On Extract-then-Expand Key Derivation Functions and an HMAC-based KDF *

Hugo Krawczyk†

March 10, 2008

Abstract

This paper describes and provides detailed rationale for a simple (and fully specified) HMAC-based key derivation function (KDF) that can serve multiple applications under a wide variety of requirements and assumptions, and whose design is backed by careful cryptographic analysis. The proposed scheme follows the extract-then-expand paradigm for KDF design (which we present and discuss in great detail) and results in a hash-based KDF whose security relies on as weak as possible assumptions from the underlying hash function. Moreover, the same approach allows for alternative implementations fully or partially based on block ciphers (e.g., AES).

The proposal is intended to address two important and timely needs of crypto applications: (i) providing a single hash-based KDF design that can be standardized for use in multiple and diverse applications, and (ii) providing a conservative, yet efficient, design that exercises much care in the way it utilizes a cryptographic hash function. The bulk of the paper is dedicated to present detailed cryptographic analysis for the proposed design based on recent research in this area and with particular attention to minimizing assumptions on the underlying hash function.

*See `http://www.ee.technion.ac.il/~hugo/kdf/` for updates.

†IBM T.J. Watson Research Center, Hawthorne, New York. Email: hugo@ee.technion.ac.il

Contents

1 Introduction

A Key derivation function (KDF) is a basic and essential component of cryptographic systems: Its goal is to take some *source of initial keying material,* usually containing some good amount of randomness, but not distributed uniformly or for which an attacker has some partial knowledge, and derive from it one or more *cryptographically strong* secret keys. The number and lengths of such keys depend on the specific cryptographic algorithms for which the keys are needed. We associate the notion of "cryptographically strong" keys with that of *pseudorandom* keys, namely, keys that cannot be distinguished by feasible computational means from a random uniform string of the same length.[1] In particular, knowledge of part of the bits, or keys, output by the KDF should not leak information on the other generated bits. Typical examples of initial keying material are a Diffie-Hellman value computed in a key exchange protocol, a bit sequence obtained by a statistical sampler (such as sampling system events or user keystrokes), the output of an imperfect physical random number generator, and more.

The main difficulty in designing a KDF relates to the form of the initial keying material (which we refer to as source key material). When this material is given as a uniformly random or pseudorandom key K then one can use K to key a pseudorandom function (PRF) to produce additional cryptographic keys. However, when the source keying material is not uniformly random or pseudorandom then the KDF needs to first "extract" from this imperfect keying material a first pseudorandom key from which further keys can be derived using a PRF. Thus, we identify two logical modules in a KDF: a first module that takes the source keying material and extracts from it a fixed-length pseudorandom key K, and a second module that expands the key K into several additional pseudorandom cryptographic keys.[2]

We observe that most KDF designs in practice *do not explicitly differentiate between the extract and expand phases;* rather they combine the two in ad-hoc ways under a single cryptographic hash function (often thought of as an ideal random function). Since this is a fundamental aspect in our analysis and in our comparison to existing KDFs, let us stress this point using schematic representations of our approach and of the traditional one that is found in innumerable applications and standards (e.g., [4, 5, 57, 40]). Our scheme, presented in detail in Section 2, consists of two stages that correspond, respectively, to the extract and expand modules discussed above. It can roughly be described as follows (we will see that our HMAC-based specification in Section 2 further improves upon this sketch in some significant ways):

1. $K = \mathsf{Hash}(SKM)$
2. $KM = \mathsf{PRF}(K, \text{“1”} \parallel info) \parallel \mathsf{PRF}(K, \text{“2”} \parallel info) \parallel \ldots \parallel \mathsf{PRF}(K, \text{“}t\text{”} \parallel info)$

where SKM denotes the source key material, "i" denotes a fixed-length counter, $\parallel$ denotes concatenation, and *info* describes (optional) context- and application-specific information. The value t depends on the amount of generated keying material, KM, required by the application. $\mathsf{PRF}(K, \cdot)$ denotes a pseudorandom function keyed with key K, and Hash is a hash function used to "extract" randomness from the (imperfect) source key material. In our HMAC-based scheme, Hash is replaced with HMAC for some good reasons that we will explain in detail, and we also use HMAC as an implementation of the PRF.

[1] See appendices A and B for formal definitions.

[2] It is not uncommon to see in the literature references to KDF as a function that simply expands a *strong* key into several additional keys; this ignores the extract functionality which is central to a general *multi-purpose* KDF.

In contrast, the traditional approach combines the extraction and expand stages resulting in a repeated application of the hash function on the source key material, namely:

$$KM = \mathsf{Hash}(SKM \parallel \text{“1”} \parallel info) \parallel \mathsf{Hash}(SKM \parallel \text{“2”} \parallel info) \parallel \ldots \parallel \mathsf{Hash}(SKM \parallel \text{“}t\text{”} \parallel info)$$

Here Hash is typically implemented with a plain hash function such as SHA-1. Below, we refer to schemes of the latter form as the “traditional scheme” and to the former as the XtX scheme (for eXtract-then-eXpand).

The main (and essential) difference between the above two schemes is the way in which the function Hash is applied. In both cases we have a single initial value *SKM*, which comes from a distribution with some level of randomness (or *entropy*), and the role of Hash is to *concentrate* this entropy in the output bits. However, while in the traditional scheme Hash is used *repeatedly* to extract *many* such bits (as many as required by the application), in our proposed scheme a *single* application of Hash is used to extract a *minimal* number of bits (i.e., a single key). In other words, if one measures the quality of extraction from *SKM* by the ratio between the number of bits output by Hash(*SKM*) and the input entropy, the XtX scheme performs significantly better than the traditional one. This makes for a substantial difference in the level of security (or strength) that we need to assume from the hash function. For example, extracting a small (relative to the initial entropy) number of bits can be achieved using information-theoretic (i.e., *unconditionally secure*) extractors[3], while extracting a large number (as in the traditional scheme) usually requires the modeling of a hash function as a random oracle. Even in the case, as in our HMAC-based proposal, where one implements the extract stage with a cryptographic hash function instead of an information-theoretic extractor, the required assumptions on the hash function are often significantly more relaxed than those needed to justify the traditional scheme.

Of course, we need to output a large number of bits also in the XtX scheme, so don’t we lose the above advantages in doing so? The point is that the many bits are generated by a PRF (pseudorandom function). PRFs are specifically designed and analyzed to do exactly that: take a single good key and generate many pseudorandom bits as output. Our confidence in practical PRF schemes, such as CBC with block ciphers or HMAC with hash functions, is far higher than the confidence on strong properties of plain hash functions. Very significantly, note that in the traditional scheme, the plain Hash is repeatedly applied to the same input *SKM* with little variation introduced by the counter value. This makes the various inputs to Hash different but still highly correlated. Even the best information-theoretic extractors are not capable of outputting so many bits, even less so under *dependent inputs*. Therefore, we want to avoid as much as possible the assumption that our hash functions do achieve such strong properties. In contrast, in the XtX scheme the dependent inputs go into the PRF since PRFs are designed to remain secure even when the inputs (except for the key) are fully known and correlated, or even adversarially chosen.[4]

Note. It is illustrative to note that when *SKM* is already a good key (as it is the case of some KDF applications, e.g., *SKM* is output by a strong RNG), the XtX scheme can skip the first stage and use *SKM* directly as the PRF key. The traditional scheme, in contrast, would still be using *SKM* with the plain hash function, effectively acting as a PRF in “key-prepended mode”. However, this mode has long been deprecated in favor of schemes such as HMAC. In particular, key-prepended schemes need to care about extension attacks resulting in the need for careful prefix-free encoding specifications. Moreover, such a scheme

[3]The notion of *randomness extractors* [56] has been widely researched in the complexity-theory literature (see [55, 59] for technical surveys) and serves as the basis for our ideas and approach. We expand on this notion, informally but in quite detail, in Section 3 and subsequent sections, and present formal definitions in Appendix A.

[4]We will see later that our actual design is further strengthened by minimizing (via feedback mode) the dependencies, low variability (e.g., low-Hamming differentials), and non-secrecy of the inputs to the PRF.

does not even follow the usual practice of aligning (via padding) the prepended key to a block boundary (as to induce a random IV). The NIST construction [57, 19] (described in Section 9) that swaps the order of the counter and *SKM* is even less conventional as a PRF.

Summarizing the above discussion, it is one of the main recommendations in this work, and a basis for the specific design presented here, that a good multi-purpose KDF, namely one to be used in different applications and with different sources of keying material, should follow the above two-module extract-then-expand approach. One main reason to separate these two modules is that they are functionally very different. The expand part is standard (can be implemented using regular PRFs) and applies to any situation. The extract module is generally more involved as it depends on the form of initial keying material and this may vary greatly, putting different requirements on the extract module. Depending on the specific case, this module may be instantiated with a purely combinatorial scheme such as universal hashing, or it may require a cryptographic hash function, or even an idealized "random oracle" (we discuss extensively these cases in the following sections). However, since we are interested in a single hash-based design that supports all these situations, we specify a single extract module that builds on our current knowledge about extractor properties of cryptographic hash functions. Specifically, we use recent results (e.g., [23, 17]) that show that HMAC's structure is well-suited as an "extractor mode of operation" for hash functions, especially in comparison to plain hash functions. See Section 8 for a summary of results and properties of HMAC as an extractor mode. In addition, another important advantage of using HMAC for extraction is that it can also be used as a PRF in the expand module of the KDF, hence simplifying the design and implementation of this KDF.

In contrast, constructions that do not follow the extract-then-expand approach often use ad-hoc designs that are hard to justify with formal analysis; these designs tend to "abuse" the hash functions, requiring them to behave in an "ideally random" way even when this is not strictly necessary for the KDF functionality. Our view is that *given the current (healthy) skepticism about the strength of our hash functions we must strive to design schemes that use the hash function as prudently as possible.*

Note. Most KDF schemes can be proven by abstracting the hash function as a random oracle (at least under prefix-free inputs). This idealized approach, however, is not the best "adviser" for KDF design: It does not differentiate schemes that are potentially vulnerable to a specific attack (such as off-line collisions) from those that are not, it does not care about where a key is positioned in a hash function (in a random oracle all key positions are equally strong), and leaves us wondering about the security of a scheme once weaknesses in the hash function are found. We strive to minimize random oracle assumptions as the basis of our design. We do accommodate it where needed but confine its use to the minimal possible, in particular avoiding requirements that are not known to be fulfilled by non-idealized schemes. On the other hand, we do address cases (e.g., when the constraints of an application are very stringent) in which the random oracle model is instructive for analyzing KDF schemes. See Section 6.

One important ingredient in the XtX scheme we propose, and not included in the above discussion, is that we take advantage, whenever available, of an additional parameter that we call *XTS* for *eXTractor Salt*. This is a random *but non-secret* string that we use as a "key" to Hash (or, in our actual design, to HMAC) in the extractor step of the XtX scheme. Doing so increases the provable strength of our KDF under weaker assumptions on the hash function. The power of extra randomness is well known from the research on randomness extractors in the complexity-theory literature. In particular, such random salt is essential for designing extractors that can extract randomness from *any* source with sufficient entropy (see Section 3), and it may help enforcing independence between sources of key material and the hash function. A practical question is which KDF applications may afford the *XTS* salt. After all, the whole idea of extractors is to generate

randomness, so if one already has such a random salt why not use it directly as a PRF key? The answer is that this randomness needs not be secret while in KDF applications we want the output of the extractor to be secret. Obtaining public randomness is much easier than producing secret bits (especially that in most cases the parameter *XTS* can be used repeatedly by the extractor). Indeed, many common applications can use an *XTS* value, including Diffie-Hellman protocols, imperfect RNGs [8], and more. See Sections 3 and 4 for an extensive discussion of these issues, and Sections 5 and 6 for cases in which salt is not available.

Bridging between theory and practice. We formulate and analyze a fully practical KDF scheme based on current theoretical research involving random extractors, hard-core functions, pseudorandomness, random oracles, and hash functions. The end result is a well-defined hash-based KDF scheme applicable to a wide variety of scenarios and which exercises much care in the way it utilizes cryptographic hash functions. While the considerations involving our design and its justification are highly technical, *we keep the presentation in this paper mostly informal stressing the conceptual aspects rather than the formal details of our analysis.* On the other hand, as an additional contribution, we present a formal definition of key derivation function (Appendix B) which reflects the comprehensive study of the KDF functionality developed in this work.

Related Work. In spite of being one of the most central and widely used cryptographic functionalities (in particular, specified in numerous standards – see [2]), there appears to be surprisingly little formal work on the specific subject of multi-purpose key derivation functions. The first work to analyze KDFs in the context of cryptographic hash functions and randomness extractors appears to be [23], which was followed-up, in the context of random oracles, by [17]. The former work laid the formal foundations for the KDF scheme presented here. This scheme, in turn, is based on the key derivation function designed by this author for the IKE protocols [34, 43] and which put forth the extract-then-expand paradigm in the context of KDFs. A variant of the expansion (PRF) stage of our KDF has also been adopted elsewhere, e.g. into TLS [20]. (We note, however, that TLS does not use our extract approach; for example, keys from a DH exchange are used directly as PRF keys without any extraction operation.) The extract-then-expand approach has subsequently been taken in [6] in the context of designing "system random number generators"; that work shares many elements with ours although the papers differ significantly in emphasis and scope (but note that our HMAC-based design can be applied in the RNG setting studied in [6] as well). Throughout our exposition we touch upon (and discuss) many other works on which we base our design rationale and analysis. In addition, we dedicate Section 9 to discuss some of the specific proposals for generic KDFs (including the influential NIST's design [57, 19] and the scheme from [2]) and to mention some source-specific KDFs as well as KDF-related applications (such as password-based KDFs and entropy-renewable random-number generators).

Organization. We organize the paper as follows. We first present, in Section 2, the extract-then-expand approach and its instantiation with HMAC, and dedicate the subsequent sections to expand on the rationale of this design (these latter sections can be skipped by those solely interested in our KDF specification). In Section 3 we recall the notion of extractor and the related randomness measure called min-entropy, which are central to our design and analysis. In Section 4 we provide some examples of the type of sources of (imperfect) keying material encountered in practice and for which multi-purpose KDFs are required to work. The rest of the work focuses on the analysis of our scheme and how it fits the theory and practice of KDFs in the many scenarios encountered in real applications. Specifically, Section 5 discusses the use of non-secret random keys (or salt) in the context of extraction and KDFs, and stresses its importance and practicality as well as some

of its limitations. Deterministic extractors are also discussed in this section. In Section 6 we study computational extractors at length, and analyze the use of hard-core functions and random oracles in the extraction process. Section 7 presents the rationale for the specific design of the expand module of our KDF scheme. In Section 8 we summarize known results about the HMAC scheme that serve as the foundation for the use of HMAC in our design. Section 9 presents a critique of prior KDF designs and compares them to our scheme and to the basic extract-then-expand approach; it also discusses additional related work. In Appendix A we recall formal definitions for many of the notions used throughout this paper, and in Appendix B we introduce formal definitions of key derivation functions and related notions.

2 The Proposed KDF Scheme

In this section we specify the *extract-then-expand key-derivation function* that we propose as well as the specific implementation using HMAC. The rest of the paper is dedicated to provide the rationale and theoretical foundations to this design, including the definition and discussion of the terms used in this section (though, a reading of the introduction should already provide sufficient background to understand the basic terminology and goals).

Components and Notation. A *key derivation function* receives as its main inputs a piece of partially-random *source key material SKM* (such as a Diffie-Hellman value, the output from some system sampling process, etc.) and a number L, and it outputs L bits of *keying material*, *KM*, of sufficient quality to be used as keys to cryptographic algorithms as required by the application calling the KDF (e.g, an AES key and an HMAC-SHA256 key). We build our key-derivation function, KDF, on the basis of a *randomness extractor* XTR and a pseudorandom function PRF. The extractor XTR is assumed to produce "close-to-random" (in the statistical or computational sense, see Section 3) outputs on inputs from the source *SKM*. The extractor XTR may be deterministic or keyed via a given "salt value" (i.e., a non-secret random value) that we denote by *XTS* (for *extractor salt*) which is provided, optionally, to KDF as an additional input (when this value is not provided it is ignored or set to a constant). The pseudorandom function PRF is assumed to work on variable-length inputs (but we note that in our scheme all calls to PRF with a given key are of the same length) and to produce fixed-length outputs. The key to PRF is denoted by *PRK* (pseudorandom key) and in our scheme it is the output from XTR. Thus, we are assuming that XTR produces outputs of the same length as the key to PRF.

More on notation: The first argument to a keyed function will always denote the key, e.g., when we write $\mathsf{PRF}(K, x)$, the value K is the key to PRF and x its input. We use the symbol $\|$ to denote concatenation, e.g., $\mathsf{PRF}(K, x \parallel y)$ means applying PRF with key K to the input formed by the concatenation of strings x and y. Finally, given two numbers N and n the symbol $N\!:\!n$ represents the value N written as n-bit integer.

Definition of PRF^* *(feedback mode PRF)*. Before presenting our KDF scheme we need to define an extension of PRF, denoted PRF^*, that outputs variable-length pseudorandom strings. That is, in addition to a regular PRF key *PRK* and input x, the function PRF^* receives an argument L that represents the number of output bits. We limit the output L from PRF^* to a maximum of $2^{32} \cdot k$ bits[5] (remember that k is the output length from PRF). The L bits are computed as follows, where

[5] This specific limitation is not essential; we chose it for simplicity and concreteness (and because in most KDF applications this number of bits would suffice). An application for which this number is insufficient can extend the counter value (or can use a PRF tree). Also, an application that will never use more than, say, $L = 256k$, can use an 8-bit counter. In no case, however, should the counter wrap around.

$t = \lceil L/k \rceil$ and $d = L \bmod k$,

$$\begin{aligned} \mathsf{PRF}^*(PRK, x, L) &= K(1) \parallel K(2) \parallel \ldots \parallel K(t){:}d \\ \textit{where} \quad K(1) &= \mathsf{PRF}(PRK,\ \ 0{:}k \parallel x \parallel 0{:}32), \\ K(i+1) &= \mathsf{PRF}(PRK,\ K(i) \parallel x \parallel i{:}32), \quad 1 \le i < t \end{aligned}$$

It is worth noting that PRF^* is well defined even when the input x is null. This may be the case in some of the applications using the KDF defined in this paper. A full rationale for the choices in the design of PRF^* (the 'expand" part of the KDF) is provided in Section 7.

Generic Extract-then-Expand KDF Definition. The function KDF receives four inputs: the source (or seed) key material *SKM*, the extractor salt *XTS* (which may be null), the number L of key bits to be produced by KDF, and a "context information" string *CTXinfo* (which may be null). The latter string should include key-related information that needs to be uniquely (and cryptographically bound) to the produced key material. It may include, for example, information about the application or protocol calling the KDF, session-specific information (session identifiers, nonces, time, etc.), algorithm identifiers, parties identities, etc. The computation of KDF proceeds in two steps; the L-bit output is denoted *KM* (for "key material"):

$$\begin{aligned} 1.\ \ PRK &= \mathsf{XTR}\,(XTS,\ SKM) \\ 2.\ \ KM &= \mathsf{PRF}^*(PRK,\ CTXinfo,\ L) \end{aligned}$$

An HMAC-based instantiation. For the sake of implementation in real applications (and for standardization) we propose to instantiate the above general scheme with HMAC serving as both the PRF and XTR functions (note that the length of HMAC output is the same as its key length and therefore the scheme is well defined). This results in a very simple scheme backed by properties of the HMAC construction (see Section 8). That is, if k is the output (and key) length of a hash function used with HMAC then the HMAC-based key derivation function HMAC-KDF is defined as

$$\mathsf{HMAC\text{-}KDF}(XTS,\ SKM,\ CTXinfo,\ L) \ = \ K(1) \parallel K(2) \parallel \ldots \parallel K(t){:}d$$

where (as before) $t = \lceil L/k \rceil$ and $d = L \bmod k$, and the values $K(i)$ are defined as follows:

$$\begin{aligned} PRK &= \mathsf{HMAC}(XTS,\ SKM) \\ K(1) &= \mathsf{HMAC}(PRK,\ 0{:}k \parallel CTXinfo \parallel 0{:}32), \\ K(i+1) &= \mathsf{HMAC}(PRK,\ K(i) \parallel CTXinfo \parallel i{:}32), \quad 1 \le i < t \end{aligned}$$

When the extractor salt *XTS* is not provided (i.e., the extraction is deterministic) we set $XTS = 0$.
Example: Let HMAC-SHA256 be used to implement KDF. The salt *XTS* will either be a provided 256-bit random (but not necessarily secret) value or, if not provided, *XTS* will be set to 0. If the required key material consists of one AES key (128 bits) and one HMAC-SHA1 key (160 bits), then we have $L = 288$, $k = 256$, $t = 2$, $d = 32$ (i.e., we will apply HMAC-SHA256 with key *PRK* twice to produce 512 bits but only 288 are output by truncating the second output from HMAC to its first 32 bits). Note that the values $K(i)$ do *not necessarily* correspond to individual keys but they are concatenated to produce as many key bits as required.

Notes 1. In the case in which the source key material *SKM* is already a random (or pseudorandom) string of the length of the PRF key, there is no need to extract from it further randomness and one

can simply use *SKM* as the key to PRF*. Also, when *SKM* is from a source or form that requires the use of the extraction step but the number of key-material bits required is no larger than the output *PRK* of the extractor, one could use directly this key as the output of the KDF. Yet, in this case one could still apply the PRF* part as an additional level of "smoothing" of the output of XTR. In particular, this can help against potential attacks on XTR since by applying PRF* we make sure that the raw output from XTR is never directly exposed.

2. Applications that use our HMAC scheme need to specify the contents of *CTXinfo*. As said this is useful (and sometimes essential) to bind the derived key material to application- and context-specific information, so the information in *CTXinfo* needs to be chosen judiciously. On the other hand, a general specification as ours, does not define any particular format or value for *CTXinfo* (contrast this to NIST's KDF [57] discussed in Section 9). We note that *CTXinfo* should be independent of *SKM*; more precisely, the entropy of the source conditioned on *CTXinfo* should be sufficiently high (see Appendix B). For example. in a DH application, *CTXinfo* should not include $SKM = g^{xy}$ but could include g^x, g^y.

3. One can consider the addition of a "context field" (similar to *CTXinfo*) also under the extract operation (e.g., concatenated to the *SKM* value). We recommend against this and hence do not specify such option; indeed, we show throughout this work many instances where concatenating information to *SKM* may reduce security. If an application still wishes to do so it will need to specify the formatting of this information and justify why it is necessary and secure to do so.[6]

4. Applications may consider inputting the length L as a parameter to PRF*. This is necessary only if an application may be calling the KDF twice with the same *SKM* and the same *CTXinfo* field but with different values L, and still expect the outputs to be computationally independent of each other. While possible, we do not see this as a usual scenario and hence prefer not to mandate it; specific applications can chose to do this either as an additional parameter to the KDF or as part of the *CTXinfo* field. (We do not have any important application in mind for which inputting the length would be a problem but we prefer to be as liberal as possible as to avoid usability problems in the future – see the discussion on the "over-specification" of NIST's KDF in Section 9.)

5. The HMAC-based scheme presented above is similar to the scheme designed by this author for IKE [34, 43]; specifically, in the case where IKE's PRF is instantiated with HMAC. One difference is that IKE omits the value "$0 : 32$" in the computation of $K(1)$. The proposal here treats the extract-then-expand paradigm in more general terms (for example, as pointed out, one may have different hash functions implementing the first and second stage), and it omits from the KDF those elements that are particular to the IKE setting. The fact that there is implementation experience with this scheme is an important consideration for its potential adoption into further standards. We also remark that a variant of our PRF* construct (but not the extract part) has also been adopted by the TLS protocol [20].

6. Our generic extract-then-expand specification can be instantiated with other-than-hmac functions. For example, one could instantiate XTR with HMAC but PRF with AES in CBC-MAC (or OMAC [42]). One particularly advantageous instantiation is to use HMAC-SHA512 as XTR and HMAC-SHA256 in PRF* (in which case the output from SHA-512 is truncated to 256 bits). This makes sense in two ways: First, the extraction part is where we need a stronger hash function due

[6] Some applications may find it convenient to reuse the KDF construct also as a "random oracle" applied to a *non-secret* input (e.g., to a message in a signature scheme or to public values in a key-exchange protocol). While a possible benefit of doing so is that HMAC provides a "better approximation" to a random oracle than a plain hash function [17], we note that such use fully departs from the functionality of a KDF, which is intended to process (semi) secret sources, and hence our analysis does not apply to it.

to the unconventional demand from the hash function in the extraction setting. Second, it is shown in [23] (see Section 8) that using HMAC with a truncated output as an extractor allows to prove security under considerably weaker assumptions on the underlying hash function. Yet another option for specific applications is to use a dedicated extractor that works well with the specific source (e.g., Diffie-Hellman values over some group family) together with an unrelated PRF (such as HMAC). We discuss source specific extractors in Section 5. Also, we note that our PRF* construct can be replaced with any variable-length output PRF, such as counter-mode PRF (but we have some good reasons, at least heuristically, to prefer PRF* – see Section 7), and in applications where *CTXinfo* is null, PRF* can be replaced with a pseudorandom generator seeded with *PRK*.

Note. Our generic KDF definition implicitly assumes that the output from XTR is no shorter than the key to PRF*. This is always the case in our HMAC scheme that uses the same function for both XTR and PRF*, but it could be an issue, for example, if one implements XTR with SHA-256 and PRF* with HMAC-512. In this case (which we do not recommend since we want to put the stronger primitive on the more demanding XTR part) one needs to define a larger output from XTR. If and how this can be done depends on the specific extractor function in use (e.g., if the extractor function is also designed as a good PRF, as in the HMAC case, one could use a scheme similar to PRF* to obtain more bits from XTR).

3 Introduction to Min-Entropy and Randomness Extractors

In this section we introduce two central notions used in this paper: min-entropy and randomness extractors. The presentation here is informal. Please refer to Appendix A for formal definitions.

The goal of the extract part of our KDF scheme is to transform the input source (seen as a probability distribution) into a close-to-uniform output. For this to be possible one needs the initial source to have "enough randomness" (even if not present uniformly). The classical notion of Shannon's entropy that measures the amount of "average randomness" in a source is not sufficient for general extraction. Instead what is needed is a "worst case" notion of entropy called min-entropy. The min-entropy of a probability distribution $\mathcal{S}$ is defined as the minimum integer m such that no element in the support of $\mathcal{S}$ is assigned a probability larger than 2^{-m}. A probability distribution $\mathcal{S}$, say on strings of length n, may have a significant min-entropy m, e.g., $m = n/2$, and still be very far from uniform. To see that the requirement of high min-entropy is a necessary condition for extraction, consider a probability distribution with a an element with high probability, say $1/2$, then the output distribution from an extractor (the KDF in our case) will also have such a high-probability element and hence far from uniform (also note that such source can have high Shannon entropy which shows the insufficiency of this entropy notion in our context).

Therefore, for producing close-to-uniform bits out of an input distribution $\mathcal{S}$ with min-entropy m, one needs an "extractor" function ext that outputs strings of some specified length m', and such that if s is a random variable distributed according to $\mathcal{S}$, then the (transformed) random variable $r = \mathsf{ext}(s)$ is "statistically close" to uniform (over the set of strings of length m'). The measure of *statistical distance* used in this case is standard and represents an upper bound on the total distance from uniform (in L_1 norm) of the distribution $\mathsf{ext}(\mathcal{S})$. (In particular, if the statistical distance between two distributions is ε, then no algorithm, even if computationally unbounded, can distinguish between the two with probability better than ε.) Clearly, m' cannot be larger than m and, in general, the closeness to uniform of the produced output will depend on the difference $m - m'$ (see below). Note that KDF functions need to have the property that when called on different samples from the source key material (e.g., different Diffie-Hellman values) the derived keys must be independent of each other. For this, one needs that the outputs of the extractor

function, computed on *independent samples* from the input distribution, also be independent.

The procedure ext as described above corresponds to what is called in the literature a *deterministic extractor*. While useful, deterministic extractors are of limited applicability since they can only work with *specific* distributions (no deterministic extractor can extract entropy from arbitrary distributions). A much more useful notion is that of generic *randomized extractors*, namely, a family of extractors (indexed by a set of keys K) that transforms **any** source distribution $\mathcal{S}$ of min-entropy at least m into a close-to-uniform output. More precisely, for each such source $\mathcal{S}$, if one choses a key κ at random from K, then with high probability the output distribution $\mathsf{ext}_\kappa(\mathcal{S})$ is close to uniform. It is not obvious that such families of extractors exist but they do and relatively simple constructions, for example based on universal hashing [15], are known and unconditionally proven to have the above extraction property [35, 41].[7] A crucial technical point is that in our application of extractors to KDFs, the random extractor key is public (i.e., known to the attacker) and hence we need the statistical distance from uniform be small even when this key is known (this key is what we call in Section 2 *XTS*, for "extractor salt").[8]

One may wonder where does the extractor key, or salt, come from in practical KDF settings. Moreover, if we already have a random string why do we need the extractor at all? The point is that, as said, the extractor salt can be public whereas the output it produces needs to be secret. Moreover, in most applications the extractor key κ can be used repeatedly with many (independent) samples from the same source, hence it can be chosen in an out-of-band or setup stage and be re-used later. For example, a random number generator that requires an extractor to "purify" its possibly imperfect output can simply have a random, non-secret, extractor key built in [8]. In other cases, such as key-exchange protocols extraction keys can be generated as part of the protocol (e.g., by using random nonces exchanged in the clear [34, 43]). We expand on this important issue in Sections 4 and 5. Applications that cannot produce or store an extractor salt need to resort to deterministic extractors with their intrinsic limitations (see Section 5).

For more background and constructions of extractors, in particular those built on the basis of combinatorial schemes such as universal hashing, see the surveys [55, 59]. One advantage of our extract-then-expand approach to KDFs is that it allows for the use of such combinatorial extractors. On the other hand, when designing a KDF that needs to serve also situations where traditional statistical extractors are insufficient (we will discuss those later), we need to resort to more ad-hoc construction based on cryptographic hash functions. Rather than outputting bits that are statistically close to uniform, the objective of these cryptographic constructions is to output *pseudorandom* bits, namely, bits that are *computationally indistinguishable* from uniform (i.e., a computationally bounded attacker cannot distinguish the extractor's output from a truly uniform string). Clearly, this is sufficient for cryptographic applications that already assume bounded attackers. We discuss these computational extractors in detail in Section 6.

One further advantage of using computational extractors is that one can relax the notion of min-entropy and only require computational min-entropy [35]. Roughly, a source distribution $\mathcal{S}$ has computational min-entropy m if $\mathcal{S}$ is *computationally indistinguishable* from another distribution with (regular) min-entropy m. The Diffie-Hellman case presented in the next section shows an

[7] For example, if the input distribution $\mathcal{S}$ is over strings of length n then a good extractor that outputs m' bits is built as follows. The key to the extractor is chosen as a random $m' \times n$ binary matrix M, and on input an n-bit element x from $\mathcal{S}$ the extractor outputs Mx. If $\mathcal{S}$ has m bits of min-entropy then the produced output is close to uniform up to a $2^{-\frac{m-m'}{2}}$ statistical difference. This statistical closeness holds even if the key M is known and if it is used repeatedly on multiple (independent!) samples from $\mathcal{S}$. More efficient variants, with shorter random keys, exist (e.g., replacing the above matrix with a random Toeplitz matrix), but the above statistical distance is optimal [59].

[8] Extractors with these properties are referred to as *strong extractors* in the literature but we drop the 'strong' since we only talk about strong extractors (except when dealing with deterministic ones).

important example where one starts with only computational min-entropy (and zero regular min-entropy) and yet is able to extract sufficient pseudorandom bits.

Finally, we point out that since KDFs are intended to output bits that remain secret to an attacker or observer, and given that such an attacker usually has some a-priori information about the source (e.g., it knows the public DH values g^x, g^y from which at the source key material g^{xy} is derived), then a formal treatment of extraction in our setting uses a notion of min-entropy (statistical or computational) that is *conditioned* on such a-priori information. See details in Appendix B. In the sequel all uses of the term entropy refer, implicitly, to (conditional) min-entropy.

A note about terminology. As one can see from the above discussion, there are different flavors of extractors. We will use "extractor" mainly to refer to strong randomized (non-deterministic) extractors where closeness to (or indistinguishability from) uniform is achieved even when the random key to the extractor is known. Also, we use the term extractor both in its regular statistical meaning, namely, one that outputs strings that are statistically closed to uniform, as well as in its "computational" variant where "close-to-uniform" means being computationally indistinguishable from uniform (i.e., pseudorandom). When a distinction is needed we will call the former "statistical extractors" and the latter "computational extractors". In some cases we will talk about non-randomized extractors in which case we will refer to these, explicitly, as deterministic extractors. See Appendix A for formal definitions.

4 Sources of Initial Keying Material

As pointed out before, the extract part of a KDF may depend on the type of initial keying material, the available entropy, the number of bits to output, etc. Here we discuss several examples of sources of initial keying material for KDFs as encountered in important practical applications. In particular, this serves to show the multi-functionality of KDFs and the challenge in creating a *single* KDF scheme that can deal with all these cases.

Physical RNG. Consider a physical random number generator RNG. Such generator often produces outputs that are not perfectly uniform but do have sufficient min-entropy (a simple example is a generator that outputs 0 with fixed probability $p < 1/2$ but not less than, say, $1/4$). To produce a truly (close to) uniform output one uses an extractor on a (sufficiently long) sequence generated by the RNG.[9] We note that in this application one can salt (or key) the extractor with a non-secret random value that is chosen once (from some off-line randomness-producing process) and fixed as an RNG parameter. See [8] for more on the use of extractors in the context of physical RNGs.

Ad-hoc statistical sampling. Consider the typical practice of sampling system events, clocks, user keystrokes, etc., to produce an initial source of randomness for seeding a PRG or other randomness applications (especially when a source of physical randomness is not present). This application is based on the assumption that the sampled source contains sufficient variability such that it is partially unpredictable by an external attacker. In other words, the source is assumed to behave as a probability distribution with sufficient min-entropy conditioned on information held by the attacker (which can see part of the events in a system or even partially influence them). Here again, applying an extractor on a (long enough) sequence of such samples produces a close-to-uniform

[9] In this example if one samples n bits from RNG one gets that the most probably string is the all-1 string which has probability at most $(3/4)^n$, i.e. the source has min-entropy $-\log_2(3/4)n \approx 0.4n$. For example, if $n = 1000$ one can use an extractor as in footnote 7 and output 200 bits with a statistical deviation from uniform of at most 2^{-100}.

output. As in the RNG example, one can set a random extractor key at some initial stage and use this salt repeatedly with multiple samples. See [6] for a detailed treatment of this scenario.

KEY EXCHANGE AND DIFFIE-HELLMAN PROTOCOLS. Diffie-Hellman protocols output random group elements of the form g^{xy} represented by strings of some length n. Clearly, these values are not uniformly distributed over $\{0,1\}^n$ and therefore cannot be used directly as cryptographic keys. In turn, keys are usually derived from g^{xy} via a KDF under the assumption that g^{xy} contains "sufficient randomness" that an attacker cannot predict. More precisely, when modeled appropriately, the values g^{xy} and their probability distribution, as implied by the DH protocol, become a particular case of an imperfect source of randomness. One essential difference with previous examples is that in the Diffie-Hellman setting the notion of min-entropy is *computational* (note that an attacker that knows g^x, g^y has full information to compute g^{xy}, and therefore the statistical entropy of g^{xy} given g^x, g^y is zero.) Fortunately, the computational min-entropy of a source is sufficient for deriving *pseudorandom* bits via regular randomness extractors as considered here (see Section 3), and hence by incorporating a randomness extractor into the KDF design (as we do) one gets assurance of the pseudorandomness of the output from the KDF.

While this generation of pseudorandomness via statistical extraction from DH values applies to many protocols and DH groups, there are important cases where this does not hold. For example, not all Diffie-Hellman groups offer "sufficient" computational min-entropy for extraction purposes. In particular, this can only be guaranteed (see [29]) in groups where the Decisional Diffie-Hellman (DDH) assumption holds in a sufficiently large subgroup. This is *not* the case, for example, in DH groups based on pairings where the DDH assumption does not hold. In addition, in order to apply a randomness extractor in the setting of DH key-exchange protocols, it is necessary that the value g^{xy} computed by the parties be indistinguishable (by the attacker) from a random group element. This is the case for some DH protocols but not for all. For example, in a protocol such as MQV (or its provable variant HMQV) [50, 49] one can argue that the shared group element computed by the protocol is hard to fully guess by an attacker but not necessarily hard to distinguish from random group elements. Finally, even when one can use the DDH assumption one may not have enough min-entropy to extract the required number of bits (e.g., when requiring the extraction of 160 bits of key from a DH value over a 192-bit prime order elliptic curve). In all these cases, one cannot use regular generic extractors but rather needs to resort to stronger properties or assumptions (including, in some cases, modeling a hash function as a "random oracle") or use group-specific extractors. We expand on these cases in Section 6.

One additional consideration when using randomness extractors in the setting of DH protocols is that these extractors work well only on *independent samples*. That is, if one uses the extractor to generate a key out of a value g^{xy} and later to generate another key from a second value $g^{x'y'}$, the two keys will be pseudorandom and independent *provided that the two DH values are independent*. In particular, an attacker should not be able to force dependencies between DH values exchanged between honest parties. This is the case in a (well designed) DH protocol with explicit authentication. In protocols with implicit authentication or those that authenticate the protocol via the output of a KDF, one may not obtain the independence guarantee. This is yet another case where stronger extractors are required (see more in Section 6.2).

We end by commenting that one good property for the use of extractors in Diffie-Hellman applications is that the parties to the protocol can provide the necessary "salt" to key the extractor. Indeed, such value can be obtained from random nonces chosen by the parties and exchanged authenticated but *in the clear*. This is the approach we took in the design of IKE [34, 43, 48].

5 Randomized vs. Deterministic Extractors

As we have already stressed in previous sections *generic* extractors, i.e., those working over arbitrary high min-entropy sources, must be randomized via a random, but not necessarily secret, key (or "salt"). Indeed, for every deterministic extractor ext one can show a source, or probability distribution, where the output of ext is not close-to-uniform. For example, consider an extractor ext acting on n-bit inputs and with m'-bit outputs, and define I_0 (resp. I_1) as the set of n-bit inputs mapped to strings whose least significant bit is 0 (resp. 1). Assume wlog $|I_0| \geq |I_1|$, and define the uniform distribution on I_0. If ext is a generic extractor it should output a close-to-uniform output on such distribution (whose min-entropy is $n-1$), yet its output has the least significant bit fixed to 0 and hence is far (and easily distinguishable) from uniform. We note that this limitation of deterministic extractors holds also with respect to computational ones. This is even the case, if one uses a truly random function as the deterministic extractor. In particular, as we show in Section 6, even an idealized random-oracle instantiation of extractors may also require a random key to overcome certain inherent restrictions.

In general, randomization provides a way to *enforce independence between the source distribution and the extractor itself*, and between different uses of the same extractor scheme by an application. The importance of such independence is well demonstrated by the following "real-life" example. The STS protocol [22] is a well-known Diffie-Hellman protocol authenticated via digital signatures and symmetric encryption. A variant of it, intended to get rid of the symmetric encryption for authentication, was proposed in [52] (and used in Photuris, a predecessor of today's IKE protocol). This variant authenticates a DH key g^{xy} by signing it. However, since a signature (e.g., RSA) may reveal the signed value, what is signed is $H(g^{xy})$ rather than g^{xy} itself where H is hash function such as SHA-1. In this case the attacker may learn $H(g^{xy})$ via the signature but not g^{xy}, and if one thinks of H as a random oracle then nothing useful is learned about g^{xy} from seeing $H(g^{xy})$. Now, suppose that the above protocol derives keys via a traditional KDF, namely, keys are computed as $H(g^{xy} \parallel v)$ for some public values v. In this case, if we follow the H = SHA-1 example and assume g^{xy} is of length 1024, we get that computing $H(g^{xy} \parallel v)$ is the same as computing H on v with the IV of H set to $H(g^{xy})$ (for simplicity assume that the computation of H on g^{xy} does not use length appending). However, this means that the attacker, that learns $H(g^{xy})$ from the signature, also learns all the derived keys.

The failure of this protocol is *not* due to a specific weakness of H; the same problem happens if we replace the compression function of SHA-1 with a fully random function. Instead, the problem stems from the fact that two different functionalities in the protocol, the hash function applied to g^{xy} and the KDF, both use the same (deterministic) H. Keying H in its use by the KDF with a random non-secret key (or salt), would enforce independence between the two functionalities. We will see the role of independence, in a more technical sense, in the results presented in Section 6, where we also discuss the necessity of randomness in cases where one uses hard-core functions instead of extractors. We also comment that using random keys with extractors has additional advantages, at least at the heuristic level, such as avoiding potentially weak functions in a hash family and, importantly, preventing attacks based on pre-processing that targets a specific function. In all cases, caution must be exercised to keep the salt independent from the source keying material.

A valid question is whether typical KDF applications can use randomization (or salt). As we argued in Section 3 and exemplified in Section 4 this is indeed the case in many important applications (including RNGs, key exchange, etc.). Yet, when such salt is not available one has to resort to *deterministic extractors*. These require source-specific assumptions as well as independence assumptions on how an application uses the extractor. The above example regarding the

STS variant from [52] should, however, call for much caution in making this form of assumptions especially when building KDFs with "generic tools" such as cryptographic hash functions. This is the case even if one models the hash function as a random oracle or under specific assumptions such as the "hash Diffie-Hellman (HDH) assumption" [1] where one assumes that a specific hash function, say SHA-1, acts as a good extractor from a specific group (or family of groups). Another option when randomization is not possible is to resort to extractors that work in particular cases and hence of restricted applicability. Examples of such "domain-specific" extractors include the hard-core schemes for RSA [3, 27] and discrete-log based functions [38, 58], and the recent elegant extraction functions specific to some Diffie-Hellman groups in [16, 28].

An interesting direction for further investigation is the use of deterministic extraction when one is provided with two independent randomness sources; see [7] (interestingly, the extractor from [16] may be seen as a specific case of the above approach).

On secret salt. Some applications may have available a *secret* key K and still need to extract another key from a source such as a Diffie-Hellman value. This is the case in IKE (public-key encryption and preshared modes) [34, 43] and TLS's preshared key mode [25]. For this, it is best to use the secret K as a key to a PRF applied to *SKM*. Our KDF is particularly well-suited for this case since by setting *XTS* to K one obtains exactly the above PRF effect (historically, this dual use of salt has been one of the motivations for our design of the IKE's KDF[10]).

6 Computational Extractors and Random Oracles

We have considered two main notions of extractors. The traditional, statistical extractors, that guarantee an output that is statistically close to uniform, and the computational ones that output a distribution that is not necessarily statistically close but rather "computationally close" to uniform, namely pseudorandom. Clearly, the former have the advantage of providing the extraction capability in an unconditional (information-theoretic) form and require no computational assumptions or computational bounds on the observer. Very importantly, they are *generic*, namely, work on any source of sufficient min-entropy. There are several reasons, however, to consider computational extractors in the cryptographic context. First, the cryptographic applications themselves most often assume computationally-bounded attackers. This is even more so when the source itself has computational, rather than statistical, entropy (e.g., Diffie-Hellman protocols), in which case, even if one applies a statistical extractor the output is still only pseudorandom. Second, computational extractors can be built, under suitable assumptions (as we do in this paper), on cryptographic hash functions. The latter are more practical (i.e., more efficient and use less public randomness) and are more available than statistical extractors (indeed, such hash functions are already part of most cryptographic applications and are part of all cryptographic libraries). Finally, and very significantly, computational extractors may help overcoming some of the inherent limitations of statistical extractors as we discuss next.

Statistical extractors require a significant gap between the min-entropy m of the source and the required number m' of extracted bits (no statistical extractor can achieve a statistical distance, on arbitrary sources, better than $2^{-\frac{m-m'}{2}}$ [59]). That is, one can use statistical extractors only when the min-entropy of the source is significantly higher than the length of output. These conditions may be met by some applications, e.g., when sampling a physical random number generator or when gathering entropy from sources such as system events and human typing where higher min-entropy

[10] Contrast this with the ad hoc "key concatenation" solution for building the premaster key in [25].

can be achieved by repeated sampling. However, in other cases, very notably when extracting randomness from computational schemes such as the Diffie-Hellman key exchange discussed earlier, the available gap may not be sufficient (e.g., when extracting 160 bits from a DH over a 192-bit group). In these cases, statistical extractors are out of the question while computational ones, as those based on cryptographic hash functions, may plausibly provide better extraction quality.

Another parameter where computational extractors may provide more practical alternatives to statistical ones is key (or salt) size. Indeed, statistical extractors, such as those based on universal hashing, require quite long keys (e.g., a random Toeplitz matrix as in footnote 7), certainly much longer than those based on cryptographic hash functions.

Computational extractors also fare very well when considering specific sources (e.g., DH over specific families of groups), in which case computational *deterministic* extractors may exist without an equivalent statistical counterpart (see Section 5). While these deterministic extractors are, by nature, of limited application, we have structured our KDF schemes such that an application can take advantage of these extractors by using them to implement the extraction phase of the KDF. Finally, when more realistic assumptions do not provide for a general or strong enough extractor, one may resort to random oracles which provide another form of computational extraction. We expand on this subject on Section 6.2.

As we see, the most practical benefits of computational extractors are obtained when one implements them with cryptographic hash functions. Especially in light of recent cryptanalytical advances, one has to be careful to design such extractors on as weak as possible assumptions on the underlying hash function. We will see in Section 8 how this is done for our HMAC-based scheme. In some cases, however, the application itself assumes much stronger properties from the extractor, such as outputting (almost) all the min-entropy of a source, working on insufficient min-entropy, assuming independence of the extractor output on non-independent samples of the source distribution, and more. In the rest of this section we will discuss how to deal with these situations. We first consider extraction of pseudorandom bits from cryptographic hardness (or unpredictability) without assuming additional (computational) min-entropy and then follow with a study of the use of the random oracle model in the setting of extraction. Also here, we try to confine the reliance on random oracles and related "ideal creatures" to the minimum we know.

6.1 Extraction from Computational Hardness

Consider the problem of extracting pseudorandom bits from a Diffie-Hellman value shared between two parties. As discussed in Section 4, extraction in this case is possible if the DH group satisfies the Decisional Diffie-Hellman (DDH) assumption with sufficiently high computational min-entropy (relative to the number of required pseudorandom bits) [29]. In this case, one can extract the computational min-entropy present in the value g^{xy} (conditioned on g^x, g^y) by applying to this value a statistical or computational extractor. However, when working over groups where the DDH assumption does not hold (e.g., bilinear groups), generic extractors (even computational ones) provide no guarantee that the output will be close to (or indistinguishable from) uniform; indeed, in this case there is no guaranteed computational min-entropy in g^{xy}.

Fortunately, one may still extract some pseudorandom bits from the mere fact that it is hard to compute g^{xy} from the pair g^x, g^y (i.e., the Computational Diffie-Hellman, CDH, assumption). For this one resorts to the theory of hard-core functions [13, 32]. Roughly, given a *one-way function* f over $\{0,1\}^n$, we say that h is a hard core function of f if for random $x \in \{0,1\}^n$, the string $h(x)$ is pseudorandom (i.e., computationally indistinguishable from uniform) even when $f(x)$ is given. It is well-known that every one-way function has a hard-core function, for example the Goldreich-Levin

GL function [31]. This is a *generic* hard-core, namely, one that works with *any* one-way function. The GL function is keyed with a key consisting of a random (Toeplitz) $n \times m'$ matrix M (for some value m' – see below). On input an n-bit value x, GL outputs the m'-bit value Mx. This value is guaranteed to be pseudorandom even when the value $f(x)$ and the matrix M are known. Note that the GL function is *randomized* (or keyed); this is necessary since as we argue below *no deterministic* hard-core function can be generic, i.e., work with all one-way functions.

So, we know how to extract pseudorandom bits from one-way functions. Note, however, that strictly speaking the DH problem (from which we want to extract bits) cannot be stated in terms of traditional one-way functions, e.g., there is no easy way, in general, to test for a given triple (g^x, g^y, g^z) whether $g^z = g^{xy}$ (except in groups where the DDH is easy). Yet, with some more work, GL can be applied in this case too as a hard core function (for the details see [61]), thus allowing us to extract pseudorandom bits from DH values solely based on the CDH assumption. The bad news is that the number of such bits cannot be too large as we see next.

In the above definition of the GL function we did not specify the parameter m', namely, the number of output bits. In [31] it is shown that the GL function may output $m' = O(\log n)$ pseudorandom bits. This is often too little for many applications, including DH. Unfortunately, this cannot be improved by *generic* hard-core schemes: the results from [30] imply that the maximal number of bits that can be extracted from a one-way function f by any generic hard-core function, i.e., one that treats f as a black box, is linear in t, where 2^t is the time (or circuit size) of the best algorithm to invert f in a constant fraction of inputs. We will refer to t as the hardness of f. (See [39] for a treatment of computational hardness in terms of entropy.)

In order to evaluate the practicality of these parameters when extracting bits under the CDH assumption, we consider the optimized security analysis of GL in [36]. We get that if we are interested to use GL in order to output m' pseudorandom bits that withstand distinguishers running time $2^{t'}$ (for simplicity, we fix some constant distinguishing advantage, say 1%), then we need to start with a one-way function with t bits of hardness where t must be at least $t' + 2m'$. If we assume that solving the CDH problem on a group of order q requires $\sqrt{q} = 2^{|q|/2}$ operations, we have that for extracting m' pseudorandom bits that are $2^{t'}$-indistinguishable, we need to work with a group of order q such that $|q|/2 \geq t' + 2m'$. Setting, for example, both t' and m' to the value 128 we need q to be of length at least 784. Such size of q may be fine in some cases, e.g., with DH groups using strong primes (as in IKE [34]), but are usually much smaller in DSA or elliptic-curve type groups. Thus, in many practical instances, the provable GL cannot be used.

Another problematic parameter in GL is the key size. For example, in order to output 128 pseudorandom bits from a DH value over Z_p^*, with a 1024-bit p, one would need 1152 bits of randomness for the Toeplitz matrix. This is even worse using the results from [36] which provide a better rate of pseudorandom output but use fully random matrices as the key, i.e. 128Kb in the above example. Ideally, one would like to get rid of these keys at all. However, as pointed out in [32], random keys are essential for generic hard-core functions (i.e., those working with all one-way functions). The argument is simple: if f is a one-way function and h is a deterministic hard-core function then f' defined as $f'(x) = (f(x), h(x))$ is also one-way but, obviously, h is not a hard-core function for it. This example is worth keeping in mind as just another indication of the importance of using randomized (i.e., keyed) extractors as discussed in Section 5. It is also interesting to note the similarity between this theoretical argument and the very practical example from Section 5 based on the STS/Photuris protocol.

Due to the above limitations of (provable) generic hard-core functions (some essential and some due to the state of our knowledge), one often resorts to specific extractors for specific applications. For example, more efficient extractors exist for the RSA and discrete-log functions [38, 37, 27] than

the generic ones discussed above. Yet, even these extractors are not always practical enough. In these cases, one may build on special assumptions on a given family of hash functions. In the context of DH applications, for example, this gives rise to the hashed Diffie-Hellman (HDH) assumption [1], that basically assumes that a specific hash function (e.g. SHA-1), or family of hash functions, is a good extractor from DH values over specific groups. There is little hope that one could prove anything like this for regular cryptographic hash functions such as SHA. So even if the assumption is well defined for a specific hash function and a specific group (or collection of groups), validating the assumption for standard hash functions is quite hopeless. This is even worse when requiring that a family of hash functions behaves as a *generic* extractor (i.e., from arbitrary sources) as needed in a multi-purpose KDF as designed in this paper. In these cases, one "sanity check" is to see if the required properties hold at least in the idealized case that one replaces the hash function with a truly random function. In other words, this is one place where the random oracle model is instructive for analyzing KDF schemes. We expand on this and on other applications of random oracles in the KDF context in the following subsection.

6.2 Random Oracles as Extractors

As we concluded above, the random-oracle model may be invoked in the context of KDF design as a "sanity check" for the use of hash functions as (specific or) generic extractors; e.g., as a way to model the hashed DH assumption without entering the specific details of a group or a hash function. As with any application of the random-oracle model [26, 12] one has to be careful about the way one interprets results obtained in this model. The main usefulness of this model is in uncovering possible *structural weaknesses* in a cryptographic design but it ultimately says little about the actual security of a specific real-world design where one uses a hash function instead of the random oracle. Thus, while we warn against the widespread practice of building KDF schemes *solely* on the basis of abstracting hash functions as random functions, we do not ignore the potential benefits of this model as a verification tool, especially when we do not have a better way to validate a design or its application in specific scenarios. Moreover, some KDF applications do assume the KDF to behave as a random oracle (see below) and hence we also study to what extent our hash-based KDF design emulates a random oracle when instantiated with a primitive such as a random compression function. (We note that building on a random compression function does not automatically mean the KDF, or even the underlying hash function, behaves as a random oracle).

Next, we provide examples of extraction properties of random oracles that are required by some significant applications, especially in the context of key-exchange and Diffie-Hellman protocols, and that are not guaranteed by regular, even computational, extractors. We also discuss some results that measure the quality of random oracles as extractors. In Section 8 we will cover specific results showing that our HMAC-based KDF design emulates a random oracle when modeling its underlying compression function as a random function.

6.2.1 Extraction applications served by the random oracle model

As said, there are some important situations where one has too little entropy or to little gap (i.e., the number of required output bits is close to the min-entropy of the source) to apply regular extractors, or even computational ones. In many of these cases, the "last resort" for analysis and validation of a design is to model the extractor component as a random oracle. In the next subsection we study the extraction properties of random oracles, but first we present a list of common cases where regular extractors do not suffice (these examples are mainly from the Diffie-Hellman application

but they show up in other scenarios as well).

- A DH group where the Decisional Diffie-Hellman (DDH) is assumed but the amount of computational min-entropy in it [29] is close to the number of required bits (e.g., extracting 160 bits from a 192-bit group, or even from a 160-bit group).
- A group where the DDH assumption is not assumed or is known to fail (such as in bilinear groups). All one has is the Computational Diffie-Hellman (CDH) assumption and hence only extraction via hard core functions is possible, yet the number of required bits is larger than what can be obtained via a generic hard core such as Goldreich-Levin as discussed earlier.
- Cases where the application does not guarantee *independence of samples*. This is the case, for example, in DH protocols where a party is allowed to re-use a DH value, or use non-independent values such as using g^x in one session and g^{x+1} in another (see discussion in [47]). It is also the case for protocols in which the attacker may be able to generate valid DH values that are dependent on previous values (this is common in "implicitly authenticated protocols", e.g., [50, 49], where DH values are input into the KDF before explicit authentication).
- KDF (and other extraction) applications where a protocol explicitly models the KDF as a random oracle due to additional requirements from the output of this function (the previous item is such an example from the area of key-exchange protocols).
- Cases where regular, provable extractors would work but they are not practical enough, such as for reasons of performance, amount of required randomness (salt), or simple software/hardware availability.

We stress that even in these cases, where a "super hash function" is needed to fulfill the stringent constraints of the application, one should still strive to minimize the extent of idealized assumptions. For example, hash functions that are vulnerable to extension attacks should never be considered as random oracles; at best one should only model the compression function as random. See the HMAC-related results discussed in Section 8.

6.2.2 Properties of random oracles as extractors

Here we mention some results showing that random oracles have good properties as extractors. This is certainly not surprising, yet, as we'll see, these results hold under some restrictions that may not be apparent or intuitive at first. Also, note that the results in the random oracle model are only computational since they require to bound the number of queries to the random oracle allowed to the attacker. Moreover, some of the intrinsic limitations of extractors and hard-core functions mentioned earlier cannot be avoided even if one implements the extractor using a fully random (fixed) function. This is the case, for example, for deterministic extractors: even a random function (namely, one chosen at random but then fixed) cannot serve as an unkeyed generic extractor or hard core. This shows, in particular, the pitfalls (and potential vulnerabilities) of using a fixed deterministic hash function, say SHA-256, to implement a KDF.

The first result that we recall here is the following lemma from [23] that shows *(under careful restrictions!)* that a random oracle can output all the entropy of the source.

Lemma 1 [23] *Let H be a random function with domain $\mathcal{D}$ and range $\{0,1\}^k$, and $\mathcal{S}$ be a probability distribution over $\mathcal{D}$ with min-entropy m and independent from H. Then the distribution $H(\mathcal{S})$ is $q2^{-m}$-indistinguishable from k random bits by any attacker that queries H on at most q points.*

Note that the independence condition in the above lemma is essential. For example, the uniform distribution on the set of points mapped by H into the all-zero strings has a large min-entropy (assuming $|\mathcal{D}| \gg 2^k$), but the output of H on this distribution is trivially (i.e., with $q = 0$) distinguishable from random. One could be tempted to make the assumption, in practice, that actual distributions (or sources) of interest will be independent from the hash function used as random oracle. This, however, is unrealistic in general. Consider a DH application that uses a hash function H, say SHA-256, to hash DH keys of the form g^{xy} (i.e., it uses H as its KDF), but also uses SHA-256 as the basis for the pseudorandom generator that generates the DH exponents x and y. In this case there is no independence between the "random oracle" H and the source, $\mathcal{S}$, of DH values g^{xy}.

The way to enforce independence between H and $\mathcal{S}$ is to model H as a *family* of random functions indexed by a key rather than as a single deterministic function. Then, if the KDF application chooses a function from the family H by choosing a random key (this key needs to be random but not secret, as our salt *XTS* – see Section 5), and the key is independent from the source $\mathcal{S}$, then we satisfy the independence condition from the above Lemma. Applications that cannot afford the use of salt must resort to the explicit assumption that H and $\mathcal{S}$ are independent.[11]

As discussed earlier in this section, we are sometimes faced with the need to model a hard-core function as a random oracle. The following lemma shows, in a quantitative way, that a random oracle is as close to a generic hard-core function as one can hope. Note the independence requirement in the lemma and the subsequent remark.

Lemma 2 *Assume f is a $(\tau, 2^t, \varepsilon)$-OWF, namely f can be computed in time τ but cannot be inverted (over random inputs) with probability better than ε in time less than 2^t (in case of non-permutation, inversion means finding any pre-image). If H is an independent random function with k bits of output then the probability distribution $(f(x); H(x))$ over uniformly distributed x is (q, T, ε)-indistinguishable from $(f(x); U_k)$ (that is, no distinguisher running time T and making q queries to H has distinguishing advantage better than ε) where $T = 2^t - q\tau$.*

The proof follows standard arguments and is presented in Appendix C for completeness and as background to the application to Diffie-Hellman described below.

Remark: Lemma 2 assumes that f is independent of H, namely, computing f does not involve queries to H. However, this assumption can be relaxed and f be allowed to depend on H as long as computing $f(x)$ (as well as sampling x) does not involve computing H on x. Note that by the example presented in Section 6.1 (and its real-life instantiation in the STS variant from [52]), this minimal independence requirement is essential. This example also shows that generic *deterministic* hard-core functions do not exist even if implemented via a random function.

Application to Diffie-Hellman. Consider the case of an attacker that is given a triple $(g^x, g^y, H(g^{xy}))$ and needs to distinguish $H(g^{xy})$ from random. We have a situation similar to the above since, intuitively, the attacker cannot distinguish without computing g^{xy} from g^x and g^y, and the latter is hard assuming the computational Diffie-Hellman (CDH) assumption. So we would like to apply the above lemma here where inverting f means finding g^{xy} given g^x, g^y. One difficulty, however, is that this function f cannot be computed efficiently in the other direction. Specifically,

[11] The independence assumption can be relaxed as follows. We let the attacker $\mathcal{A}$ choose any source $\mathcal{S}$ with min-entropy m but with the following restrictions: $\mathcal{A}$ is allowed to query H on q_1 values after which it sets the source distribution $\mathcal{S}$, possibly dependent on these queries. Next, x is chosen at random from $\mathcal{S}$ and $\mathcal{A}$ is given $H(x)$ or k random bits. $\mathcal{A}$ tries to decide which is the case by performing q_2 additional queries from H. The above lemma now holds with $q = q_1 + q_2$.

for the proof of the lemma to hold in this case we need to have an algorithm that given a triple (g^x, g^y, v) decides whether $v = g^{xy}$. This is possible in groups where the decisional Diffie-Hellman (DDH) assumption is easy (such as bilinear groups) but is hard otherwise. Yet, a variant of the reduction in the proof of the lemma (Appendix C) works here too. Instead of having I output g^{xy} we let I output the list of all q queries to H. We get that with probability ε (the assumed distinguishing advantage) the list of q elements contains the correct element g^{xy}. Now, following [61], we can use the self-reducibility property of DH to build from such algorithm I a procedure that given g^x, g^y computes g^{xy} with good probability.

7 The Key-Expansion Module

The expand functionality of a KDF is equivalent to a pseudorandom generator (PRG) (i.e., producing, out of a random seed, a sequence of bits that cannot be distinguished from the uniform distribution by feasible computation). Thus, one can use any such generator for this part including any strong stream cipher. Yet, we focus on PRG implementations using pseudorandom functions (PRF) since these are more closely aligned with the functions we use for extraction, certainly in the specific case where we want to use HMAC for both modules. More importantly, we use the PRF key expansion to also bind the key material to context information (*CTXinfo*) something that a plain PRG cannot achieve.

The most basic PRFs in practice, such as a block cipher or a keyed compression function, are defined with fixed-length input and fixed-length output. To transform these PRFs so they can accept arbitrary-length inputs one resorts to common modes of operations such as CBC-MAC or OMAC for block ciphers and Merkle-Damgard for compression functions [53, 18].

For our KDF application, we also need the PRF to support variable-length outputs, hence we need to define a way to iterate a (arbitrary-input) PRF to produce outputs of any required length. The simplest and most common solution for this is to use "counter mode", namely, computing the PRF on successive values of a counter. Since PRFs are defined and designed to produce independent pseudorandom outputs even from dependent inputs, this mode is perfectly fine with strong PRFs. In our design we take a more conservative approach[12], via "feedback mode", intended to deal with potential weaknesses of a PRF as explained next.

The feedback mode we recommend is specified in Section 2: it transforms any fixed-length output (and arbitrary-length input) pseudorandom function PRF into a variable-length output PRF*. Its difference with counter mode is that instead of computing the function PRF on successive counter values, each iteration is applied to the result of previous iteration. Our rationale is mostly heuristic (a formal analysis can be carried via the notion of *weak pseudorandom functions* [54])

Clearly, in feedback mode the attacker has less knowledge on the inputs to the PRF, except for the first input which is fully known (as in counter mode). Formally, this may not be an advantage since the output of PRF* should be unpredictable even when the attacker has knowledge of all bits except one; however, in practical terms PRF* is likely to present a significantly harder cryptanalytical problem. This includes a smaller amount of known plaintext than counter mode (or even chosen plaintext if the attacker can force some counter values). Another good example of the advantage of highly variable inputs (even if they were known to the attacker) is that successive values of a counter differ in very few bits which creates relationships such as low Hamming differentials between inputs, a potential aid for cryptanalysis.

[12]The strength of the PRF is important since its cryptanalysis could compromise past communications and information protected under keys derived from using the PRF.

Two more elements in our design are as follows. We concatenate a counter value to each input to PRF. This is intended to guarantee different inputs to the PRF against potential cycle shortcuts [60]. The other is the use of an all-zero input in the first iteration to occupy the feedback field used in later applications of PRF. This measures makes all inputs to the PRF to be of the same length. This is not strictly needed with HMAC (which, for example, is not vulnerable to extension attacks) but it may be needed with other PRF modes (such as CBC-MAC), and it is often a good basis for better analytical results. We note that this all-zero value is one difference between the proposal in this paper and the PRF^* version used in IKE.

8 Why HMAC?

Here we summarize some of the results from the literature indicating the suitability of HMAC for the key derivation functionality, and for the specific way we use it. These results show that the structure of HMAC works well in achieving the basic functionalities that we need here, such as PRF, extraction, and random-oracle domain extension. Also, HMAC supports working with a secret key, a random non-secret key (salt), or deterministically (i.e., with a fixed value key).

HMAC's use as a PRF is very widespread; the theoretical basis for this use is provided in [10, 11]. Although the security of HMAC as PRF degrades quadratically with the number of queries, such attack would require the computation of the PRF (by the owner of the key) over inputs totalizing $2^{k/2}$ blocks. This is even less of a concern in typical KDF applications where the number of applications of the PRF is relatively small.

The results regarding the security of HMAC as extractor are more complex. We provide an informal summary here. Refer to the cited works for formal details and quantification of security.

Notation. We use H to denote a Merkle-Damgard hash function and h the underlying compression function. We also consider these as keyed families, where h_κ and H_κ denote, respectively, the compression function and the Merkle-Damgard hash with their respective IVs set to κ; the key and output lengths of these functions is denoted by k. We will abuse notation and talk about "the family h_κ" instead of the more correct $\{h_\kappa\}_{\kappa\in\{0,1\}^k}$; same for H_κ. When we say that "the family h_κ is random", we mean that each of the functions h_κ is chosen at random (with the corresponding input/output lengths). When we talk about HMAC (or NMAC), we assume underlying functions h and H (or their keyed versions).

As extension attacks show, the hash function H does not behave as a random function even if the compression function h is completely random. This limitation holds even if one considers H as a keyed family and even if the attacker is computationally bounded. As for extraction, the function H cannot be a generic extractor since no *single* function can be such an extractor (see Section 5). What is more surprising is that even if we consider the *keyed family h_κ, where h_κ are random functions*, the resultant keyed family H_κ is *not* a generic extractor either: [23] shows that the output of such H_κ on any distribution for which the last block of input is fixed is necessarily statistically far from uniform. Furthermore, the output of H_κ is close-to-uniform if and only if the last block has high entropy. Interestingly, this result serves as another (non-intuitive) indication that for extracting a key from input *SKM* it is best to hash *SKM* by itself, as in our scheme, rather than with appended information as in traditional KDFs.

Since even a random family h_κ is insufficient to guarantee generic statistical extraction, the next step is to consider whether H_κ may serve as a *computational extractor*. In this case, instead of modeling h as a random function we model it as a *random oracle (RO)*, namely, a random function which the attacker can only query on a limited number q of queries. In this case, [17] shows (using

the framework of indifferentiability from [51]) that if the function h is a random oracle, then so is HMAC on *any set of inputs*. This result together with Lemma 1 (Section 6.2) implies:

If the compression function h is modeled as a RO, then the application of HMAC to any source distribution with min-entropy m produces an output that is $q2^{-m}$-close to uniform.

More precisely, as pointed out in Section 6.2, this is the case only for source distributions that are (sufficiently) *independent* from the function h. Thus, to obtain a generic extractor one needs to randomize the scheme, which we do by using a salt value (what we called the "extractor salt" *XTS*) to key HMAC. Similarly, combining the result from [17] and our Lemma 2, if h_κ is a RO family then HMAC over the family H_κ is a generic hard-core family.

NMAC as extractor. The following results from [23] apply to NMAC [9] and can be adapted to HMAC as indicated later (our presentation is informal, please refer to [23] for exact statements). They show that HMAC has a structure that supports its use as a generic extractor and, in particular, it offers a much better design for extraction than the plain hash function H.

Recall that $\mathsf{NMAC}_{\kappa_1,\kappa_2}(x) = H_{\kappa_2}(H_{\kappa_1}(x))$. We call H_{κ_2} the "outer function" and H_{κ_1} the "inner function". Also, the results below involve the notion of "almost universal (AU)" hash functions [15, 62]: A family h_κ is δ-AU if for any inputs $x \neq y$ and for random κ, $Prob(h_\kappa(x) = h_\kappa(y)) < \delta$. This is a natural (combinatorial) property of hash functions and also one that any (even mildly) collision resistant hash family must have and then a suitable assumption for cryptographic hash functions. Specifically, if the hash family h_κ is δ-collision-resistant against linear-time adversaries (i.e., such an attacker finds collisions in h_κ with probability at most δ) then h_κ is δ-AU [23].

In the cases below where the outer function of NMAC is modeled as a RO, this outer function can be a single random function (in which case the source distribution needs to be independent of this function) or a keyed family of random functions (in which case only the key needs to be chosen independently of the source distribution). We use m to denote the min-entropy of the source and q the number of RO queries performed by the attacker. [23] shows:
If the outer function is modeled as a RO and the inner function is δ-AU then NMAC produces an output that is $\sqrt{q(2^{-m} + \delta)}$-close to uniform. The same holds under the assumption that the inner function is δ-collision resistant against linear-time adversaries.

One natural question is whether one can ensure some good extraction properties for NMAC on the sole assumption that the family of compression functions h_κ is in itself a good extractor. For this, [23] show the following result that *does not resort to any idealized RO assumptions.* Denote by $\hat{h}_\kappa$ a family defined on the basis of the compression function h but where the key and input are swapped relative to the definition of h_κ. Then (refer to [23] for precise security quantification):

If h_κ is a good extractor family and $\hat{h}_\kappa$ is a pseudorandom family, then applying NMAC to a source distribution where each input block has min-entropy at least k (also when conditioned on other blocks) produces an output that is computationally indistinguishable from uniform.

An example of a practical application where this non-idealized result can be used is the IKE protocol [34, 43] where all the defined "mod p" DH groups have the required block-wise (computational) min-entropy. In particular, the output from our KDF is guaranteed to be pseudorandom in this case without having to model h as a random oracle.

Truncated NMAC. Stronger results can be achieved if one truncates the output of NMAC by c bits (e.g., one computes NMAC with SHA-512 but only outputs 256 bits). In this case one can show that NMAC is a good *statistical* extractor (not just computational) under the following sets of assumptions:

- *If h_κ is a family of random compression functions and the source distribution has min-entropy $\geq k$ and n input blocks, then truncated NMAC is $\sqrt{n2^{-c}}$-close to uniform.*
- *If the inner function is δ_1-AU and the outer is δ_2-AU then truncated NMAC is $\sqrt{2^c + \delta_1 + \delta_2}$-close to uniform for source distributions with min-entropy $\geq k$. The same holds if the inner function is δ_1-collision resistant against linear-time attackers.*

Note that the latter result is *unconditional*: it makes no idealized assumptions and ensures statistical extraction. This is an important result that can be applied in our KDF design; e.g., using SHA-512 truncated to 256 bits for extraction and SHA-256 as PRF (see Section 2).

From NMAC to HMAC. To use the above results with HMAC one needs to assume the computational independence of the values $h(\kappa \oplus \text{opad})$ and $h(\kappa \oplus \text{ipad})$ for random κ. In the cases where h is modeled as a random function this requires no additional assumption.

9 Other Designs and Related Work

Extract-then-Expand vs. Traditional. One of the best ways to understand and evaluate our KDF scheme, the extract-then-expand paradigm and its instantiation to HMAC, is to compare this approach to the existing KDFs in the literature and standards. With very few exceptions (IKE being the most notable) all common standards follow the "traditional approach" introduced and discussed in the Introduction. Refer to that section for a high-level comparison between this approach and ours. A few more remarks are in place.

Our extensive covering of extraction properties and schemes in previous sections shows that with the exception of the case where one assumes Hash to be a an idealized "random function", generic extractors (those that work well on any source with sufficient min-entropy) are *only* guaranteed to work on repeated inputs *when these inputs are independent from each other*. Even then the number of bits to extract are limited by the min-entropy of the source. The traditional scheme violates these limitations both in the repeated use of Hash on highly correlated inputs, e.g., on $SKM \parallel 0$ and $SKM \parallel 1$, as well as in the number of bits it outputs. To illustrate these points, consider an *unconditional* extractor such as the one implemented via random binary-matrix multiplication [15, 35] and mentioned in Section 3. In this scheme, the extractor's salt *XTS* corresponds to the description of a matrix M and the inputs to the extractor (in this example the strings $SKM \parallel i$) are treated as bit-vectors; the output of the extractor is the multiplication of the matrix M times the input vector. Since when considered as bit-vectors the strings $SKM \parallel 0$ and $SKM \parallel 1$ differ only in one bit, then their multiplication by M results in two vectors that differ exactly by the last column of M. However, this value is public (since M is the known salt) and therefore learning the output on input $SKM \parallel 0$ reveals also the output on $SKM \parallel 1$. This violates the essential independence requirement from a KDF, namely, information from some of the output bits (or keys) should provide no information on other bits.

One can argue that this example is irrelevant to the case where the extractor is implemented with SHA-1 (or the like) which does not have such a "linear behavior". However, what this example shows is that even if we eventually come up with a cryptographic hash function that does have good (maybe even provable) extraction properties, this property will give us *no assurance* in the output of the above KDF since we "abused" the extraction property by applying the hash to *dependent inputs*. (Note, by the way, that extraction properties should be part of the requirements for cryptographic hash functions; certainly for those that want the hash function to "emulate random functions".) A similar example applies to dedicated source-specific extractors. For example, [28] shows that the

LSB extractor, that outputs a number of least significant bits from its input, is a secure extractor when applied to DH keys over some specific groups. Now, if one applies the LSB extractor to $SKM \| 0$, where SKM is the DH key, one would either output the lsb's of the counter 0 (if the LSB extractor is applied to the entire input) or the LSBs of SKM (if the LSB extractor is applied to SKM only). In the first case, one end outputting known bits from the counter value into the key material, while in the latter the counter value is ignored and then the output from both $SKM \| 0$ and $SKM \| 1$ is the same. (Same moral as before: In the traditional scheme, even if the hash function is a good extractor, generic or specific, the security of the KDF is not guaranteed.)

Another shortcoming of the traditional scheme is that it will not take advantage of randomizing salt even if this is available (and we've seen many advantages of such a use throughout this paper). One could remedy this situation by including the salt as part of the *info* field but this clearly adds another level of ad-hoc design (or abuse) to the KDF. As usual, if Hash is a random oracle, any placement for the salt is good. But if we want to deal with real hash functions, we should better recognize the salt functionality and structure it into the KDF design.

Having a well-understood structure in the KDF design, as implied by the extract-then-expand approach, has many other virtues. In particular, the fact that one can "mix and match" different extraction modules with different PRFs. This can be useful for using, for example, an HMAC-based extractor step as in our scheme with a CBC-MAC PRF (these two stages may even be implemented separately, say in s/w and h/w, respectively). Or, using HMAC with SHA-512 as the extractor and HMAC with SHA256 as the PRF; a combination that, as pointed out in Section 2, has particularly good analytical properties.

Adams et al paper [2]. One of the few works in the literature that we are aware of that treats specifically the design of generic (multi-purpose) KDFs is the work of Adams et al [2]. While not formal in nature, the paper is instructive and includes a comprehensive description of many KDF standards. The main theme in the paper is a critique of existing schemes on the following basis (this applies to the traditional schemes as well as ours). Due to the way the hash function is used by these schemes, their strength (measured by the work factor it takes to guess the output of the KDF) is never more than 2^k where k is the output length of the hash function, and this is the case even when the entropy of the source key material is (much) larger than k. Indeed, in these schemes it suffices to guess $\mathsf{Hash}(SKM)$ to be able to compute all the derived key material. While we agree with this fact, we do not agree with the criticism. If a work factor of 2^k is considered feasible (say 2^{160}) then one should simply use a stronger hash function. Counting on a 2^k-secure hash function to provide more than 2^k security, even on a high-entropy source, seems as unnecessary as unrealistic.

Moreover, as we see next, the very scheme from [2] claimed to overcome the 2^k limitation does not achieve this goal. Specifically, [2] states the following as an explicit goal of a KDF: If the entropy of the source of key material is m and the number of bits output by the KDF and not provided to the attacker is m', then the work factor to learn these hidden m' output bits should be at least the minimum between 2^m and $2^{m'}$. In particular, in the case of a hash-based KDF, this work factor should be independent of the output length of the hash function. The scheme proposed in [2] is the following. On input source key material SKM the KDF outputs the concatenation of $K(1)\ldots, K(t)$ where $K(1) = H(S); K(i) = H_{K(i-1)}(S),\ i = 2, \ldots, t$, (here H_K denotes the hash function H with the IV set to K). In Appendix D we show, however, that this scheme does not achieve the intended goal, by presenting an attack that retrieves the full KDF output in much less than the bounds set as the scheme's security.

Therefore, we repeat, if a 2^k work factor is deemed feasible, then go for a stronger hash function. It is better to focus on the right structure of the KDF and to avoid the multiple extraction from

the same *SKM* value which is common to the traditional scheme as well as to the scheme from [2].

NIST's variant of the traditional scheme. NIST's KDF is defined in [57, 19] and intended for the derivation of keys from a secret shared via a key-exchange protocol. The scheme is a variant of the "traditional KDF scheme" discussed before (see Section 1). The difference is in the ordering of the value *SKM* and the counter which are swapped relative to the traditional scheme, namely, NIST's scheme is:

$$KM = \mathsf{Hash}(\text{“1”} \parallel SKM \parallel info) \parallel \mathsf{Hash}(\text{“2”} \parallel SKM \parallel info) \parallel \ldots \parallel \mathsf{Hash}(\text{“}t\text{”} \parallel SKM \parallel info)$$

One can wonder about the reasons for this deviation from the traditional scheme; unfortunately, NIST documents do not offer such rationale. Clearly, this change does not help against the problems we identified in the traditional scheme, such as the potential vulnerabilities to extension attacks, the sensitivity to prefix-free encoding, and the more structural issues arising from the repeated application of the hash function to the same *SKM* value and input dependencies. Moreover, when NIST's KDF is examined as a PRF (namely, when the key material *SKM* is strong enough as a PRF key) it results in a *very non-standard* PRF that inherits weaknesses from both the "key-prepend mode" (i.e., $\mathsf{Hash}(K \parallel \cdot)$) and the "key-append mode" (i.e., $\mathsf{Hash}(\cdot \parallel K)$). A possible clue into NIST's rationale can be found in [2] where the prepended counter is discussed as a possible defense against the "2^k attack" mentioned before. However, NIST's own document [19] correctly points out that their KDF scheme is not intended to provide security beyond 2^k. And, as said, if anyone is worried about security beyond, say 2^{160}, then just choose a stronger hash.

Another peculiarity of NIST's KDF is that the document ([57], Sec. 5.8) explicitly forbids (with a strong **shall not**) the use of the KDF for generating non-secret randomness, such as IVs. If there is a good reason to forbid this, namely the KDF is insecure with such a use, then there must be something very wrong about this function (or, at least, it shows little confidence in the strength of the scheme). First, it is virtually impossible to keep general applications and implementors from using the KDF to derive auxiliary random material such as IVs. Second, if the leakage of an output from the KDF, in this case the public IV, can compromise other (secret) keys output by the KDF, then the scheme is fully broken; indeed, it is an essential requirement that the leakage of one key produced by the KDF should not compromise other such keys.[13]

Finally, NIST's KDF is "over-specified" in that it mandates specific information to be used in the *info* field such as parties identities. This makes the KDF specific to key-exchange (KE) protocols (this may be a deliberate choice of NIST but not one that helps on standardizing a multi-purpose KDF). Moreover, this specification excludes perfectly fine KE protocols, such as IKE, where party identities are not necessarily known when the KDF is called. (NIST mandated the use of identities for reasons that are specific to the KE protocols defined in [57] and not as a general principle applicable to *all* sound KE protocols.)

IKE. As we mentioned earlier, the extract-then-expand approach to KDF design was first adopted (at least in the setting of real-world protocols) with our design of the KDF of the IKE protocol (IPsec's Key Exchange) [34]. The scheme presented here is very close to that design. What has changed substantially since the original design of IKE's KDF is the understanding of the theoretical foundations for this design. The present work is intended to stress the virtues of this design as a generic multi-purpose KDF, and to present in detail the design rationale based on this new understanding.

Other related work. We have mentioned already [23, 17], and used them extensively in Sec-

[13] It seems that NIST's spec assumes another source of public randomness from which IVs and similar values are derived. If such a source is available it should be used to salt the KDF as we recommend here.

tion 8, as two works that serve as the more direct theoretical foundations for our KDF design and the HMAC-specific implementation. A related paper is [16] which contains a good discussion of extraction issues in the context of key derivation functions in the Diffie-Hellman setting. In particular, it presents a dedicated deterministic extractor for specific DH groups. Another such extractor (very different in techniques and applicability) is presented in [28].

Password-based KDF. One specific area where KDFs are used is in deriving cryptographic keys from passwords. For these password-based KDFs (PBKDF) the goal is to derive a key from a password in such a way that dictionary attacks against the derived key will not be practical. For this, typical PBKDFs (e.g., PKCS #5 [44]) use two main ingredients: (i) salt to prevent building universal dictionaries, and (ii) the slowing down of the KDF operation via repeated hash computations (the latter technique referred as "key stretching" in [45]). This makes PBKDFs very different than the general-purpose KDFs studied here. In particular, while passwords can be modeled as a source of keying material, this source has too little entropy to meaningfully apply our extractor approach except when modeling the hash function as a random oracle. Also the slowing-down approach of PBKDFs is undesirable for non-password settings. Our two-stage KDF approach from Section 2 could still be used to build PBKDFs by replacing the extraction stage with a key-stretching operation (implemented via some hash repetition technique), or by interleaving such key stretching between the extraction and expansion phases. In the latter case, one would first derive an intermediate key P from the password pwd and salt XTS as $P = \mathsf{HMAC}(XTS, pwd)$, then apply key stretching to P to obtain a key PRK, and finally expand PRK to the number of required cryptographic keys via PRF^*. The interested reader should consult [63] for a formal treatment of PBKDFs and [45] for the properties required from key-stretching techniques and their underlying hash functions. We point out that another formalism that may help bridging between our extraction approach and PBKDFs is the notion of *perfectly one-way probabilistic hash functions* from [14].

Applications to System RNGs. KDFs can be used as a basis for a system "random number generator" (RNG) which typically consists of some sampling process used to "extract and renew" randomness from some source in order to seed a pseudorandom generator (PRG). The first work to treat this problem formally is [6] which also proposes the use of the extract-then-expand approach to the construction of such generators. This work bears many common elements with ours although the (multi-purpose) KDF functionality presents a wider range of applications and scenarios, and hence requires a more comprehensive treatment which we offer here. On the other hand, [6] provides a much more detailed and formal study of the specific system RNG application (taking care of randomness refreshing, dealing with partial leakage of internal state, etc.). We note that our KDF design, specifically our HMAC-based instantiation, is well suited for the above application as well, and would result in a much sounder RNG than many of the ad-hoc constructions used in practice (c.f., [24]). Another related work is [8] which proposes the use of combinatorial extractors in the design of *physical* random-number generators (see the first example in Section 4) and points out to the potential practicality of these extractors in this setting. Both [6, 8] offer interesting perspectives on the use of randomness extractors in practice that complement our work.

10 Conclusions

In spite of their central role in almost every cryptographic system, the design of multi-purpose key derivation functions (KDF) has traditionally been carried in ad-hoc ways with limited analytical foundation. This is especially true for KDFs based on cryptographic hash functions (which is the most common case in practice) where hash functions are "abused" by assuming that they behave

as perfect random functions. Research results obtained in recent years point out to the advantages of building KDF functions based on HMAC rather than on plain hash functions as traditional schemes do. In particular, these results show that an HMAC-based KDF can be founded on weaker assumptions on the underlying hash function. This is somewhat analogous to the use of HMAC as PRF that has been favored in the last ten years over plain-hash PRFs especially due to HMAC's better resilience to weaknesses in the underlying hash function (e.g., collision attacks).

In this paper we have provided extensive analysis and evidence to the above advantages of HMAC in the context of KDF design, especially when the KDF is intended as a single design that can serve multiple applications under a wide variety of usage scenarios and requirements. Above all we have strived to found our design on the extract-then-expand paradigm for KDF design and have offered detailed rationale for this approach in the paper. The end result is a conservative, yet efficient, KDF scheme that exercises much care in the way it utilizes a cryptographic hash function. In addition, the fact that our scheme is based on the extract-then-expand approach allows for its instantiation under other cryptographic primitives, particularly using block ciphers such as AES.

The KDF study in this paper follows the (complexity-based) computational approach to cryptographic analysis. The value of this analysis depends on the underlying assumptions on the strength of specific cryptographic primitives and relies heavily on modeling choices. Therefore, one should not see such analysis as an indication of "absolute security" (an oxymoronic notion anyway) but rather as a demonstration of the correct structure of a design as well as its versatility and superiority to alternatives.

References

[1] M. Abdalla, M. Bellare, and P. Rogaway, "The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES", *CT-RSA '01*, LNCS No. 2020., pages 143–158, 2001.

[2] Carlisle Adams, Guenther Kramer, Serge Mister and Robert Zuccherato, "On The Security of Key Derivation Functions", ISC'2004, LNCS 3225, 134-145.

[3] Werner Alexi, Benny Chor, Oded Goldreich, Claus-Peter Schnorr: RSA and Rabin Functions: Certain Parts are as Hard as the Whole. SIAM J. Comput. 17(2): 194-209 (1988)

[4] ANSI X9.42-2001: Public Key Cryptography For The Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography.

[5] ANSI X9.63-2002: Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport.

[6] Boaz Barak, Shai Halevi: A model and architecture for pseudo-random generation with applications to /dev/random. ACM Conference on Computer and Communications Security 2005.

[7] Boaz Barak, Russell Impagliazzo, and Avi Wigderson, "Extracting randomness from few independent sources", FOCS 04.

[8] Boaz Barak, Ronen Shaltiel, and Eran Tromer, "True random number generators secure in a changing environment", Cryptographic Hardware and Embedded Systems (CHES), LNCS no. 2779, 2003.

[9] M. Bellare, R. Canetti, and H. Krawczyk. "Keying hash functions for message authentication", *Crypto '96*, LNCS No. 1109. pages 1–15, 1996.

[10] M. Bellare, R. Canetti, and H. Krawczyk. "Pseudorandom Functions Revisited: The Cascade Construction and Its Concrete Security", *Proc. 37th FOCS*, pages 514–523. IEEE, 1996.

[11] Mihir Bellare, "New Proofs for NMAC and HMAC : Security Without Collision-Resistance", CRYPTO 2006, LNCS 4117. pp. 602-619.

[12] Mihir Bellare, Phillip Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols", ACM Conference on Computer and Communications Security 1993.

[13] Manuel Blum, Silvio Micali: How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. SIAM J. Comput. 13(4): 850-864 (1984)

[14] R. Canetti, D. Micciancio, and O. Reingold, "Perfectly one-way probabilistic hash functions", Proc. STOC'98, pp. 131-140.

[15] L. Carter and M. N. Wegman. "Universal Classes of Hash Functions", *JCSS*, 18(2), 1979.

[16] Olivier Chevassut, Pierre-Alain Fouque, Pierrick Gaudry, David Pointcheval: The Twist-AUgmented Technique for Key Exchange. Public Key Cryptography 2006: LNCS 3958.

[17] Jean-Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, Prashant Puniya: "Merkle-Damgard Revisited: How to Construct a Hash Function", CRYPTO'05, LNCS 3621, 430-448.

[18] I. Damgard. A Design Principle for Hash Functions. *Crypto '89*, LNCS No. 435, pages 416–427.

[19] Q. Dang and T. Polk, "Hash-Based Key Derivation (HKD)", draft-dang-nistkdf-01.txt (work in progress), June 23, 2006.

[20] T. Dierks and C. Allen, ed., "The TLS Protocol – Version 1", *Request for Comments 2246*, 1999.

[21] T. Dierks and E. Rescorla, ed., "The TLS Protocol – Version 1.1", *Request for Comments 4346*, 2006.

[22] W. Diffie, P. van Oorschot and M. Wiener, "Authentication and authenticated key exchanges", *Designs, Codes and Cryptography*, 2, 1992, pp. 107–125

[23] Dodis, Y., Gennaro, R., Håstad, J., Krawczyk H., and Rabin, T., "Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes", Crypto'04, LNCS 3152.

[24] Leo Dorrendorf, Zvi Gutterman and Benny Pinkas, "Cryptanalysis of the Windows Random Number Generator", ACM Computer and Communications Security Conference (CCS'07), 2007.

[25] P. Eronen and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, Dec. 2005.

[26] A. Fiat, A. Shamir, "How to Prove Yourself: Practical Solutions to Identification and Signature Problems", CRYPTO 1986: 186-194.

[27] R. Fischlin, C.-P. Schnorr, "Stronger Security Proofs for RSA and Rabin Bits", Eurocrypt'97.

[28] P.-A. Fouque, D. Pointcheval, J. Stern, S. Zimmer, "Hardness of Distinguishing the MSB or LSB of Secret Keys in Diffie-Hellman Schemes", ICALP (2) 2006: LNCS 4052.

[29] R. Gennaro, H. Krawczyk and T. Rabin. "Secure Hashed Diffie-Hellman over Non-DDH Groups", Eurocrypt'04.

[30] Rosario Gennaro, Luca Trevisan: Lower Bounds on the Efficiency of Generic Cryptographic Constructions. FOCS 2000: 305-313.

[31] Oded Goldreich, Leonid A. Levin: A Hard-Core Predicate for all One-Way Functions. STOC 1989.

[32] Oded Goldreich, *Foundations of Cryptography*. Cambridge University Press, 2001.

[33] S. Goldwasser and S. Micali, "Probabilistic Encryption", *JCSS*, 28(2):270–299, April 1984.

[34] D. Harkins and D. Carrel, ed., "The Internet Key Exchange (IKE)", *RFC 2409*, Nov. 1998.

[35] J. Hastad, R. Impagliazzo, L. Levin, and M. Luby. "Construction of a Pseudo-random Generator from any One-way Function", *SIAM. J. Computing*, 28(4):1364–1396, 1999.

[36] Johan Hstad, Mats Nslund: Practical Construction and Analysis of Pseudo-Randomness Primitives. ASIACRYPT 2001: 442-459.

[37] Johan Håstad, Mats Naslund: The security of all RSA and discrete log bits. J. ACM 51(2): 187-230 (2004).

[38] J. Hastad, A. Schrift, A. Shamir, "The Discrete Logarithm Modulo a Composite Hides O(n) Bits," *J. Comput. Syst. Sci.*, 47(3): 376-404 (1993)

[39] Chun-Yuan Hsiao, Chi-Jen Lu, Leonid Reyzin, "Conditional Computational Entropy, or Toward Separating Pseudoentropy from Compressibility", EUROCRYPT 2007, pp. 169-186

[40] IEEE P1363A: Standard Specifications for Public Key Cryptography: Additional Techniques, Institute of Electrical and Electronics Engineers.

[41] I. Impagliazzo and D. Zuckerman, "How to Recycle Random Bits", in Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science, pp. 248–253, 1989.

[42] Tetsu Iwata and Kaoru Kurosawa, "OMAC: One-Key CBC MAC", FSE 2003: 129-153.

[43] C. Kaufman, ed., "Internet Key Exchange (IKEv2) Protocol", RFC 4306, Dec. 2005.

[44] B. Kaliski, PKCS #5: Password-Based Cryptography Specification Version 2.0, RFC 2898, Sept. 2000.

[45] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure Applications of Low-Entropy Keys", *ISW 1997*, LNCS 1396, pp. 121-134.

[46] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, Feb. 1997.

[47] Krawczyk, H., "SKEME: A Versatile Secure Key Exchange Mechanism for Internet", *1996 Internet Society Symposium on Network and Distributed System Security*, pp. 114–127.

[48] H. Krawczyk. "SIGMA: The 'SiGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols", *Crypto '03*, pages 400–425, 2003

[49] Hugo Krawczyk: HMQV: A High-Performance Secure Diffie-Hellman Protocol. CRYPTO 2005. LNCS 3621: 546-566.

[50] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone, "An efficient Protocol for Authenticated Key Agreement", *Designs, Codes and Cryptography, 28, 119-134, 2003.*

[51] Ueli M. Maurer, Renato Renner, Clemens Holenstein: Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. TCC 2004: 21-39.

[52] A. Menezes, P. Van Oorschot and S. Vanstone, "Handbook of Applied Cryptography," CRC Press, 1996.

[53] Ralph C. Merkle: One Way Hash Functions and DES. CRYPTO 1989: 428-446.

[54] Moni Naor, Omer Reingold: Synthesizers and Their Application to the Parallel Construction of Pseudo-Random Functions. J. Comput. Syst. Sci. 58(2): 336-375 (1999).

[55] N. Nisam and A. Ta-Shma. "Extracting Randomness: A Survey and New Constructions", *JCSS*, 58:148–173, 1999.

[56] N. Nisan and D. Zuckerman. "Randomness is linear in space", *J. Comput. Syst. Sci.*, 52(1):43–52, 1996.

[57] NIST Special Publication (SP) 800-56A, Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography, March 2006.

[58] S. Patel and G. Sundaram. "An Efficient Discrete Log Pseudo Random Generator", *Crypto '98*, LNCS No. 1462, pages 304–317, 1998.

[59] R. Shaltiel. "Recent developments in Extractors", Bulletin of the European Association for Theoretical Computer Science, Volume 77, June 2002, pages 67-95. Available at: http://www.wisdom.weizmann.ac.il/~ronens/papers/survey.ps

[60] Adi Shamir and Boaz Tsaban, "Guaranteeing the Diversity of Number Generators," *Information and Computation*, Vol. 171, Issue 2, 2001, pp. 350-363

[61] V. Shoup, "Lower Bounds for Discrete Logarithms and Related Problems", Eurocrypt'97, LNCS 1233, pp. 256-266

[62] Douglas R. Stinson: Universal Hashing and Authentication Codes. Des. Codes Cryptography 4(4): 369-380 (1994).

[63] Frances F. Yao, Yiqun Lisa Yin, "Design and Analysis of Password-Based Key Derivation Functions," CT-RSA 2005.

A Glossary

In this section we recall formal definitions for some of the notions used throughout this work. In the next section we introduce new definitions for key derivation functions and sources of keying materials.

Notation. In the sequel $\mathcal{X}$ and $\mathcal{Y}$ denote two (arbitrary) probability distributions over a common support set A; X and Y denote random variables drawn from $\mathcal{X}$ and $\mathcal{Y}$, respectively.

Definition 3 *We say that probability distributions $\mathcal{X}$ and $\mathcal{Y}$ have* statistical distance δ *(or are* δ-close*) if* $\sum_{a \in A} |Prob(X = a) - Prob(Y = a)| \leq \delta$.

Definition 4 *An algorithm D is an* ε-distinguisher *between distributions $\mathcal{X}$ and $\mathcal{Y}$ if* $|Prob(D(X) = 1) - Prob(D(Y) = 1| < \varepsilon$.

We note that two distributions $\mathcal{X}$ and $\mathcal{Y}$ are δ-close iff there is no ε-distinguisher between $\mathcal{X}$ and $\mathcal{Y}$ for $\varepsilon > \delta$.

By restricting the computational power of the distinguisher in the above definition one obtains the following well-known definition of "computational indistinguishability" [33] (we formulate the definition using the "concrete security" (t, ε) approach as a non-asymptotic alternative to the classical polynomial-time treatment).

Definition 5 *Two probability distributions $\mathcal{X}$, $\mathcal{Y}$ are* (t, ε)-computationally indistinguishable *if there is no ε-distinguisher between $\mathcal{X}$ and $\mathcal{Y}$ that runs in time t.*

Definition 6 *A probability distribution $\mathcal{X}$ over the set $\{0,1\}^n$ is called* (t, ε)-pseudorandom *if it is (t, ε)-computationally indistinguishable from the uniform distribution over $\{0,1\}^n$.*

Definition 7 *A probability distribution $\mathcal{X}$ has* min-entropy m *if for all $a \in A$, $Prob(X = a) \leq 2^{-m}$.*

Definition 8 *A probability distribution $\mathcal{X}$ has* (t, ε)-computational min-entropy m *if there exists a distribution $\mathcal{Y}$ with min-entropy m such that $\mathcal{X}$ and $\mathcal{Y}$ are (t, ε)-computationally indistinguishable.*

Notes on terminology. In our presentation in this paper we often use the above terms without explicit mention of the (t, ε) parameters (e.g, we talk about computational indistinguishability – or just indistinguishability – rather than the more accurate (t, ε)-computational indistinguishability). In these cases it is understood that the missing t is an amount of infeasible computation (for the attacker) while ε is typically a small or even negligible quantity. More generally, the ratio t/ε is assumed to be larger than the expected running time of any feasible attacker. As another

shorthand we use the term "entropy" instead of min-entropy (or even instead of (t, ε)-computational min-entropy). This is to be understood from the context especially that min-entropy is the only notion of entropy used in this work. One can also think of these terms in their usual polynomial-time meaning, but in this case one cannot talk of individual (finite) probability distributions as above but rather of an infinite collection of such distributions indexed by a security parameter. Finally, we note that in this paper we often use, informally, the notion of *conditional min-entropy* (both in the statistical and computational settings) with its natural meaning; in the next section we make this notion precise in our context.

Next we present a definition of a statistical extractor. This notion originates with [56] and the definition given here corresponds to what is called a *strong* extractor in the complexity-theory literature; "strong" means that the output of the extractor is close to uniform even when the salt is known. Since this is the only notion we use we drop the "strong" qualifier. On the other hand, we add the word "statistical" to differentiate those from computational extractors.

Definition 9 *A function* $\mathsf{ext} : \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^\ell$ *is a* (m, δ)-statistical extractor *if for any distribution* $\mathcal{X}$ *over* $\{0,1\}^n$ *with min-entropy* m*, the statistical distance between the distribution of pairs* (r, y)*, where* r *is chosen with uniform probability over* $\{0,1\}^t$ *and* $y = \mathsf{ext}_r(x)$ *for* x *chosen according to distribution* $\mathcal{X}$*, is* δ*-close to the distribution of pair* (r, z) *where* r *and* z *are chosen with uniform probability from* $\{0,1\}^t$ *and* $\{0,1\}^\ell$ *respectively.*

Our main application of extractors in this work does not require the output to be statistically close to uniform but just "computational close". This notion of computational extractors has been used in [29, 23].

Definition 10 *An* $(m, \delta, t, \varepsilon)$-computational extractor *is defined as in Definition 9 except that the requirement for statistical closeness between the distributions* (r, y) *and* (r, z) *is replaced with* (t, ε)*-computational indistinguishability.*

B Formalizing Key Derivation Functions

In this appendix we present a formal definition of (secure) key derivation functions and a formalization of what is meant by a "source of keying material". To the best of our knowledge, no such general definitions have been given in the literature.[14]

Note on computational model. The definitions in this section are presented without the explicit (t, ε) treatment of computational bounds as in the previous section. This is done for clarity and since adding these parameters to a definition is straightforward. As before, we note that these definitions can also be stated in terms of polynomial-time (but objects such as individual probability distributions need to be changed to infinite collections indexed by a security parameter, etc.).

We start with a definition of KDF in terms of its inputs and outputs (consistent with the KDF description in Section 2). Later, after introducing the notion of sources of keying material, we define what it means for a KDF to be secure.

Definition 11 *A* key derivation function (KDF) *is an efficient algorithm that accepts as input four arguments: a value* σ *sampled from a source of keying material (see Definition 12), a length value* ℓ*,*

[14] Yao and Yin [63] provide a formal definition of KDFs specific to the password setting which, as noted in Section 9, is different from and inapplicable to the general setting treated here.

and two additional arguments, a salt value *r defined over a set of possible salt values and a* context variable *c, both of which are optional, i.e., can be set to a null string.*[15] *The KDF output is a string of ℓ bits.*

The security and quality of a KDF will depend on the properties of the source from which the input σ is chosen. Next, we formalize the notion of a "source of keying material".

Definition 12 *A* source of keying material *(or simply* source*) Σ is a two-valued probability distribution generated by an efficient probabilistic algorithm.* (We will refer to both the probability distribution as well as the generating algorithm by Σ.)

This definition does not specify the input to the Σ algorithm (but see below for a discussion related to potential adversary-chosen inputs to such an algorithm). It does specify the form of the output: a pair (σ, κ) where σ (the "sample") represents the (secret) source key material to be input to a KDF, while κ represents some knowledge about σ (or its distribution) that is public and, in particular, known to an adversary. For example, in a Diffie-Hellman application the value σ will consist of a value g^{xy} while κ could represent a quintuple (p, q, g, g^x, g^y). In a different application, say a random generator that hashes samples of system events in a computer system, the value κ may include some of the sampled events used to generate σ. The importance of κ in our formal treatment is that we will require a KDF to be secure on inputs σ *even when the knowledge value κ is given to the attacker.*

The following definition defines the security of a KDF with respect to a specific source Σ. See Definition 15 for a more general case.

Definition 13 *A key derivation function* KDF *is said to be* secure with respect to a source of key material *Σ if no feasible attacker $\mathcal{A}$ can win the following distinguishing game with probability significantly better than 1/2:*

1. *The algorithm Σ is invoked to produce a pair σ, κ.*
2. *A salt value r is chosen at random from the set of possible salt values defined by* KDF *(r may be set to a constant or a null value if so defined by* KDF*).*
3. *The attacker $\mathcal{A}$ is provided with κ and r.*
4. *$\mathcal{A}$ chooses arbitrary values ℓ and c.*
5. *A bit $b \in_R \{0,1\}$ is chosen at random. If $b = 0$, attacker $\mathcal{A}$ is provided with the output of $\mathsf{KDF}(\sigma, r, \ell, c)$, else $\mathcal{A}$ is given a random string of ℓ bits.*
6. *$\mathcal{A}$ outputs a bit $b' \in \{0,1\}$. It wins if $b' = b$.*

It is imperative for the applicability of this definition that the attacker be allowed to see both κ and r before deciding on the bit b'. This models the requirement that the KDF needs to remain secure even when the side-information κ and salt r are known to the attacker. Note that also the choice of c and ℓ may depend on κ and r but not on σ.[16]

[15] The values σ, ℓ, r, c correspond to the values $SKM, L, XTS, CTXinfo$ in the description of Section 2.

[16] In some applications the value of c may be chosen by the legitimate party that has access to the value of σ. Yet, the value c should be chosen independently of σ also in this case. (See Section 2.)

For simplicity we defined the above game in which $\mathcal{A}$ only inputs a single input ℓ, c to the function. This can be extended to allow for multiple such queries with the *same* value σ (in which case all queries except the last one are answered with the real output of KDF and only the last one is answered according to the bit b – obviously, the attacker is not allowed to query the same c with two different values ℓ)[17]. Allowing for multiple values of c under the same input σ to KDF ensures that even if an attacker can force the use of the same input σ to the KDF in two different contexts (represented by c), still the outputs from the KDF in these cases are computationally independent (i.e., leak no useful information on each other).

The following definition extends the min-entropy definitions from Appendix A to the setting of keying material sources (for a detailed treatment of *conditional* (computational) entropy as used in the next definition see [39]).

Definition 14 *We say that Σ is a* statistical m-entropy source *if for all s and k in the support of the distribution Σ, the conditional probability $Prob\,(\sigma = s \mid \kappa = k)$ induced by Σ is at most 2^{-m}. We say that Σ is a* computational m-entropy source *(or simply an m-*entropy source*) if there is a statistical m-entropy source Σ' that is computationally indistinguishable from Σ.*

We note that in the above definition we can relax the "for all k" to "all but a negligible fraction of k". That is, we can define $\kappa = k$ to be "bad" (for a given value m) if there is s such that $Prob\,(\sigma = s \mid \kappa = k) > 2^{-m}$ and require that the joint probability induced by Σ on bad k's be negligible.

Definition 15 *A KDF function is called m-*entropy secure *if it is secure for all m-entropy sources.*

We note for the most part of this paper the (implicit) notion of security of a KDF corresponds to this last definition, namely, we think of KDFs mainly as a *generic* function that can deal with different sources as long as the source has enough computational min-entropy. As we have repeatedly noted, this notion of security can only be achieved for randomized KDFs where the salt value r is chosen at random from a large enough set. Yet, the paper also touches (e.g., Section 5), on deterministic KDFs that may be good for specific applications and sources and whose security is formalized in Definition 13.

On adversarially-chosen inputs to Σ. While the algorithm that generates the source Σ may have arbitrary inputs, these inputs are not provided (by our formalism) to the attacker. All side information that the attacker may have on the source Σ is modeled via the κ output. But how about situations where not only has the attacker side information on the source but it can actually influence the generation of the distribution Σ? For example, an attacker can influence system events sampled by a random number generator or it can choose the parameters p, q, g (or an elliptic curve group) in a Diffie-Hellman application. Such influence could be modeled by letting the attacker choose some inputs to Σ. This, however, requires much care. First, one needs to restrict inputs only to a "legitimate set" that will ensure sufficient min-entropy in the source (e.g., disallowing $g = 1$ in a DH application). More significantly, however, is the need for independence between the source Σ and the salt value used by the KDF. Allowing the attacker to influence Σ after seeing the salt value r may result in a completely insecure KDF. As an extreme example, imagine a scenario where the algorithm Σ accepts as input a value v and then for every output σ it outputs $\kappa = \mathsf{ext}_v(\sigma)$ where ext is an extractor function that also implements the extract part of a KDF. In this case, if the

[17]The latter restriction can be removed for our KDF if the length ℓ is also input into PRF^* in the expansion stage.

attacker knows the value of the random salt r in advance (as it is the case in some applications), it could input $v = r$ into Σ and hence learn, via κ, the whole output from the KDF. Thus, we have to impose careful limitations on the attacker's ability to influence the (input to) Σ source as to enforce sufficient independence with a salt value available a-priori to the attacker.

We do not pursue this formalism any further here. Instead, we refer the interested reader to [8] where the above restrictions on adversarial inputs are carefully formalized via the notion of *t-resilient extractors*. One crucial point to realize is that applications need to be designed to *enforce* sufficient independence between the source and the KDF's salt. For example, in a Diffie-Hellman protocol where honest parties choose their own salt for the KDF, this salt needs to be transmitted to the peer in an *authenticated* way or else the attacker can choose the salt itself. In the example of a random number generator that fixes a non-secret random salt for its internal extractor, the application may allow some level of attacker's intervention in the generation of the source to the RNG but it must restrict it enough as to ensure the necessary level of independence with the salt value. See [8] for an extensive discussion of these issues.

C Proof of Lemma 2

Lemma 2 *Assume f is a $(\tau, 2^t, \varepsilon)$-OWF, namely f can be computed in time τ but cannot be inverted (over random inputs) with probability better than ε in time less than 2^t (in case of non-permutation, inversion means finding any pre-image). If H is an independent random function with k bits of output then the probability distribution $(f(x); H(x))$ over uniformly distributed x is (q, T, ε)-indistinguishable from $(f(x); U_k)$ (that is, no distinguisher running time T and making q queries to H has distinguishing advantage better than ε) where $T = 2^t - q\tau$.*

Proof: Given distinguisher D that runs time T_D, queries H at most q times, and distinguishes with advantage ε, we build inverter I as follows: On input $y = f(x)$, I provides D with input (y, z) where $z \in_R \{0,1\}^k$. For every query x' made by D to H, I checks whether $f(x') = y$; if so I outputs x' and halts, else if D halts before querying a preimage of y, then I ouputs fail. Clearly the work of I is at most $T_I = T_D + q\tau$ and its probability of success is equal to the probability, p, that D queries H on a pre-image of y. Intuitively, D has no chance to distinguish without querying such pre-image, so the probability p must be at least as the advantage ε of D. Next, we formally prove the above, namely that the probability p that D queries a pre-image of y (or equivalently that I inverts) is at least ε.

We need to analyze the work of D in two cases, when z is random and when $z = H(x)$. We are given that

$$\varepsilon = |Prob_{z \in_R \{0,1\}^k}(D = 1) - Prob_{z=H(x)}(D = 1)| \tag{1}$$

where probabilities are taken over the coins of D, the choice of x (and its induced distribution on y), the choice of random H (in the $z = H(x)$ case) and the choice of random z (in the uniform case). Denoting by p the probability that D queries a preimage of y, we have that

$$\begin{aligned} Prob(D = 1) &= Prob\,(D = 1\text{: } D \text{ queries preimage})\ p & (2)\\ &\quad + Prob\,(D = 1\text{: } D \text{ does not query preimage})\ (1 - p) & (3)\end{aligned}$$

Now, note that the value of p is the same in the case that $z = H(x)$ than in the case that z is uniform. Indeed, as long as D did not query a preimage there is no distinction between a uniformly chosen z and the uniformly chosen $H(x)$. For the same reason, the probability

$Prob\,(D = 1$: D does not query preimage) is the same in the two cases. Thus, the expression (1) can be re-written as the difference between the expression (2) in the case $z \in_R \{0,1\}^k$ and the expression (2) in the case $z = H(x)$. Canceling the equal term $Prob\,(D = 1$: D does not query preimage$)(1-p)$ in both expressions and using the fact that p is the same we get

$$\begin{aligned}\varepsilon &= |Prob_{z\in_R\{0,1\}^k}(D=1) - Prob_{z=H(x)}(D=1)| \\ &= p\,|Prob_{z\in_R\{0,1\}^k}(D=1:\ D \text{ queries preimage}) - Prob_{z=H(x)}(D=1:\ D \text{ queries preimage})| \\ &\leq p\end{aligned}$$

(last inequality due to the fact that the probabilities difference is at most 1).
Hence, we showed $\varepsilon \leq p$, namely, the inversion probability of I is at least ε. □

D Attack on the Adams et al. KDF Scheme

We build a counter-example using the following scenario. The hash block size is 512 bits and the source S consists of strings of length 1024 and has entropy m (for simplicity of presentation we assume that in the computations of $K(i)$ the length appending of typical hash functions is omitted; the argument holds also with length appending). There are numbers m_1, m_2, each smaller than 512, such that $m = m_1 + m_2$, and $m = 2k$. The first half of S is taken uniformly from a set M_1 of size 2^{m_1} and the second half is taken uniformly from a set M_2 of size 2^{m_2}, and both M_1 and M_2 can be efficiently enumerated (e.g., M_1 is the set of all strings with $512 - m_1$ least significant bits set to 0 and the rest chosen uniformly).

We now mount the following "meet in the middle" attack. Say the output of the KDF is $K_1, \ldots, K_t$ for $t \geq 4$ (we are using notation K_i instead of $K(i)$) and we are given K_1 and K_2 (i.e., $m' \geq 2k$). We find the value of S by running the following procedure, and from S we compute the remaining values $K_3, K_4, ...$

1. For each $S_2 \in M_2$ and $K \in \{0,1\}^k$, check if $H_K(S_2) = K_1$. Create a table T for all pairs (K, S_2) passing the check.

2. For each $S_1 \in M_1$ check if the pair $(H(S_1), S_2)$ is in T for some S_2. If so check if $H_{K_1}(S_1||S_2) = K_2$. If both checks are satisfied output $S = S_1||S_2$. If no such pair is found return 'fail'.

First, note that if the above procedure returns S then we have that $H(S) = K_1$ and $H_{K_1}(S) = K_2$. Moreover, since we know that such S exists (K_1, K_2 are given to us as the output produced by such a value S) then the procedure never fails. The question is whether the (first) value found is the correct S. Now, assuming a random H, the number of values S expected to produce a given K_1 and a given K_2 is $2^m/2^{2k}$. Since we choose $m = 2k$ then this value is 1 and hence we find the correct S with high probability. Clearly, once we found S we can compute the remaining K_i's.

The complexity of the above procedure is as follows. We need time 2^{k+m_2} for step 1. To help in step 2 we need to sort the table T. Since the expected size of T is 2^{m_2} (we go through 2^{k+m_2} values each with probability 2^{-k} to hit the value K_1) then the sorting work is negligible relative to 2^{k+m_2}. Step 2 itself takes 2^{m_1} checks where each check entitles a table lookup into T which takes $O(\log |T|) = O(m_2)$. In all the work is order of $2^{k+m_2} + 2^{m_1} m_2$. Putting $m_1 = 1.5k$ and $m_2 = 0.5k$ we get that the whole work takes approximately $2^{1.5k}$ which is much less than the 2^{2k} time that [2] intended the work of finding K_3 and K_4 (and beyond) to take.

Note 1: if we assume that the block K_2 is used by a specific algorithm then we do not even need to learn K_2 to mount the above attack but only have a way to test it (e.g., if K_2 is used to key an encryption algorithm then we need plaintext-ciphertext pairs computed under that key).

Note 2: If we replace H with HMAC in Adams we have the same meet-in-the-middle attack; even if salted with *XTS*. The attack is also possible against our scheme but we do not claim security beyond 2^k.