HKDF: Key Derivation and Extraction in Practice

Hugo Krawczyk
IBM Research

(more references at the end)
Key Derivation Functions (plan)

- Extract-then-Expand approach
- HKDF (new KDF standard)
  - WhatsApp, Facebook Messenger, Google QUIC and Allo, Signal, TLS 1.3, NIST, ...
- HKDF design and rationale
- Sample results
- Applications
Key Derivation Functions (KDF)

- A truly fundamental primitive in applied cryptography
  - A process producing cryptographic keys out of some initial input
  - A somewhat overlooked crucial component of key exchange

- Zillion applications (over-charged notion):
  - Key expansion, key extraction, key hierarchies
  - Key-exchange protocols, Hybrid encryption, Key wrapping, Physical RNGs, System PRNGs, Password-derived keys

- So what is it, really?

- Can we have a single scheme for all these uses?
Surprisingly Little Formal Work

- Research: Surprisingly little literature
- Practice: Plagued by multiple schemes, almost all ad-hoc, little or naïve rationale
- Dominated by hash-based schemes that treat hash as perfect function (“random oracle”)
- Needed: Widely accepted multi-purpose standard mechanism
The Challenge

- A practical but theoretically well-founded KDF scheme
  - But we do not even have definitions (or a full understanding of the extensive meaning/requirements of KDFs)
- Prudent use of hash functions: Minimize as much as possible assumptions on underlying hash scheme
  - Different uses → different requirements
- Single scheme, simple, efficient, hash-based
- Suitable for industry-wide standard
KDF: Two Main Functionalities

- **Key Extraction**: Derive a cryptographically strong key from an "imperfect source of key material"
  - Imperfect RNG, system entropy sources, Diffie-Hellman (KE), ...

- **Key Expansion**: Given a cryptographically strong key derive more keys

- Two fundamentally different functionalities

- Often mixed/confused in ad-hoc KDF schemes
  (a recipe for weaknesses and pitfalls)

  \[
  \text{Keys} = \text{Hash}(s \ || \ "1") \ || \ \text{Hash}(s \ || \ "2") \ || \ ... 
  \]
Example of Sources of Key Material

- A uniform random and secret master key (say, 256 bits)
  - The key expansion case

- Imperfect physical RNG (random number generators)
  - e.g., bit 0 with ~0.45 probability

- Software PRNG
  - Entropy source: e.g. sampled events, user’s key strokes, etc
  - Attacker has partial knowledge, can even influence source, yet conditional entropy (attacker’s uncertainty) assumed to be significant

- A Diffie-Hellman value $g^{x+y}$ output by a key exchange
  - restricted/computational entropy
DH as a source of randomness

- Diffie-Hellman key exchange outputs $g^{xy}$ in a group $G$ from which one needs to “extract” a cryptographic key.
  - We treat $g^{xy}$ as a source of “imperfect randomness”
- DDH: $g^{xy}$ indistinguishable from random element in $G$
  - Example. $G$ over $\mathbb{Z}_p^*$ of order $q$, $|q|=256$, $|p|=2048$
    $\rightarrow g^{xy}$ has 256 bits of entropy “trapped” in a 2048 long number
  - Very non-uniform in $\mathbb{Z}_p^*$ but sufficient entropy (256-bit) to extract key
- Sufficient entropy? Statistical entropy of $g^{xy}$ is 0 (attacker knows $g^x, g^y$)
  But computationally (by DDH) attacker has no information on $g^{xy}$
  $\Rightarrow$ sufficient computational entropy for extracting a key

See [Gennaro-K-Rabin, Eurocrypt 2004]
The DH Example (cont.)

- What if DDH does not hold, or protocol does not guarantee indistinguishability from uniform?

- Can only rely on CDH: $g^{xy}$ hard to guess but not necessarily indistinguishable from uniform

  - Need to extract keys based on unpredictability of $g^{xy}$
  - Hard-core function as extractor (can use dedicated functions, e.g. lsb’s, or crypto hash functions under suitable assumptions)

- Other considerations: Independence of samples ($g^{xy}$ vs $g^{x(y+1)}$), (independence of samples an issue for all extractor applications)
Imperfect Source of Randomness
(source key material)

- Imperfect: non-uniform, partial knowledge by attacker
- But substantial *conditional entropy*, e.g. 160 bits, though not necessarily uniform
  - Entropy is *conditioned* on knowledge by attacker
  - **Entropy can be computational** (e.g. Diffie-Hellman)
    - Computational hardness as a source of randomness (uncertainty)
    - HILL entropy (indistinguishable from a high-entropy source, DDH)
    - Unpredictability entropy (one-wayness, e.g. CDH)
Source Entropy: min-entropy

- Large Shannon entropy of source not sufficient to guarantee close-to-uniform output
  - Can have a high-probability element in the source which implies a high-probability value in the output, i.e. far from uniform.

- Need \textit{min-entropy}: No input assigned too high probability
  - A probability distribution \(X\) has \textit{min-entropy} \(m\) if for all \(x\), \(\text{Prob}_X(x) \leq 2^{-m}\) (i.e. \(m = -\log_2\) of highest probability)

- In our applications, \textit{computational} \textit{min-entropy} suffices
  - Source is computationally indistinguishable from a distribution that has that amount of true \textit{min-entropy}
Module I: Key Extraction

- Key Extraction: Derive a cryptographically strong key from a given *source of keying material*
  - imperfect source but with *sufficient* min-entropy

- Process: Source --> Sample --> Extract --> Key
  - Output key used to bootstrap the key expansion stage
Module II: Key Expansion

- Given a first strong key derive more keys
  - $K \rightarrow K_1, K_2, K_3$ (e.g. keys for MAC, encryption, etc)
  - Requirement: pseudo-randomness (even given partial knowledge)
    (pseudorandom = computationally indistinguishable from uniform)
  - Standard implementation via PRG/PRF

- Usually additional “context parameter” (need for PRF)
  - For example: $K_i = \text{PRF}_K(i, \text{“context”})$
  - “context” could be a functionality (“mac”), a protocol name (“ssl”), a session or user identity, etc. (a.k.a. domain separation)
Extract-then-Expand

- Two well differentiated modules, for the two well differentiated functionalities

- Basis for design and analysis
  - modules are orthogonal and replaceable
  - can implement both with same underlying cryptographic primitive (hash functions or block ciphers)
  - HKDF: a specific hash-based design, uses HMAC for both

- First, we need some definitions
Formalizing KDFs

- KDF: A transformation from a (weak) source of keying material to a pseudorandom key. But
  - Attacker has full knowledge of source distribution and partial knowledge on specific sample
  - Attacker can influence output by choosing context information (e.g. user identities, nonces, etc.)

- I am skipping formal definitions for this class
  - See next hidden slides and HKDF paper
Extract-then-Expand

“Extract-then-expand” paradigm

\[ K_{prf} = \text{Extract}(\text{salt}*, \text{skm}) \quad \text{skm} = \text{source key material} \]

\[ \text{Keys} = \text{Expand}(K_{prf}, \text{Keys-length, ctxt\_info}) \]

- salt: practice jargon for “a random non-secret quantity”; in our setting it works as an extractor seed (\(\rightarrow\) strong extractor)
Instantiating Extract-then-Expand

- Expand: Just a PRF (with variable input/output length)
- Extract: (strong) randomness extractors

Limitations of info-theoretic/combinatorial extractors

- practical schemes require large salt (~ |input|)
- entropy loss* (e.g. 256-bit DH → 160-bit SHA: security of $2^{-48}$)
- unsuited for extraction-from-unpredictability (e.g. only CDH) or deterministic extraction (“hard-core functions”)
- some crypto scheme proven only with RO-derived keys
- cases where independence of samples is not ensured
Idea: Use a PRF for both Expand and Extract

- We need a PRF for expand, can we use it for extract?
- Replace PRF’s key with a random, but known, seed (salt)
  - Extract(salt, sample) = PRF_{salt}(sample)
- Unfortunately, a PRF w/ a known key has no guarantee
  - Counter-examples use artificial (PK-based) constructions
  - Maybe practical hash-based PRFs do work (somehow)?
  - HMAC: The standard hash-based PRF
- We’ll see: HMAC enjoys good extraction properties
  ⇒ HKDF
Merkle-Damgard Hash Functions

- Compression function

\[ \chi = 512 \text{ bits} \]

\[ K = 160 \text{ bits} \]

\[ f \]

\[ f_K(\chi) = 160 \text{ bits} \]
Merkle-Damgard Hash Functions

- **Compression function**
  \( \chi = 512 \text{ bits} \)

- **(Unkeyed) Merkle-Damgard iterated hash**

\[
\begin{align*}
F_K(X) &= \text{Keyed via IV} \\
\mathcal{H}(X) &= f \quad f_{K}(\chi) = 160 \text{ bits}
\end{align*}
\]
NMAC: PRF mode for Merkle-Damgard

- \( \text{NMAC}_{K_1,K_2}(x) = f_{K_2}(F_{K_1}(x)) \)

- \( f = \) comp. function, \( F = \) keyed M-D

- Provable PRF if compression function is PRF

- HMAC = Same with \( K_1, K_2 \) derived from a single \( K \) (and black box use of hash function)
HKDF: HMAC-based KDF
(HMAC as extractor and PRF)

\[ K_{prf} = \text{HMAC}(\text{salt}, \text{skm}) \quad \text{skm} = \text{source key material} \]

\[ \text{Keys} = \text{HMAC}^*(K_{prf}, \text{keys\_length}, \text{ctxt\_info}) \]

where \( \text{Keys} = K_1 \parallel K_2 \parallel \ldots \)

\[ K_{i+1} = \text{HMAC}(K_{prf}, K_i \parallel \text{ctxt\_info} \parallel i) \]

Feedback mode

Note use of a PRF with salt, a random but non-secret "key"

(sometimes we’ll set salt = 0)
HKDF: HMAC-based KDF
(HMAC as extractor and PRF)

$K_{prf} = HMAC(salt, skm)$  $skm=$ source key material

Keys = HMAC*(K_{prf}, keys_length, ctxt_info)

where Keys = $K_1 || K_2 || \ldots$

$K_{i+1} = HMAC(K_{prf}, K_i || ctxt\_info || i)$

Feedback mode

Note use of a PRF with salt, a random but non-secret “key”
(sometimes we’ll set salt = 0)
Properties of HMAC to support HKDF

- Results that back HMAC in a variety of relevant applications:
  - Single function (hash, random oracle)
  - Family of functions with secret or public keys
  - Functionalities: PRF, extractor, random oracle, collision resistance

- Results in the form of: *If compression function has property A then HMAC has property A’*
  - Examples: PRF, delta-AU, extractor, RO
  - Note: NMAC vs HMAC
PRF and RO-based results

- If compression function \( f \) is PRF then NMAC is a PRF

- If \( f \) is a RO family then HMAC is indifferentiable from RO ("indifferentiable" = indistinguishability for ideal objects)

- Corollary: If \( f \) is RO, HMAC is a good extractor and a good hard-core (on distributions that are independent from \( f \))
  - Useful in restricted cases: CDH-only, small gap, no salt, ...

- \( f(H_K(x)) \) is a good extractor if \( f \) is RO and \( H_K \) is \( \delta \)-AU
  - \( \delta \)-AU is implied by collision resistance (design goal for hash f’n)
Non-idealized Assumptions

- If \( \{f_k\} \) is a good extractor family and also a PRF then NMAC is a good k-bit extractor on any distribution with blockwise entropy k.
  - Application to IKE/DH with safe primes

- If \( \{f_k\} \) is strongly universal and \( \{H_k\} \) is coll. resistant against linear-size circuits, then NMAC truncated by c bits is \( (n^{2-c/2}) \)-statistically close to unif.
  - Application: HKDF with SHA-512 for extraction, SHA-256 for PRF → 128-bit security under very mild assumptions
(versatile) application of HKDF

- **IKE (IPsec Key Exchange)**
  - $SK = HKDF(\text{nonces}, g^{xy})$ - (nonces exch’d and auth’d during KE)
  - Dual use of HKDF:
    - cleartext nonces $\rightarrow$ HKDF as extractor (nonces = salt)
    - Secret nonces $\rightarrow$ HKDF as PRF (PKE mode of IKE)

- **TLS 1.3 with shared key $K$ (e.g. resumption)**
  - $SK = HKDF(K, g^{xy})$
  - If $K$ revealed, $K$ acts as salt and HKDF as extractor (PFS)
  - If $K$ secret and $g^{xy}$ revealed, HKDF acts as PRF.
Application Example (OPTLS KDF)

- $C \quad g^x \quad S \ (s, g^s)$

- $g^y, \text{cert}(g^s), \text{MAC}_{K_m}(g^x, g^y)$

- $SK \leftarrow \text{derived from } g^{xs} \text{ (static) and } g^{xy} \text{ (ephemeral/PFS) via HKDF}$
  - $K_{xs} = \text{HKDF}(0, g^{xs})$
  - $K_{xy} = \text{HKDF}(0, g^{xy})$

- $SK = \text{HKDF}(K_{xs}, K_{xy})$: Secure as long as one of $g^{xs}, g^{xy}$ not exposed
Application Example (OPTLS KDF)

\[ C \rightarrow g^x \rightarrow S (s, g^s) \]

\[ g^y, \operatorname{cert}(g^s), \operatorname{MAC}_{K_m}(g^x, g^y) \]

- SK ← derived from \( g^{xs} \) (static) and \( g^{xy} \) (ephemeral/PFS) via HKDF
  - \( K_{xs} = \text{HKDF}(0, g^{xs}) \): Implements \( \text{RO}(g^{xs}) \) for CCA security (\( \sim \)DHIES)
  - \( K_{xy} = \text{HKDF}(0, g^{xy}) \): Implements \( \text{Extract}(g^{xy}) \) with salt=0
- SK = \( \text{HKDF}(K_{xs}, K_{xy}) \): Secure as long as one of \( g^{xs}, g^{xy} \) not exposed
  - If \( g^{xs} \) not compromised then \( \text{HKDF}(K_{xs}, ...) \) a PRF
  - If \( g^{xs} \) eventually compromised (the forward secrecy case) then \( \text{HKDF}(K_{xs}, ...) \) works as extractor w/ random but public salt \( K_{xs} \)
    - \( K_{xs} \) was generated by honest parties, hence uniform

Application Example (OPTLS KDF)

\[ C \rightarrow g^x \rightarrow S (s, g^s) \]

\[ g^y, \operatorname{cert}(g^s), \operatorname{MAC}_{K_m}(g^x, g^y) \]
Note: Why salt=0 in $K_{xy}$ and $K_{xs}$?

- Because we don’t have authenticated randomness to use as extractor seed.

- Unauthenticated seed can be chosen by attacker and break source-seed independence or chosen as “weak seed” (e.g. DRST’13)
  - Contrast IKE where salt = (nonce$_A$, nonce$_B$) which are signed before use.
    - Note: KE guarantees security of a key only with honest peer.
Example (TLS 1.3 Resumption)

- \( K_{\text{res}} \leftarrow \text{derived from } K_{\text{res}} \) (static) and \( g^{xy} \) (ephemeral/PFS) via HKDF
  - \( K_{xs} = \text{HKDF}(0, K_{\text{res}}) \): Implements \( \text{RO}(K_{\text{res}}) \) if \( K_{\text{res}} \) is low entropy, e.g. pwd
  - \( K_{xy} = \text{HKDF}(0, g^{xy}) \): Implements \( \text{Extract}(g^{xy}) \) with salt=0
- \( SK = \text{HKDF}(K_{\text{res}}, K_{xy}) \): Secure as long as one of \( g^{xs}, g^{xy} \) not exposed
  - If \( K_{\text{res}} \) not compromised then \( \text{HKDF}(K_{\text{res}}, ...) \) a PRF
  - If \( K_{\text{res}} \) eventually compromised (the forward secrecy case) then \( \text{HKDF}(K_{\text{res}}, ...) \) works as extractor w/ random but public salt \( K_{\text{res}} \)
    - \( K_{\text{res}} \) was generated by honest parties, hence uniform
HKDF as Collision Resistant

- TLS 1.3: Simultaneous RO, PRF, Extractor,… CRHF

- Use case: Binding resumption key to original HS session

  - bind(C,S, session-id), Mac_{Km}(bind(...), ...)
  - bind can be CRHF(C, S, session-id) but allows traceability
  - Instead: \( K_{\text{bind}} = \text{HKDF}(g^{xy}, C, S, \text{session-id}) \) at orig session
  - During resumption use \( K_{\text{bind}} \) as a key to create a one-time bind value \( \text{MAC}_{K_{\text{bind}}} (...) \)

- Crucial point: Derivation of \( K_{\text{bind}} \) requires CR key deriv.
  - Another HKDF goodie (derives from underlying hash)
Standards and Deployments

- Becoming the industry-wide standard for KDF
- IETF (RFC 5869): Already 18 RFC’s use it + many internet drafts (incl. TLS 1.3)
- NIST: NIST SP 800-56C (Recommendation for Key Derivation through Extraction-then-Expansion)
- Industry implementations: TLS 1.3, Google QUIC, WhatsApp, Facebook Messenger, "Snowden's" Signal, ...

- Bonus: “extract” made it into IETF jargon/notion...
Theory and Practice

- **Theory:** understanding requirements, formalizing, weaknesses in existing solutions, generalization, design, analysis, minimize RO

- **Practice:** Engineering considerations, minimize compromise, conservative design
  - minimize RO, “bad adviser”

- **Combination:** Proof-driven design®