$	11 $M \leftarrow M \cup \{m_i\}$
12	12 If $B[m_i] = 1$:	12 If $\mathsf{F}_2(k, m_i) = 1$:
13 Return $\mathsf{Hash}(m_i)^d$ (G_0)	13 Abort	13 Abort
14 Return $\mathsf{Hash}(m_i)$ (G_1)	14 Return $\mathsf{Hash}(m_i)$	14 Return $\mathsf{Hash}(m_i)$
Procedure Coins	Procedure Coins	Procedure Coins
15 $j \leftarrow j + 1$	15 $j \leftarrow j + 1$	15 $j \leftarrow j + 1$
16 Return r_j	16 Return r_j	16 Return $\mathsf{F}_0(k, j)$
Procedure $\mathsf{Fin}(m^*, \sigma^*)$	Procedure $\mathsf{Fin}(m^*, \sigma^*)$	Procedure $\mathsf{Fin}(m^*, \sigma^*)$
17	17 If $B[m^*] = 0$:	17 If $\mathsf{F}_2(k, m_i) = 0$:
18	18 Stop with 0	18 Stop with 0
19 If $m^* \in M$:	19 If $m^* \in M$:	19 If $m^* \in M$:
20 Stop with 0	20 Stop with 0	20 Stop with 0
21 If $(\sigma^*)^e = \mathsf{Hash}(m^*)$:	21 If $(\sigma^*)^e = \mathsf{Hash}(m^*)$:	21 If $(\sigma^*)^e = \mathsf{Hash}(m^*)$:
22 Stop with 1	22 Stop with 1	22 Stop with 1
23 Stop with 0	23 Stop with 0	23 Stop with 0

Fig. 13. Games G_0 to G_3 for the proof of Theorem 5.

generality that any message procedures ProcSign or Fin are queried on was before already queried to Hash.

Game G_0 is the standard UFCMA game as in Fig. 2 instantiated with the RSA-FDH algorithms and with the randomness for adversary A provided via procedure Coins, so

$$\mathbf{Succ}(\mathsf{UFCMA}^\mathsf{A}) = \mathbf{Succ}(\mathsf{G}_0^\mathsf{A}). \tag{6}$$

In G_1, instead of returning $\mathsf{H}(m)$, the Hash procedure returns $\mathsf{H}(m)^e$ and the ProcSign procedure computes signatures as $(\mathsf{H}(m)^e)^d = \mathsf{H}(m)$ accordingly. This doesn't change the distribution of the hash values and the signatures, so

$$\mathbf{Succ}(\mathsf{G}_0^\mathsf{A}) = \mathbf{Succ}(\mathsf{G}_1^\mathsf{A}). \tag{7}$$

Game G_2 introduces a couple of aborting conditions. With probability $1/q_s$ abort condition $B[m^*] = 0$ of line 17 does not occur. Furthermore, for each message

m_i the probability that abort condition $B[m_i] = 1$ of line 12 does not occur is given by $1 - 1/q_s$. Adversary A makes at most q_s queries to $\mathsf{ProcSign}$. Hence,

$$\mathbf{Succ}(\mathsf{G}_2^{\mathsf{A}}) \geq 1/q_s(1 - 1/q_s)^{q_s} \cdot \mathbf{Succ}(\mathsf{G}_1^{\mathsf{A}}) \geq 1/(eq_s) \cdot \mathbf{Succ}(\mathsf{G}_1^{\mathsf{A}}). \tag{8}$$

In Game G_3 randomness is replaced by PRF F, whose range we split into $\mathsf{F} = \mathsf{F}_0||\mathsf{F}_1||\mathsf{F}_2 \in \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}$. Sampling of random coins is replaced in Game G_3 by evaluating F_0 on counter j, sampling the values $H[m_i]$ and $B[m_i]$ is replaced by evaluating F_1 and F_2 on m_i, respectively. For simplicity we assume that F_2 is a pseudorandom function that outputs elements in $\mathbb{Z}_N \approx \{0,1\}^\lambda$ and that F_2 is a α-biased pseudorandom function with $\alpha := 1/q_s$. (This is formally not correct but we do not want to distract from the main points of our proof, which is about memory-tightness.) We proceed by constructing an adversary B_1 against the PRF game such that

$$\mathbf{Adv}(\mathsf{PRF}^{\mathsf{B}_1}) \geq |\mathbf{Succ}(\mathsf{G}_2^{\mathsf{A}}) - \mathbf{Succ}(\mathsf{G}_3^{\mathsf{A}})|, \tag{9}$$

$$\mathbf{LocalTime}(\mathsf{B}_1) \approx 2\mathbf{LocalTime}(\mathsf{A}) + \mathbf{Time}(\mathsf{RSA}_\lambda), \tag{10}$$

$$\mathbf{LocalMem}(\mathsf{B}_1) = \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{GenRSA}_\lambda) + 6. \tag{11}$$

The definition of B_1 is in Fig. 14. Adversary B_1 sets up the values (N, e, d) using GenRSA, samples $x \xleftarrow{\$} \mathbb{Z}_N$, sets $y \leftarrow x^e$ and runs A on input (N, e). It simulates the procedures Hash, $\mathsf{ProcSign}$ and Coins by invoking its PRF oracle O_F. When A calls Fin on message-signature pair (m^*, σ^*) adversary B_1 rewinds A to line 03, answering all of its queries in the same way. Note that this is possible, since all replies to queries on Hash, $\mathsf{ProcSign}$ and Coins are derived using O_F. During the rewinding B_1 raises a flag coll if A queries procedure $\mathsf{ProcSign}$ on m^*. Hence the event $\{\mathsf{coll} = 1\}$ is equivalent to condition $m^* \in M$ of line 19 of games G_2 and G_3. When A calls Fin a second time on (m^*, σ^*), adversary B_1 stops with 0 or 1 according to the message-signature pair. If B_1 interacts with PRF-game Random it provides A with a perfect simulation of game G_2, if it interacts with Real with a perfect simulation of game G_3. Hence Eq. (9) follows. We now analyze B_1's running time and memory consumption. B_1 runs GenRSA_λ

B_1

Procedure Init
00 $(N, e, d) \xleftarrow{\$} \mathsf{GenRSA}_\lambda$ (1)
01 $x \xleftarrow{\$} \mathbb{Z}_N$ (1)
02 $y \leftarrow x^e$ (1)
03 Invoke A on (N, e)

Procedure Coins
04 $j \leftarrow j + 1$
05 Return $\mathsf{O}_{\mathsf{F}0}(j))$

Procedure $\mathsf{Hash}(m_i)$
06 If $\mathsf{O}_{\mathsf{F}2}(m_i) = 1$:
07 Return $(\mathsf{O}_{\mathsf{F}1}(m_i))^e \cdot y$
08 Return $(\mathsf{O}_{\mathsf{F}1}(m_i))^e$

Procedure $\mathsf{ProcSign}(m_i)$
09 If $m_i = m^*$: (2)
10 $\mathsf{coll} \leftarrow 1$ (2)
11 If $\mathsf{O}_{\mathsf{F}2}(m_i) = 1$:
12 Abort
13 Return $\mathsf{Hash}(m_i)$

Procedure $\mathsf{Fin}(m^*, \sigma^*)$
14 Store m^*, rewind A (1)
15 If $\mathsf{O}_{\mathsf{F}2}(m_i) = 0$: (2)
16 Stop with 0 (2)
17 If $\mathsf{coll} = 1$: (2)
18 Stop with 0 (2)
19 If $(\sigma^*)^e = \mathsf{Hash}(m^*)$: (2)
20 Stop with 1 (2)
21 Stop with 0 (2)

Fig. 14. Adversary B_1 against the PRF game for the proof of Theorem 5 in Sect. 5. B_1 rewinds A once on the same inputs. Lines marked with (i) are only executed during the i-th invocation.

once and A twice and performs some minor bookkeeping. It furthermore has to store the code of A and GenRSA_λ as well as at any point in time 6λ bits which equals 6 memory units (i.e., the three integers (N, e, y) of size 3λ, up to two messages of length λ each and a counter of size $\log_2(L) \leq \lambda$.

Adversary B_2

Procedure Init
00 $(N, e, y) \xleftarrow{\$} \mathsf{Init}_{\mathsf{RSA}}$
01 $k \xleftarrow{\$} \{0,1\}^{\kappa(N)}$
02 Invoke A on (N, e)

Procedure Coins
03 $j \leftarrow j + 1$
04 Return $\mathsf{F}_1(k, j)$

Procedure $\mathsf{Hash}(m_i)$
05 If $\mathsf{F}_2(k, m_i) = 1$:
06 Return $\mathsf{F}_1(k, m_i)^e y$
07 Return $\mathsf{F}_1(k, m_i)^e$

Procedure $\mathsf{ProcSign}(m_i)$
08 If $\mathsf{F}_2(k, m_i) = 1$:
09 Abort
10 Return $\mathsf{Hash}(m_i)$

Procedure $\mathsf{Fin}(m^*, \sigma^*)$
11 If $\mathsf{F}_2(k, m^*) = 0$:
12 Abort
13 If $(\sigma^*)^e = \mathsf{Hash}(m^*)$:
14 $x^* \leftarrow \sigma^* / \mathsf{F}_1(k, m^*)$
15 Call $\mathsf{Fin}_{\mathsf{RSA}}(x^*)$

Fig. 15. Adversary B_2 against the RSA_λ game for the proof of Theorem 5 in Sect. 5.

We conclude the proof by giving an adversary B_2 against the RSA_λ game such that

$$\mathbf{Succ}(\mathsf{RSA}_\lambda^{\mathsf{B}_2}) \geq \mathbf{Succ}(\mathsf{G}_3^{\mathsf{A}}), \tag{12}$$

$$\mathbf{LocalTime}(\mathsf{B}_2) \approx \mathbf{LocalTime}(\mathsf{A}) + (q_H + q_s + L)\mathbf{Time}(\mathsf{F}) \tag{13}$$

$$\mathbf{LocalMem}(\mathsf{B}_2) = \mathbf{LocalMem}(\mathsf{A}) + \mathbf{Mem}(\mathsf{GenRSA}_\lambda) + 6. \tag{14}$$

Then the claim of the theorem follows from Eq. (7) to (9) and Sect. 5. The definition of B_2 is in Fig. 15. It queries $\mathsf{Init}_{\mathsf{RSA}}$ to receive an RSA challenge (N, e, y) and samples a PRF key k. Then it invokes A on input (N, e) providing it with a perfect simulation of the procedures Hash, $\mathsf{ProcSign}$ and Coins. When A invokes procedure Fin on message-signature pair (m^*, σ^*), adversary B_2 checks whether $\mathsf{F}_2(k, m^*) = 0$ and—if so—aborts. Note that by definition of procedure Hash adversary B_2 not aborting implies that $\mathsf{Hash}(m^*) = (\mathsf{F}_1(k, m^*))^e y$. Hence if B_2 does not abort and if the signature is valid, i.e. $(\sigma^*)^e = \mathsf{Hash}(m^*)$ holds, then B_2's answer $x^* = \sigma / \mathsf{F}_1(k, m^*)$ to the RSA challenge is valid. Since A succeeding in game G_3 implies both aforementioned conditions Sect. 5 follows. We conclude the proof by analyzing B_2's running time and memory consumption. B_2 runs A once and F up to $(q_H + q_s + L)$ times and performs some minor bookkeeping. Furthermore it has to store the code of A and F as well as at any point in time 6λ bits which equals 6 additional memory units (i.e., a counter of bit-size $\log_2(L) \leq \lambda$, a PRF key of bit-size $\kappa \leq \lambda$, a message of bit-size λ and three integers of size λ). □

6 Memory-Sensitive Problems

In this section we discuss the memory sensitivity of two cryptographic problems, multi-collision-resistance and learning parities with noise. In the full version

of this paper [3], we will also analyze the memory sensitivity of the discrete logarithm problem in prime fields and of the factoring problem.

To quantify the memory sensitivity of a problem P we plot time/memory trade-offs as in the Fig. 1. The horizontal axis is memory consumption and the vertical axis is running time, both on a log scale. A point (x, y) is either labeled with "solvable" or "unsolvable", where solvable means that there exists an algorithm with running time at most 2^x and memory consumption at most 2^y that solves the problem. We refer to the boundary between the solvable and unsolvable regions as the *transition line*.

A time/memory trade-off plot of a non-memory-sensitive problem typically has an (approximately) horizontal transition line, and as discused in Sect. 1, a non-memory-tight reduction has less impact. The steeper the slope of the transition line, the more memory-sensitive the problem is. We refer for the introduction for an example with concrete numbers for.

k-Way Collision Resistance. The k-way collision problem CR_k is to find a k-collision in a hash function with λ output bits. The following table provides an overview over known algorithms to solve CR_k with constant success probability for $k \in \{2, 3\}$.

Algorithm A	$\mathbf{Mem}(\mathsf{CR}_t^{\mathsf{A}})$	$\mathbf{Time}(\mathsf{CR}_t^{\mathsf{A}})$
Birthday ($k = 2$)	$O(1)$	$2^{\lambda/2}$
Joux-Lucks [20] ($k = 3$)	2^{α}	$2^{\lambda(1-\alpha)}$ ($\alpha \leq 1/3$)

From the table we derive the time/memory graph of CR_k in Fig. 16. CR_3 is memory sensitive, whereas CR_2 is not (as it has a horizontal transition line).

Learning Parity with Noise. Another example of a memory sensitive problem is the well-known *Learning Parity with Noise (LPN)* problem. Let $\lambda \in \mathbb{N}$ be the dimension and $\tau \in [0, 1/2)$ be a constant that defines the error probability. The problem $\mathsf{LPN}_{\lambda,\tau}$ is to compute a random secret $s \xleftarrow{\$} \mathbb{F}_2^{\lambda}$, given "noisy" random inner products with s, i.e. samples (a_i, ν_i) where $a_i \xleftarrow{\$} \mathbb{F}_2^{\lambda}$, and $\nu_i = \langle a_i, s \rangle + e_i$ for $e_i \xleftarrow{\$} Ber(\tau)$.

Memory usage and running time of the best known algorithms for $\mathrm{LPN}_{\lambda,\tau}$ with constant success probability are given in the following table.

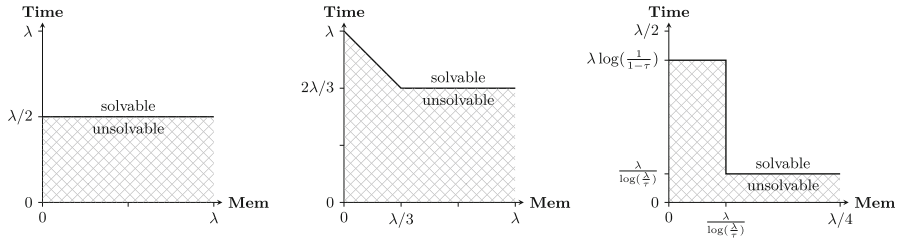

Fig. 16. Time memory graphs of CR_k for $k = 2$ (left) and $k = 3$ (middle) and of $\mathrm{LPN}_{\lambda,\tau}$ for $\lambda = 1024$ and $\tau = 1/4$ (right). Both **Time** and **Mem** are in log scale.

Algorithm A	$\mathbf{LocalMem}(\mathsf{LPN}^{\mathsf{A}}_{\lambda,\tau})$	$\mathbf{LocalTime}(\mathsf{LPN}^{\mathsf{A}}_{\lambda,\tau})$
BKW [10]	$2^{\lambda/\log(\lambda/\tau)}$	$2^{\lambda/\log(\lambda/\tau)}$
Gauss [15]	$O(1)$	$2^{\lambda \log(1/1-\tau)}$

Figure 16 provides the corresponding time/memory graph. Note that the recent work [15] also considers a hybrid algorithm between Well-Pooled Gauss and BKW, but the interval where the hybrid algorithm actually has better performance is so small that we decided to ignore it.

We note that the situation with the Learning with Errors (LWE), the Shortest Integer Solution (SIS), and the approximate SVP problem is similar to that of the LPN problem [2,13,19].

Acknowledgments. The motivation of considering memory in the context of security reductions stems from the talk "Practical LPN Cryptanalysis", given by Alexander May at the Dagstuhl Seminar 16371 on Public-Key Cryptography. We thank Elena Kirshanova, Robert Kübler, and Alexander May for their help with assessing the memory-sensitivity of a number of hard problems. Finally, we are grateful to one of the CRYPTO 2017 reviewers for his/her very detailed and thoughtful review.

Auerbach was supported by the NRW Research Training Group SecHuman; Cash was supported in part by NSF grant CNS-1453132; Fersch and Kiltz were supported in part by ERC Project ERCC (FP7/615074) and by DFG SPP 1736 Big Data.

References

1. Abdalla, M., Bellare, M., Rogaway, P.: The oracle Diffie-Hellman assumptions and an analysis of DHIES. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 143–158. Springer, Heidelberg (2001). doi:10.1007/3-540-45353-9_12
2. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015). http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml
3. Auerbach, B., Cash, D., Fersch, M., Kiltz, E.: Memory-tight reductions. Cryptology ePrint Archive, Report 2017/??? (2017). http://eprint.iacr.org/2017/???
4. Bader, C., Jager, T., Li, Y., Schäge, S.: On the impossibility of tight cryptographic reductions. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 273–304. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49896-5_10
5. Bellare, M., Boldyreva, A., Micali, S.: Public-key encryption in a multi-user setting: security proofs and improvements. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 259–274. Springer, Heidelberg (2000). doi:10.1007/3-540-45539-6_18
6. Bellare, M., Ristenpart, T.: Simulation without the artificial abort: simplified proof and improved concrete security for waters' IBE scheme. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 407–424. Springer, Heidelberg (2009). doi:10.1007/978-3-642-01001-9_24
7. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: Ashby, V. (ed.) ACM CCS 1993, Fairfax, Virginia, USA, 3–5 November 1993, pp. 62–73. ACM Press (1993)

8. Bellare, M., Rogaway, P.: The exact security of digital signatures-how to sign with RSA and rabin. In: Maurer, U. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 399–416. Springer, Heidelberg (1996). doi:10.1007/3-540-68339-9_34
9. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006). doi:10.1007/11761679_25
10. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. J. ACM **50**(4), 506–519 (2003). http://doi.acm.org/10.1145/792538.792543
11. Chatterjee, S., Koblitz, N., Menezes, A., Sarkar, P.: Another look at tightness II: practical issues in cryptography. Cryptology ePrint Archive, Report 2016/360 (2016). http://eprint.iacr.org/2016/360
12. Chatterjee, S., Menezes, A., Sarkar, P.: Another look at tightness. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 293–319. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28496-0_18
13. Chen, Y., Nguyen, P.Q.: BKZ 2.0: better lattice security estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25385-0_1
14. Coron, J.-S.: On the exact security of full domain hash. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 229–235. Springer, Heidelberg (2000). doi:10.1007/3-540-44598-6_14
15. Esser, A., Kübler, R., May, A.: LPN decoded. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10402, pp. 486–514. Springer, Cham (2017)
16. Galbraith, S.D., Gaudry, P.: Recent progress on the elliptic curve discrete logarithm problem. Cryptology ePrint Archive, Report 2015/1022 (2015). http://eprint.iacr.org/2015/1022
17. Galindo, D.: The exact security of pairing based encryption and signature schemes. In: Based on a talk at Workshop on Provable Security, INRIA, Paris (2004). http://www.dgalindo.es/galindoEcrypt.pdf
18. Gay, R., Hofheinz, D., Kiltz, E., Wee, H.: Tightly CCA-secure encryption without pairings. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 1–27. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49890-3_1
19. Herold, G., Kirshanova, E., May, A.: On the asymptotic complexity of solving LWE. Des. Codes Crypt. 1–29 (2017). http://dx.doi.org/10.1007/s10623-016-0326-0
20. Joux, A., Lucks, S.: Improved generic algorithms for 3-collisions. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 347–363. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10366-7_21
21. Kalyanasundaram, B., Schnitger, G.: The probabilistic communication complexity of set intersection. SIAM J. Discrete Math. **5**(4), 545–557 (1992). http://dx.doi.org/10.1137/0405044
22. Pointcheval, D., Stern, J.: Security arguments for digital signatures and blind signatures. J. Cryptol. **13**(3), 361–396 (2000)
23. Pollard, J.M.: A monte carlo method for factorization. BIT Numer. Math. **15**(3), 331–334 (1975). http://dx.doi.org/10.1007/BF01933667
24. Razborov, A.A.: On the distributional complexity of disjointness. Theor. Comput. Sci. **106**(2), 385–390 (1992). http://dx.doi.org/10.1016/0304-3975(92)90260-M
25. Roughgarden, T.: Communication complexity (for algorithm designers) (2015). http://theory.stanford.edu/~tim/w15/l/w15.pdf