*This paper was originally published in: 16th USENIX Security Symposium—SECURITY 2007.*
*This version is available from the author at:* `http://www.cs.stanford.edu/~xb/security07/`.
*Date of preparation: November 11, 2007.*

# Halting Password Puzzles
## Hard-to-break Encryption from Human-memorable Keys

XAVIER BOYEN — `xb@boyen.org` | Voltage Security

**Abstract**

We revisit the venerable question of "pure password"-based key derivation and encryption, and expose security weaknesses in current implementations that stem from structural flaws in Key Derivation Functions (KDF). We advocate a fresh redesign, named Halting KDF (HKDF), which we thoroughly motivate on these grounds:

1. By letting password owners choose the hash iteration count, we gain operational flexibility and eliminate the rapid obsolescence faced by many existing schemes.
2. By throwing a Halting-Problem wrench in the works of guessing that iteration count, we widen the security gap with any attacker to its theoretical optimum.
3. By parallelizing the key derivation, we let legitimate users exploit all the computational power they can muster, which in turn further raises the bar for attackers.

HKDFs are practical and universal: they work with any password, any hardware, and a minor change to the user interface. As a demonstration, we offer real-world implementations for the TrueCrypt and GnuPG packages, and discuss their security benefits in concrete terms.

# 1 Introduction

For a variety of reasons, it is becoming increasingly desirable for the denizens of *transparent societies* to attend to a *last bastion of privacy*: a stronghold defended by secret-key cryptography, and whose key exists but in its guardian's mind. To this end, we study how *"pure password"-based encryption* can best withstand the most dedicated offline dictionary attacker—regardless of password strength.

## 1.1 Human-memorable Secrets

**Passwords.** Passwords in computer security are the purest form of secrets that can be kept in human memory, independently of applications and infrastructures. They can be typed quickly and discreetly on a variety of devices, and remain effective in constrained environments with basic input and no output capabilities. Not surprisingly, passwords and passphrases have become the method of choice for human authentication and mental secret safekeeping, whether locally or remotely, in an online or offline setting.

Passwords have the added benefit to work on diminutive portable keypads that never leave the user's control, guaranteeing that the secret will not be intercepted by a compromised terminal. User-owned and password-activated commercial devices include the *DigiPass* [12] for authorizing bank transactions, the *CryptoCard* [10] for generating access tokens, and the ubiquitous cellular phone which can be used for making payments via SMS over the GSM network.

Nevertheless, the widespread use of passwords for securing computer systems is often deplored by system administrators, due to their low entropy and a propensity to being forgotten unless written down, which in turn leads to onerous policies that users deem too difficult to follow [43]. In this work, by contast, we seek not to change people's habits in significant ways; rather, our goal is to maximize security for passwords that are actually used, no matter how weak these might be.

**Alternatives.** A number of alternatives have been suggested to alleviate the limitations of passwords, including inkblots [39], visual recognition [29], client-side puzzles [21, 11], interactive challenges [32], word labyrinths [6], but any of them has yet to gain much traction.

Multi-factor authentication systems seek not to replace passwords, but supplement them with a second or third form of authentication, which could be a physical token (*e.g.*, SecurID [38]) or a biometric reading. These approaches are mostly effective in large organizations.

Compelling as these sophisticated proposals may be, multi-factor authentication is no panacea, and the various mental alternatives to passwords tend to be slow, complex, and error-prone, and depend on a particular medium or infrastructure. For instance, mental puzzles typically require multiple rounds of interaction to gather enough entropy, whereas image recognition tasks will never work without a display; with either approach, simple portable keypads are pretty much out of the question. The usual criticisms that have been levelled at passwords, such as low entropy and poor cognitive retention, apply to these alternatives as well.

## 1.2 Application Contexts

**Online Uses.** In the online setting, the main use of passwords is for remote user authentication. Password-based Encrypted Key Exchange (EKE) [5] and Authenticated Key Exchange (PAKE) [17] protocols enable high-entropy session keys to be established between two or more parties that hold a low-entropy shared secret, ideally with mutual authentication. The threat model is the online attack, conducted by an opponent who can observe and corrupt the lines of communication, and sometimes also the transient state of a subset of the participants, but without access to the long-term storage where the password data are kept.

What makes the online setting favorable for password-based authentication, is that participants can detect (in zero knowledge) when an incorrect password is used, and terminate the protocol without leaking information. The attacker can always run a fresh instance of the protocol for every candidate password, but many EKE and PAKE protocols [20, 4, 8] achieve theoretically optimal security by ensuring that no adversary can do better than this. Online guessing is easy to detect in practice, and can be defeated by locking out accounts with repeated failures. Dealing with passwords in the pure online setting is in that respect a mostly solved problem, and is the topic of the ongoing IEEE 1363.2 standardization effort [19]. We will not discuss online passwords further.

**Offline Uses.** In the offline setting, passwords are mainly used for login and to encrypt data at rest in local storage. Typical applications of password-based encryption range from user-level encryption of PGP or S/MIME private keys, to kernel-level enforcement of access permissions, to hardware-level encryption of a laptop's hard disk by a security chip or by the drive itself.

Despite their limitations, passwords tend to be preferable to other types of credentials. Physical tokens able to store large cryptographic keys are susceptible to theft along with the laptop they are supposed to protect. Biometrics are inherently noisy and must trade security for reliability; they are also tied to a specific user and cannot be revoked. Visual and other alternatives to passwords are often complex and too demanding for low-level operation or in embedded systems; at any rate they do not have clear security benefits over passwords.

The main threat faced by password-based encryption is the offline dictionary attack. Unlike the online guessing discussed earlier, in an offline attack the adversary has access to the complete ciphertext and all relevant information kept in storage—except the password—and does not need the cooperation of remote parties to carry out the attack. Tamper-resistant hardware may complicate ciphertext acquisition, but, past that point, the adversary is bound only by sheer computational power: this is what makes low-entropy passwords so much more damaging offline than online.

## 1.3 Password-based Encryption

Aside from the peril of dictionary attacks, passwords are not usable natively as encryption keys, because they are not properly distributed. Key Derivation Functions (KDF) let us solve this.

**Key Derivation.** The goal is to create a uniform and reproducible key from a password. The universally accepted practice is to mangle the password through a hash function a number of times, after blending it with random data called *salt* that is made public. The many hash iterations serve to make offline dictionary attacks slower, and the salt is to preclude using lookup tables as a shortcut [18, 30, 3]. Virtually all KDFs follow this model; however, it is not a panacea.

For ones, referring to the apparent futility of suppressing (targeted) offline dictionary attacks, in the full version of their recent CRYPTO '06 paper, Canetti, Halevi, and Steiner [9] lament:

> [...] *typical applications use a key-derivation-function such as* SHA1 *repeated a few thousand times to derive the key from the password, in the hope of slowing down off-line dictionary attacks.* [...] *Although helpful, this approach is limited, as it entails an eternal cat-and-mouse chase where the number of iterations of* SHA1 *continuously increases to match the increasing computing powers of potential attackers.*

Instead, these authors propose to treat the password as a path in a maze of CAPTCHAs [42], whose (secret) answers will provide the key. Alas, such augmented-password schemes tend to be unwieldy; here, gigabytes of CAPTCHAs must be pre-generated and then retrieved in secret, which relegates them to local storage, lest a dictionary attack on the remote access pattern betray the password.

In general, while it is true that secrets with visual or interactive components are likely to hamper mechanical enumeration, old-fashioned passwords will remain faster, less conspicuous, and much more convenient for humans to handle and recall. Still, the problem remains to design a good KDF.

**Iteration Count.** To perceive the difficulty of KDF design, recall that Unix' `crypt()` hashing for `/etc/passwd` back in the seventies took a quarter of a second [33] to perform two dozen iterations of the DES cipher (with salt). The original PKCS#5 key derivation standard from the early nineties [37] was content to use a "positive number" of applications of MD2 or MD5, but has since been updated [22] to recommend "at least 1000" iterations of MD5 or SHA1. This recommendation has been followed in the recent and well-regarded *TrueCrypt* software [40], albeit perhaps under the wire, with merely 2000 iterations of SHA1 or RipeMD160, or 1000 iterations of WhirlPool.

An endemic problem with KDF implementations is that these numbers tend to be set in stone without user override. For example, the custom "s2k" (string-to-key) function of *GnuPG* [15] will hash 65536 bytes of password-derived data, which amounts to a few thousand iterations of SHA1. This number is frozen in the program, which is a missed opportunity, because it is also recorded in the OpenPGP [7] ciphertext, and could therefore be varied. Indeed, the KDF in *GnuPG* can be recompiled to hash up to 65011712 bytes of data, without losing decryption compatibility with the official version. Sadly, even that ostensibly large number appears pathetic by today's standards, as it takes only two seconds to digest those 65 million bytes on a 1.5 GHz laptop *circa* 2005.

## 1.4 The Problem, and Our Solution

The balancing act in KDF design is to choose a large enough iteration count to frustrate a dictionary attack, but not so large as to inconvenience the user. Any choice made today is likely to prove wholly inadequate a few years from now. Furthermore, this assessment should be made in view of the lifespan and sensitivity of the plaintext, as well as the estimated strength of the password—two crucial tidbits of which only the actual user (and not the system designer) is privy.

**Security Maximization and User Programmability.** Given the constraints, the primary goal is to maximize—by technical means—the "gap" between user inconvenience and the costs inflicted on attackers. Secondarily, it is crucial—for policy and deeper reasons—that users be free to vary the (secret) level of inconvenience they are willing to accept on a case-by-case basis. In essence, we:

*(i) let the user choose the amount of work he or she deems appropriate for the task,*
*(ii) keep the choice secret from attackers (and allow the user to forget it too),*
*(iii) and ensure that all user-side computing power can be exploited.*

We emphasize again that human-selected passwords tend to be by far the weakest link in a typical cryptographic chain [26], which is why we seek to squeeze as much security from them as we can.

**"Halting" Key Derivation Functions.** HKDFs are the practical embodiment of all the above requirements. They consist of two algorithms, Prepare and Extract. The principle is as follows:

- To create a random encryption key, the user launches a randomized algorithm HKDF.Prepare on the password, lets it crunch for a while, and interrupts it manually using the user interface, to obtain an encryption key along with some public string to be stored with the ciphertext.
- To recover the same key subsequently, the user applies a deterministic algorithm HKDF.Extract on the password and the public string from the first phase. The algorithm halts spontaneously when it recognizes that it has recovered the correct key, barring which it can be reset manually.

Thus, if the user entered the correct password, HKDF.Extract will halt and output the correct key after roughly the same amount of time as the user had let HKDF.Prepare run in the setup phase. However, if the user entered a wrong password, at some point he or she will find that it is taking too long and will have the option to stop the process manually in order to try again.

Notice that the public string causes the derived key to be a randomized function of the password, and thus also plays the role of "salt". HKDFs can be used as drop-in substitutes for regular KDFs, pending addition in the user interface of a button for interrupting the computation in progress.

**HKDF Ramifications.** The above idea is as simple as it is powerful, though surprisingly it has not been investigated or implemented before. Ramifications are deep, however:

1. **(Stronger crypto)** Two extra bits of security can be reclaimed *from any password.*

   A paradoxical result that we prove in this paper is that, if the attacker does not know the iteration count, and is then compelled to use a "dovetail" search strategy with many restarts, then the attack effort is multiplied by $\sim 4\times$ (a 2-bit security gain), at no cost to the user.

   Intuitively, our design will force any game-theoretic optimal brute-force attacker to overshoot the true iteration count when trying out wrong passwords. By contrast, when the user enters the correct password, the key derivation process will be halted as soon as the programmed number of iterations is reached (using some mechanism for detecting that this is the case).

2. **(Flexible policies)** Long-term memorable passwords for key recovery become a possibility.

   Sophisticated users should be able to choose any password that they will remember in the long term, even with low entropy, as long as they are used with a large enough iteration count to keep brute-force attackers at bay (at the cost of slowing down legitimate uses correspondingly).

   This opens the possibility of using multiple passwords of reciprocal strength and memorability: one high-entropy password with a small iteration count for fast everyday use; and a second, much more memorable password for the long term, protected by a very large iteration count, to be used as a backup if the primary password is forgotten.

3. **(Future proofing)** Password holders automatically keep pace with password crackers.

   Indeed, if every time a user's password is changed, the iteration count is selected to take some given amount of time on the user's machine, then the iteration count will automatically increase with any hardware speed improvement. This will negate all advantage that a brute-force attacker might gain from computers becoming faster, if we make the natural assumption that technological progress benefits password verifiers at the same rate as password crackers.

4. **(Resource maximization)** User-side parallelism is exploited to raise the cost of attacks.

   Users care about (real) elapsed time; attackers about cumulative CPU time. Independently of the idea of hiding the iteration count, we design the key derivation to be parallelizable even for a single key. With the popularization of multi-core PCs, users will then be able to increase the total cost of key derivation without increasing the observed elapsed time that matters to them. The heightened total cost is however borne in full by the adversary, who gains nothing by parallelizing "within" single passwords, as opposed to "across" several ones.

These benefits are complementary rather than independent: for example, by accentuating the iteration unpredictability, Properties 2 and 3 solidify the Property 1 security gains that ride on it. Property 4 is orthogonal, but is equally crucial to our goal of making attacks maximally expensive.

**User Acceptance.** Observed routine behavior and common practices suggest that user acceptance should be easy. Indications abound that deliberately sluggish password interfaces are the norm that everyone has come to expect. The main commercial operating systems even use login screens that frustrate casual password guessing with naive hacks such as fake delays; and, although this charade provides but illusory protection against true offline attacks, it eloquently demonstrates that users (or system provisioners) *demand* that those who try out bad passwords be punished.

HKDFs fulfil these expectations in a cryptographically sound way; but, in stark contrast to those commercial approaches, HKDFs side with the users, and seek to empower rather than burden them. *E.g.*, the special UI control to "stop and retry" should be second nature to anyone accustomed to "reloading" a stalled web page in a browser; power users and fast typists should find it gratifying.

## 1.5 Related Work

The first deliberate use of expensive cryptographic operations to slow down brute-force attacks, in the `crypt()` password hashing function on Unix systems, coincides with the public availability of the DES cipher. Since then, a lot of progress has been made.

A parameterizable alternative called `bcrypt()` [33] was proposed to avoid the obsolescence problems associated with fixed iteration counts; unlike in our proposal, the cost parameter was preset by the system administrator, and was stored locally in the clear. For client/server authentication, it has been suggested [16] to slow down key derivation the first time, and make it subsequently faster by caching some state on the client, in an attempt to impede online trial-and-error attacks without becoming completely vulnerably to client-side exposure. Similarly, several online PAKE protocols have taken steps to thwart offline dictionary attacks on compromised servers, either by keeping user passwords in encrypted form, or by distributing them among several locations; *cf.* [27] and the references therein. Yet other approaches to password management have sought to prevent dictionary attacks in mixed offline/online contexts: PwdHash [36] is a browser plug-in that tailors a master password into unique passwords for different web sites; PassPet [44] is a comparable plug-in that also lets the user choose the iteration count with visual feedback on the estimated strength; here, the selected iteration count is stored on a dedicated remote server, and retrieved as needed.

Deliberately expensive cryptography has also been applied in "proof-of-work" schemes for combatting junk email [14] as well as for carrying out micro-payments [2], among other similar applications. These CPU-bound constructions are based on easy-to-verify but hard(er)-to-compute answers to random challenges built from hash functions. Memory-bound proof-of-work schemes have also been proposed [13], motivated not by the desire to prevent parallelism, but rather by the observation that memory chips have narrower and more predictable speed ranges than CPUs. At the other extreme of this spectrum, time-lock puzzles [35] are encryption schemes designed to be decryptable, without a key, after a well-defined but very long computation; these schemes are based on algebraic techniques, and view the publicity of the decryption delay as a feature [28].

Regarding parallel hashing schemes, we mention Split MAC [41], which is a parallelizable version of HMAC [24] for hashing long messages (but not suitable for hashing a short password repeatedly). On the cryptanalytic side, we mention the classic time/space trade-off analysis [18] of unsalted deterministic password hashing, and its modern reincarnation known as the "rainbow-table" attack [30]. See also [3] for a theoretical study of these types of algorithms.

**Contribution.** The point of this paper is as much to study HKDFs for their own sake as a new cryptographic and security tool, as it is to advocate their deployment in all practical systems that do password-based encryption.

In Section 2 we define HKDFs, construct them generically, and prove their basic security. We also parameterize them for the long term, and discuss user-side parallelism. In Section 3 we adopt a theoretical stance and study the origin of the $\sim 4\times$ security factor that seems to arise magically.

In Section 4 we put on a systems hat and show how to integrate HKDFs in popular software such as *TrueCrypt* and *GnuPG*. We plan to release our implementations as open-source C code.

## 2 HKDF Design

The guiding design principles of Halting Key Derivation Functions are the following:

1. the cost of key derivation is programmed by the user and has no prior upper bound;
2. the amount of work for each key is independent and secret;
3. the key derivation memory footprint grows in lockstep with computation time;
4. the computation for deriving a single key can be distributed if needed.

We have already mentioned the motivation for (1.) letting the user program the iteration count $t$ arbitrarily, and (4.) providing user-side parallelism. The justification for (2.) keeping $t$ a secret, and (3.) having the memory footprint grow linearly with $t$, are to force the attacker to make costly guesses as it tries out wrong candidate passwords from its dictionary $D$.

Suppose the adversary is certain the true password $w$ belongs in $D$, but has no idea about $t$. The obvious approach is to try out all the words in $D$, in parallel, for as many iterations as needed. However, this attack is incredibly memory-consuming since for each word there is state to be kept: terabytes or more for mere 40-bit entropy passwords ($\#D = 2^{40}$).

If the attacker cannot maintain state across all of $D$ as the iteration count is increased, the only alternative is to fix an upper bound $\bar{t}$ for $t$ and try each word for $\bar{t}$ iterations, and then start over with a bigger $\bar{t}$. Clearly, this is more expensive since much of the computation is being redone. How much more expensive depends on the schedule for increasing $\bar{t}$. Increase it too slowly, *e.g.*, $\bar{t} = 1, 2, 3, ...$, and most of the work ends up being redone. Increase it too fast, *e.g.*, $\bar{t} = 1!, 2!, 3!, ...$, and the true value of $t$ risks being overshot by a wide margin. Either way, work will be wasted.

We shall see that with the optimal strategy the attacker can keep the cost as low as $\sim 4\times$ as much as if $t$ has been public. The user does not pay this penalty since on the correct password the

HKDF halts spontaneously at the correct iteration count $t$ (which the user need not recall either). This gives us $\sim 2$ bits of extra security essentially *for free*.

The memory footprint growth in $\Theta(t)$ is a technicality to ensure that this conclusion holds for arbitrarily large $t$, lest there exist a threshold value of $t$, no matter how enormous, beyond which it became more economical to purchase gigantic memory banks and conduct a persistent attack.

## 2.1 Formal Specification

As briefly outlined in Section 1.4, an HKDF suite consists of a pair of functions:

**Prepare** : $(r, t, w) \mapsto v$ which, given a random seed $r$, an iteration count $t$, and a password $w$, produces a publishable verification string $v$ (and also, if ordered, a deterministic key $k$);

**Extract** : $(v, w) \mapsto k$ which, given a verification string $v$, and a password $w$, deterministically produces, either a key $k$ upon halting, or a silent failure by not halting in polynomial time.

In this abstraction, Prepare is fed an iteration count $t$ at the onset; in practice, the user can fix the parameter $t$ by interrupting the computation as she pleases, *"by feel"*, with the proper interface.

**Security Model.** We write $[a]$ and $[a \mid b]$ to denote marginal and conditional distributions of random variables. Let $\mathcal{U}_S$ denote the uniform distribution over a set $S$, often implicit from context.

Pick $r \in_\$ \{0,1\}^\ell$, *viz.*, so that $[r] \equiv \mathcal{U}_{\{0,1\}^\ell}$, to be our $\ell$-bit random seed for some parameter $\ell$. We first demand that the extracted keys be uniform and statistically independent of the secrets:

– Key uniformity: $[k \mid t, w] \equiv \mathcal{U}$ where $k = \mathsf{Extract}(\mathsf{Prepare}(r, t, w), w)$.

We also impose lower and upper computational complexity bounds on the functions:

– Preparation complexity: $\mathsf{Prepare}(r, t, w)$ always halts in time $\mathrm{O}(t)$, for all inputs.

– Extraction complexity: $\mathsf{Extract}(v, w)$ requires time and space $\Theta(t)$, for all $v = \mathsf{Prepare}(r, t, w)$.

– Conditional halting: $\mathsf{Extract}(v, w')$ does not halt in polynomial time when $w' \neq w$.

We then ask that the key be unknowable without the requisite effort, even with all the data:

– Bounded indistinguishability: $[v, k \mid t, w] \overset{\mathrm{o}(t)}{\equiv} \mathcal{U}$ for $v = \mathsf{Prepare}(r, t, w)$ and $k = \mathsf{Extract}(v, w)$.

*I.e.*, for any randomized algorithm running in space (and hence time) strictly sub-linear in $t$, the joint $[v, k]$ is computationally indistinguishable from random even given $t$ and/or $w$.

As a consequence of the latter, the public string $v$ is computationally indistinguishable from random to anyone who has not also guessed (and tested for $t$ iterations) the correct password against it.

To summarize, for random $r$, it must be infeasible to find, in polynomial time in the security parameter, a tuple $(k, t, w, w')$ such that $k = \mathsf{Extract}(\mathsf{Prepare}(r, t, w), w')$ and $w \neq w'$. Furthermore, finding a tuple $(k, t, w)$ such that $k = \mathsf{Extract}(\mathsf{Prepare}(r, t, w), w)$ must require $\Theta(t)$ units of time and memory, barring which no information about the correct $k$ must be obtained from $r, t, w$.

## 2.2 Generic Construction

There are many ways to realize HKDFs, depending on the computational assumptions we make. One of the simplest constructions is generic and is based on some cryptographic hash function $H : \{0,1\}^{2\ell} \to \{0,1\}^\ell$ viewed as a random oracle, for a security parameter $\ell$.

To capture the main idea, we start with a sequential HKDF construction. The construction is:

**HKDF$^H$.Prepare**$(r, t, w)$

Inputs: random seed $r$, iteration count $t$ (can be implicit from user interrupt), password $w$.
Output: verification string $v$, and optionally, derived key $k$.

```
z ← H(w, r)                              /* init z from password and seed */
FOR i := 1, ..., t   or until interrupted
    y_i ← z                              /* copy z to new array element y_i */
    REPEAT q times
        j ← 1 + (z mod i)                /* map z to some j ∈ {1, ..., i} */
        z ← H(z, y_j)                    /* update z using y_j */
v ← (r, H(y_1, z))                       /* output seed r and test hash H(y_1, z) */
k ← H(z, r)
```

**HKDF$^H$.Extract**$(v, w)$

Inputs: verification string $v$ (composed of seed and test substrings $r$ and $h$), password $w$.
Output: derived key $k$, or no output for failure to halt.

```
parse v as (r, h)                        /* initialization and termination data */
z ← H(w, r)
FOR i := 1, ..., ∞                       /* unbounded loop */
    y_i ← z
    REPEAT q times
        j ← 1 + (z mod i)
        z ← H(z, y_j)
    IF H(y_1, z) = h THEN BREAK          /* halting condition */
k ← H(z, r)                              /* derived key */
```

The constant $q$ is a parameter that determines the ratio between the time and space requirements. Since the Extract function may not halt spontaneously, it must be resettable by the user interface.

## 2.3 Security Properties

It is easy to see that the key output by Prepare is random and correctly reproducible by Extract. As for the HKDF security properties, we state the following lemmas.

**Lemma 1.** *Key uniformity:* $[k \mid t, w] \equiv \mathcal{U}_{\{0,1\}^\ell}$ where $k = \mathsf{Extract}(\mathsf{Prepare}(r, t, w), w)$.

**Lemma 2.** *Preparation complexity:* $\mathsf{Prepare}(r, t, w)$ halts in time $\Theta(qt)$ on all inputs, for fixed $q$.

**Lemma 3.** *Extraction complexity:* $\mathsf{Extract}(v, w)$ halts in time $\Theta(qt)$ and uses $\Theta(t)$ bits of memory, for any $v = \mathsf{Prepare}(r, t, w)$ with same $w$.

*Proofs.* Since $r$ is random and $H$ is a random function, $k = H(z, r)$ is uniformly distributed $\forall z$, which establishes Lemma 1. Lemmas 2 and 3 follow by inspection of the algorithms. □

**Lemma 4.** *Conditional halting:* Except with negligible probability, $\mathsf{Extract}(v, w')$ halts in super-polynomial time $\Omega(2^\ell q)$ for any $v = \mathsf{Prepare}(r, t, w)$ and $w' \neq w$, where the probability is taken over the random choice of $H$ for arbitrary inputs.

*Proof.* For $w' \neq w$, the value of $y_1 = H(w, r)$ in $\mathsf{Prepare}$ and $y'_1 = H(w', r)$ in $\mathsf{Extract}$ will be statistically independent since $H$ is a random function, and therefore so will be the benchmark $h = H(y_1, z)$ and its comparison value $H(y'_1, z')$ for all $z'$. Since the constant $h$, the variable $z'$, and the value $H(y'_1, z')$, are all $\ell$-bit binary strings, we find that the probability of halting on the wrong password in sub-exponential time $i < 2^{\mathrm{o}(\ell)}$ is negligible. *In extenso*, letting $\ell \to \infty$,

$$\Pr(\mathsf{Extract} \text{ loops indefinitely}) = e^{-1} \approx 0.3678794 \ ,$$
$$\Pr(\mathsf{Extract} \text{ halts before count } i) = (1 - e^{-1})(1 - e^{-i/2^{\ell}}) \ . \qquad \square$$

**Lemma 5.** *Bounded indistinguishability:* the distributions $[v, k \mid t, w]$ and $\mathcal{U}_{\{0,1\}^{3\ell}}$ are perfectly indistinguishable by any algorithm running in sub-linear time and/or space $\mathrm{o}(t)$ in the iteration count, for any $v = \mathsf{Prepare}(r, t, w)$ and $k = \mathsf{Extract}(v, w)$.

*Informal proof sketch.* We deal with the time-bound indistinguishability claim first. Observe that both $v$ and $k$ are independent ouputs of a chain of $qt$ applications of $H$, seeded by $r$. Since $H$ is a random oracle, a standard argument shows that no information about $(v, k)$ can be obtained without $qt$ queries to $H$, which establishes the time-bound indistinguishability claim.

For the stronger space-bound indistinguishability claim, a more subtle argument shows that, with overwhelming probability, all possible computation paths require that "almost all" $y_i$ for $i = 1, ..., t$ be stored in memory. The argument is based on the following sequence of observations: (1) For all $i \in \{1, ..., t\}$ and all $i' \in \{i, ..., t\}$, the value $y_i$ computed at step $i$ will be needed at a subsequent step $i'$ with probability $\Pr(y_i \text{ needed at step } i') = q/i'$, independently of its prior uses. (2) The expected number of times that $y_i$ will be needed in the course of the entire computation is $\#\{i' : y_i \text{ needed at step } i'\} \doteq \sum_{i'=i+1}^{t} (q/i') \approx q \ln(t/i)$, which is $\geq nq$ for any $n > 0$ and $i \leq e^{-n} t$. (3) The probability that for fixed $i \leq e^{-n} t$ the value $y_i$ is never needed is $\Pr(y_i \text{ not needed}) \leq e^{-nq}$, which whenever $n > \ell/(q \ln 2)$ is a vanishingly small function of the effective security parameter $\ell$. (4) Since, for such $n$, the difference $e^{-n} t - e^{-n-1} t$ is a linear function of $t$, the sub-linear memory constraint requires that some $y_j$ with $j \leq e^{-n-1} t$ be dropped prior to reaching the $\lceil e^{-n} t \rceil$-th step. (5) With overwhelming probability $\Pr \geq 1 - e^{-nq}$, the dropped value $y_j$ appears in the computation path of some $y_i$ where $j < e^{-n} t < i$, and without the value of $y_j$ the key derivation cannot proceed.

The outcome of this reasoning is that before we can compute $y_i$, we need to recompute the dropped value $y_j$, which itself requires the recomputation of some earlier values still: some of these values must also have been dropped, as the same reasoning shows using an incremented $n \leftarrow n + 1$ (with recursion upper bound $\lfloor \ln t \rfloor$). To complete the argument, we note that for some $l$ where $j < l < i \leq t$, the recomputation of $y_j$ needed for $y_i$ will require freeing up some previously stored value $y_l$, which is still needed for the calculation of $y_i$, and whose recomputation will require $y_j$; when this happens, the algorithm will be stuck. This shows that the *intrinsic* space complexity of computing $\mathsf{HKDF}^H.\mathsf{Extract}$ by whatever means in the random oracle model is $\tilde{\Theta}(\ell t)$. $\square$

A consequence of Lemma 5 is that, unless the attacker has an enormous and linearly increasing amount of memory at its disposal, it will not be able to mount a "*persistent*" attack against all $D$ (or any significant fraction thereof). It will have to choose which bits of state must be kept, and which ones must be erased to make room for others: the attack will necessarily be "*forgetful*".

## 2.4 Leveraging Parallelism

In addition to allowing arbitrarily large $t$ and forcing the adversary to guess it, a complementary way to increase the adversary's workload is to exploit any parallelism that is available to the user. Indeed, users care about the *real elapsed time* for processing a *single password*, whereas attackers worry about the *total CPU time* needed to cycle through the *entire dictionary*. Hence, we can hurt the adversary by increasing the CPU-time/elapsed-time ratio, using a parallelizable key derivation.

We stress that this new notion, *password-level parallelism*, is safe, for it only benefits the user—despite being at odds with an old apocryphal wisdom on password hashing that abhors parallelism—as attackers would rather rely on *dictionary-level parallelism*, cruder but always available to them.

We propose two different (and combinable) ways of parallelizing HKDFs on the user's side.

**Pooled-Memory Multi-Core Device.** Let a "maximum parallelism" parameter $p$ numerate the virtual computing cores owned by the HKDF, all sharing read access to an ephemeral array of $y_i$. Let $\{\text{STATEMENT}(l)\}_{l=1,...,p}$ be shorthand for the $p$ independent $\text{STATEMENT}(1), ..., \text{STATEMENT}(p)$. Fix a cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^\ell$ with security parameter $\ell$. The scheme is:

$p$**HKDF**$^H$.**Prepare**$(r, t, w)$

Inputs: random seed $r$, iteration count $t$ (can be implicit from user interrupt), password $w$.
Output: verification string $v$, and optionally, derived key $k$.

$\{z_l \leftarrow H(w, r, l)\}_{l=1,...,p}$ /* init each $z_l$ independently */
$z \leftarrow H(z_1, ..., z_l)$ /* init $z$ from all the $z_l$ */
FOR $i := 1, ..., t$ or until interrupted
  $y_i \leftarrow z$ /* copy $z$ to new array element $y_i$ */
  REPEAT $q$ times
    $\{j_l \leftarrow 1 + (z_l \bmod i)\}_{l=1,...,p}$ /* map each $z_l$ to some $j_l \in \{1, ..., i\}$ */
    $\{z_l \leftarrow H(z_l, y_{j_l}, l)\}_{l=1,...,p}$ /* update each $z_l$ independently */
  $z \leftarrow H(z_1, ..., z_l)$ /* update $z$ globally */
$v \leftarrow (r, H(y_1, z))$
$k \leftarrow H(z, r)$

$p$**HKDF**$^H$.**Extract**$(v, w)$

Inputs: verification string $v$ (composed of seed and test substrings $r$ and $h$), password $w$.
Output: derived key $k$, or no output for failure to halt.

parse $v$ as $(r, h)$
$\{z_l \leftarrow H(w, r, l)\}_{l=1,...,p}$ /* $p$-way parallelizable */
$z \leftarrow H(z_1, ..., z_l)$
FOR $i := 1, ..., \infty$
  $y_i \leftarrow z$
  REPEAT $q$ times /* $p$-way parallelizable across whole loop */
    $\{j_l \leftarrow 1 + (z_l \bmod i)\}_{l=1,...,p}$ /* $p$-way parallelizable */
    $\{z_l \leftarrow H(z_l, y_{j_l}, l)\}_{l=1,...,p}$ /* $p$-way parallelizable */
  $z \leftarrow H(z_1, ..., z_l)$
  IF $H(y_1, z) = h$ THEN BREAK
$k \leftarrow H(z, r)$

Total computational cost is $\Theta(pqt)$ hash evaluations. Total memory requirement is $\Theta(p + t)$ hash values, including a constant $\ell p$ bits of extra memory overhead compared to the basic construction. Complexity-wise, the parameter $p$ acts as a multiplier on the time/space proportionality ratio $q$, so that all security properties are transposed with $pq$ instead of $q$. Even if sequential machines are envisioned at first, it is wise to enable parallelism by increasing $p$ and decreasing $q$ for a target $pq$. The relative penalty exerted on the adversary will be proportional to the number $N \in \{1, ..., p\}$ of CPUs that the user does bring to the computation; this may be far less than $p$, but, ideally, $N \mid p$.

**Partitioned-Memory Assembly.** Any sequential $\mathsf{HKDF}$ or parallelizable $p\mathsf{HKDF}$ can be further parallelized $2^l$-fold by omitting $l$ bits from $r$ in the public string $v = (r, h)$ emitted by $\mathsf{Prepare}$; to decrypt, the user tries all completions of $r$ at once by running $2^l$ instances of $\mathsf{Extract}$ until one halts.

Total work and total memory footprint are both $2^l$ times that of the $\mathsf{HKDF}$ or $p\mathsf{HKDF}$ scheme; but elapsed time will be unchanged if we spread the search (and memory usage) across $2^l$ machines. Compared with the $\Theta(p + t)$ "pooled-memory" construction for matching parallelism levels $p = 2^l$, storage requirements are much stiffer at $\Theta(2^l t)$, but can be distributed into $2^l$ $\Theta(t)$-sized banks—at least if we pretend that distributed hardware is a realistic requisition for password-entry devices.

## 2.5 Practical Parameters

HKDF parameter selection is non-critical and much easier than with regular KDFs, since we are not trying to make decisions for the user, or prevent obsolescence by betting *pro* or *con* Moore's law. The only choices we need to make concern the coefficients $p$ and $q$. The rule of thumb is: maximize $pq$ in view of today's machines, and then fix $p$ to cover all foreseeable needs for parallelism.

For the sake of illustration, let $\ell = 256$, and suppose that that the user's key derivation hardware can compute $n = 2^{25}$ hashes per second (*e.g.*, with $2^3$ cores each capable of $2^{22}$ hashes per second), and suppose the device has $m = 2^{21} \cdot 256$ bits $= 64$ MiB of shared memory. Memory capacity will be reached after $T = mpq/\ell n$ seconds of elapsed computation time. Thus, if we aim for $pq = 2^{20}$, the maximum selectable processing time on the device will be $2^{16}$ seconds (close to 1 day), in increments of $2^{-5}$ second. We can take $p = 2^{10} \cdot 3^2 \cdot 5^2 = 230\,400$ and hence $q = 4$ to get $pq \approx 2^{20}$. Last, we ascertain that, per all these choices, the available memory is still much larger than the $\ell p \approx 7$ MiB of overhead that are the price to pay for the parallelization option.

Suppose then that the user settles for $t = 2^5$ iterations (to take 1 second on the current device), and chooses a weak password with only 40 bits of entropy (from an implicit dictionary of size $d = 2^{40}$). In these conditions, an adversary will need $\ell t d = 2^{53}$ bits $= 1024$ TiB of memory in order to conduct a persistent attack. On a faster and/or more highly parallelized device, the user would choose a correspondingly larger value of $t$, further increasing the load on the adversary.

It is advisable to set $p$ as a large product of small factors, to facilitate the even distribution of workload among any number $N$ of CPUs such that $N$ divides $p$; this is easy to achieve in practice since the values of $pq$ tend to be quite large, on the order of $pq \geq 1\,000\,000$. A nice consequence is that the same HKDF can be dimensioned to accommodate any reasonably foreseen amount of user-side parallelism (hence the choice $p = 2^{10} \cdot 3^2 \cdot 5^2 = 230\,400$), and still be usable on today's sequential computers (with at least $\ell p \approx 7$ MiB of memory in this example).

## 2.6 Multiple Passwords

Since a major benefit of HKDFs is to provide long-term security to long-lived passwords, it is natural to envision them in backup and recovery applications. Specifically, one would like an HKDF with multiple passwords and widely differing values for the access delay $t$, including:

– Regular passwords: random and short-lived, these would benefit from very small values of $t$, in order not to slow down day-to-day operations; they would be strong but easily forgotten.

– Backup passwords: mostly for resetting regular passwords that are forgotten, these should be easier to remember over longer periods, and protected accordingly by using larger values of $t$.

– Disaster passwords: the fall-back of last resort, these should be confidently memorable over long stretches of dormancy; preempting their anticipated weaknesses whose thus dictate very big values of $t$, as security and memorability concerns trump most convenience considerations.

Naive implementations of this concept would require, either, that the user indicate to the system which one of the valid passwords has been entered, or, that the system blindly try out all roles in parallel for the supplied password. Alas, both approaches are bad for security: the former leaks valuable information to the attacker; and the latter is slow and reduces the tolerable values of $t$.

To answer this dilemma, we need an $m$Prepare algorithm whereby multiple "password slots" could be set up at once, and a matching $m$Extract procedure that would seamlessly surmise the right slot as if it were the only one. Luckily, both our generic HKDF and $p$HKDF constructions can accommodate multiple recovery passwords without affecting extraction performance. The idea is:

$(p)$**HKDF**$^H.m$**Prepare**$(r, (t_1, w_1), ..., (t_m, w_m), k_0)$

Inputs: random seed $r$, iteration and password pairs $(t_1, w_1), ..., (t_m, w_m)$, precursor key $k_0$.
Output: seed string $r$, test strings $h_1, ..., h_m$, mask strings $c_1, ..., c_m$, and key $k \leftarrow H(k_0, r)$.

0. DO in $m$ independent threads
1. $((r, h_1), k_1) \leftarrow \mathsf{Prepare}(r, t_1, w_1)$ ; $c_1 \leftarrow k_1 \oplus k_0$

$\vdots$ ...

$m$. $((r, h_m), k_m) \leftarrow \mathsf{Prepare}(r, t_m, w_m)$ ; $c_m \leftarrow k_m \oplus k_0$

$(p)$**HKDF**$^H.m$**Extract**$(r, (h_1, c_1), ..., (h_m, c_m), w)$

Inputs: seed string $r$, test and mask string pairs $(h_1, c_1), ..., (h_m, c_m)$, password $w$.
Output: decrypted key $k$, or no output for failure to halt.

1. create the initial state $(z)$ of Extract as in HKDF or $p$HKDF using inputs $((r, \phi), w)$
2. FOR $i := 1, ..., \infty$
3. update the state $(z, y_i)$ of Extract as in HKDF or $p$HKDF at iteration step $i$

$\star$ 1. IF $H(y_1, z) = h_1$ THEN RETURN $k \leftarrow H(\underbrace{H(z, r)}_{= k_1} \oplus c_1, r)$

$\vdots$ ...

$m$. IF $H(y_1, z) = h_m$ THEN RETURN $k \leftarrow H(H(z, r) \oplus c_m, r)$

The principle is to Prepare all the passwords individually, but with the same random salt $r$, to get multiple test strings $h_1, ..., h_m$, and publish the xor masks $c_1, ..., c_m$ that map the uncorrelated keys $k_1, ..., k_m$ to a communal $k_0$. The point is that one execution of Extract suffices to regain the non-malleable key $k = H(k_0, r)$ from any password, with no hint from the user about her choice.

# 3 The Security Gap

We show that any adversary lacking enormous amounts of memory will incur a $\sim 4\times$ larger cost for not knowing the iteration count. Since the penalty only strikes on wrong guesses, the user who knows the correct password will be immune to it. We say that *HKDFs widen the "security gap"*.

## 3.1 Offline Dictionary Attack Model

We consider the simplest and most general offline attack by an adversary $\mathcal{A}$ against a challenger $\mathcal{C}$. We capture the password "guessability" by supposing that it is drawn uniformly at random from a known dictionary $D$, and define its entropy as the value $\log_2(\#D)$. The game is as follows:

**Challenge.** The challenger $\mathcal{C}$ picks $w \in_\$ D$ and $r \in_\$ \{0,1\}^\ell$ at random, chooses $t \in \mathbb{N}$, and computes $(v, k) \leftarrow \mathsf{HKDF.Prepare}(r, t, w)$. It gives the string $v$ to $\mathcal{A}$.

**Attack.** The adversary $\mathcal{A}$ outputs as many keys as it pleases, sequentially: $k_1, k_2, ...$. It wins the game as soon as some $k_i$ matches $k = \mathsf{HKDF.Extract}(v, w)$.

We assume that $\mathcal{A}$ can only retain state for a dwindling fraction of $D$, of size $o(1)$ in $t$.

**Password (Min-)Entropy.** In reality, passwords are not sampled uniformly from a fixed $D$, but rather non-uniformly from a set with no clear boundaries. The worst-case unpredictability of a password chosen in this manner is the *minimum entropy*, or *min-entropy*, defined as $-\log_2(\max_w \Pr(w))$. The uniform password model $w \in_\$ D$ conveniently and accurately reflects the difficulty of guessing from $\mathcal{C}$'s true password distribution, provided that $\log_2(\#D)$ matches the min-entropy of the latter.

### 3.2 Finding the Optimal Attack Strategy

By Lemma 5 we know that $\mathcal{A}$ cannot do better than outputting random keys until it "tries out" the correct password $w$ for $t$ iterations (using the $\mathsf{Extract}$ function). Since $\mathcal{A}$ lacks the memory to maintain concurrent instances of $\mathsf{Extract}$ for any substantial subset of $D$, the only option is to "dovetail" the search, *i.e.*:

- try all the words of $D$ one by one (or few by few) for a bounded stretch of time;
- retry the same for longer and longer time stretches, until $t$ is eventually exceeded.

We can neglect the o(1) fraction of $D$ on which $\mathcal{A}$ could run a persistent attack. Also, for uniform $w \in_\$ D$ and unknown $t$ it is easy to show that it is optimal to spend the same amount of effort on each candidate password. We deduce that the optimal algorithm for any *forgetful* attacker $\mathcal{A}$ is:

$\mathsf{Optimal\text{-}MemoryBound}\text{-}\mathcal{A}_D(v)$

Input: verification string $v$.
Output: password $\hat{w}$ and key $\hat{k}$.

FOR $\hat{t} := t_1, t_2, ...$ /* $t_1 < t_2 < ...$ : the search schedule */
  FOR $\hat{w} \in D$ /* in sequence or partially parallel */
    RUN $\hat{k} \leftarrow \mathsf{Extract}(v, \hat{w})$ for $\hat{t}$ steps
    IF $\hat{k} \in \{0,1\}^\ell$ THEN /* did $\mathsf{Extract}$ halt spontaneously? */
      RETURN $(\hat{w}, \hat{k})$

The only parameters to be specified are the increasing sequence of iteration counts $t_1 < t_2 < ...$; the optimal schedule $(t_1, t_2, ...)$ will depend on $\mathcal{A}$'s uncertainty on $t$.

**Effort and Penalty.** We now quantify the total computation effort expended by $\mathcal{A}$ in function of the attack schedule $(t_1, t_2, ...)$. Let us denote by $\mathcal{W}^{\mathcal{A}}_{t_1,t_2,...}(t)$ the total expected number of hash evaluations made by $\mathcal{A}$ is the iteration count chosen by $\mathcal{C}$ is $t$. Let $k$ be the smallest index such that $t_k \geq t$. Let $d = \#D$, and define the constant $u = dq$. Since all of $D$ will be explored for each $t_i < t$, and only half of $D$ on expectation for the first $t_k \geq t$ (and nothing thereafter), we find that:

$$\mathcal{W}^{\mathcal{A}}_{t_1,t_2,...}(t) = \left( \sum_{i=1}^{k-1} t_i + \frac{t_k}{2} \right) u \ , \qquad \text{where} \quad \begin{cases} k = \min\{i : t_i \geq t\} \\ u = (\#D)\, q \end{cases} .$$

If, on the other hand, $\mathcal{A}$ had known the value of $t$ and just had to search for the password alone, the expected attack effort, denoted $\mathcal{W}^{\mathcal{A}}_t(t)$, would have been:

$$\mathcal{W}^{\mathcal{A}}_t(t) = \left( \frac{t}{2} \right) u \ , \qquad \text{with} \quad u = (\#D)\, q \ .$$

We define the *penalty* (of not knowing $t$) as the ratio: $\pi(t) = \frac{\mathcal{W}^{\mathcal{A}}_{t_1,t_2,...}(t)}{\mathcal{W}^{\mathcal{A}}_t(t)} = \frac{2\,(t_1 + ... + t_{k-1}) + t_k}{t} \geq 1$. Next, we show how to bound $\pi(t)$.

### 3.3 Bounding the Uncertainty Penalty

First, we should clarify that the goal of $\mathcal{A}$ is to minimize the value of $\pi(t)$ *on expectation* over the random choices made by $\mathcal{A}$ and $\mathcal{C}$, and not necessarily in the extremal cases where $t$ is either very small or very large. Indeed, it is not in the interest of $\mathcal{C}$ to choose too small a value for $t$. Furthermore, $\mathcal{A}$ can easily achieve $\pi(t) = 1$ for the maximal value of $t$ (we assume that $\mathcal{A}$ knows what hardware $\mathcal{C}$ uses), simply by setting $t_1 = t_{\max}$, but this would be a Pyrrhic victory since the attack would be utterly prohibitive and probably for naught. More generally, $\mathcal{A}$ cannot simply let $t_1$ be the largest "likely" value for $t$, since then $\mathcal{C}$ would figure it out and select $t = t_1 + 1$.

The foregoing strongly suggests that the game-theoretic optimum must be *scale-invariant* over the entire range $\{t_{\text{lo}}, ..., t_{\text{hi}}\} \ni t$ that $\mathcal{C}$ considers useful. It also suggests that $\mathcal{A}$ and $\mathcal{C}$ should use mixed (*i.e.*, randomized) strategies. We use the notation $[V]$ to denote the distribution of $V$.

**Lemma 6.** *Uniform equilibrium:* There exists a constant $\pi_0$, function of $t_{\text{lo}}$ and $t_{\text{hi}}$, such that a Nash equilibrium between $\mathcal{A}$ and $\mathcal{C}$ can only be reached for a randomized attack strategy such that $\forall t \in \{t_{\text{lo}}, ..., t_{\text{hi}}\} : \pi(t) = \pi_0$. The corresponding optimal strategy for $\mathcal{A}$ exists.

*Informal proof sketch.* Let $[(t_1, t_2, ...)]$ be an optimal mixture, or distribution of schedules, for $\mathcal{A}$, and suppose toward a contradiction that for this strategy the expected penalty $\pi(t)$ is not uniform over the entire range of acceptable values for $t$. Thus, there exist $t_{\text{easy}}$ and $t_{\text{hard}}$ in the interval $\{t_{\text{lo}}, ..., t_{\text{hi}}\}$ such that $\forall t : \pi(t_{\text{easy}}) \leq \pi(t) \leq \pi(t_{\text{hard}})$. Since the mixture is optimal, $\mathcal{C}$ can compute its parameters and select $t = t_{\text{hard}}$ to exert the stiffest expected penalty on $\mathcal{A}$. Predicting this, $\mathcal{A}$ would let $\rho = t_{\text{hard}}/t_{\text{easy}}$ and switch to a new mixture given by: $[(t'_1, t'_2, ...)] = [(\rho t_1, \rho t_2, ...)]$.

It is easy to see that $\pi'(t) = \pi'(t_{\text{hard}})$ under the new mixture equals $\pi(t_{\text{easy}}) < \pi(t)$ under the original one. It follows that the new strategy performs better than the old one when $\mathcal{C}$ consistently chooses $t = t_{\text{hard}}$ (which was $\mathcal{C}$'s optimal defense in response to $\mathcal{A}$'s supposedly optimal attack). It follows that the original strategy was not optimal after all, and we conclude that any optimal randomized attack must incur the same penalty $\pi_0 = \pi(t)$ for all $t \in \{t_{\text{lo}}, ..., t_{\text{hi}}\}$, as claimed. Existence of the randomized strategy characterized above follow from Nash. $\square$

**Lemma 7.** *Scale invariance:* In the limit $(t_{\text{hi}}/t_{\text{lo}}) \to \infty$, the optimal attack and defense strategies are scale-invariant. For $\mathcal{A}$ the optimal ratio $[(t_{i+1}/t_i)]$ converges in distribution to a mixture $[\alpha]$ that is independent of $i$. For $\mathcal{C}$ the optimal parameter $[t]$ assumes a Pareto power law whose probability density function $\frac{d}{dx}\Pr(t < x)$ proportional to $x^{-\beta}$ for some negative exponent $-\beta$ in the limit.

*Informal proof sketch.* Consider an optimal mixed strategy $[(t_1, t_2, ...)]$ and an iteration count $t$. Without loss of generality, we assume that $t_{\text{lo}} \ll t \ll t_{\text{hi}}$. Fix some $\delta > 0$, and let $t' = (1 + \delta)t$. By Lemma 6, we know that $\pi(t) = \pi(t')$. Now, consider the mixed schedule $[(t'_1, t'_2, ...)]$ obtained by subtituting $t'_i = (1 + \delta)t_i$ for $t_i$ everywhere, while keeping all probabilities the same. Denote by $\pi'$ the penalty function under that new schedule. By definition, we have the identity $\pi'(t') = \frac{1+\delta}{1+\delta}\pi(t) = \pi(t)$, and by transitivity we obtain that $\pi(t) = \pi'(t)$. We conclude that $\pi(t) = \pi'(t)$ for any distribution of $t$ over the interval $\{(1 + \delta)t_{\text{lo}}, ..., (1 + \delta)^{-1}t_{\text{hi}}\}$, for any $\delta > 0$.

Since the strategy is optimal, it follows that multiplying all the values in all the schedules it comprises by any constant $(1 + \delta)$ must preserve $\pi(t)$; this also works backward for $(1 + \delta)^{-1}$, and thus this is true in the limit for any multiplier in $\mathbb{R}^+$. In other words an optimal strategy for $\mathcal{A}$ is invariant to (multiplicative) scaling. A straightforward argument then shows that this must be reciprocated by the optimal response employed by $\mathcal{C}$. Approximating $t$ as a real in $\mathbb{R}^+$, we deduce that $t$ must obey a Pareto power law, *i.e.*, with density: $\frac{d}{dx}\Pr(t < x) \propto x^{-\beta}$ for some $\beta \in \mathbb{R}$.

For the remaining claim, we first note that the scale invariance implies that all the individual schedules $(t_1, t_2, ...)$ in the mixture must satisfy $(t_{i+1}/t_i) = (t_{j+1}/t_j)$ for all $i, j$, otherwise the multiplication by a constant would result in a different mixture. We have not yet ruled out the possibility of (sub-)mixtures $[(t_1, t_2, ...)], [(t'_1, t'_2, ...)], ...$ with unequal progressions $[(t_{i+1}/t_i)] \neq [(t'_{i+1}/t'_i)]$, which is why so far we say that $[(t_{i+1}/t_i)]$ converges to a distribution $[\alpha]$ instead of a value $\alpha^*$. $\square$

**Randomized Starting Point.** Lemmas 6 and 7 show that the optimal attack schedule for $\mathcal{A}$ is a randomized sequence $(t_1, t_2, ...)$ where $t_i = t_1 \alpha^{i-1}$ for some random starting point $t_1 \approx t_{\text{lo}}$ and a progression coefficient $\alpha \in_\$ [\alpha]$. For large enough $t \gg t_1$, the penalty becomes:

$$\pi(t) = \left(2\,\frac{t_1 + ... + t_{k-1}}{t} + \frac{t_k}{t}\right) \approx \frac{\alpha+1}{\alpha-1}\gamma\ , \qquad \text{for some} \quad \gamma = \frac{t_k}{t} \in [1, \alpha)\ .$$

Applying the scale invariance principle, we know that the expected $\pi(t)$ should be constant for varying $t$, which requires that $\gamma$ be distributed with density $\propto \gamma^{-1}$:

$$\frac{d}{dx}\Pr(\gamma < x) = \begin{cases} x^{-1}/\ln\alpha & \text{for } 1 \leq x < \alpha \\ 0 & \text{otherwise} \end{cases}\ . \tag{1}$$

Since $\gamma$ has expectation $\int_1^\alpha x\,\frac{x^{-1}}{\ln\alpha}\,dx = \frac{\alpha-1}{\ln\alpha}$, the uniform penalty for all choices of $t$ is thus:

$$\pi(t) \approx \pi_0 = \frac{\alpha+1}{\ln(\alpha)}\ . \tag{2}$$

**Optimal Progression Coefficient.** The last thing we need is to compute $\pi_0$ in function of the progression coefficient $\alpha$, which is drawn from some distribution $[\alpha]$ yet to be specified. Notice from Equation (2) that $\pi_0$ is a convex function of $\alpha$ that reaches a minimum for some $\alpha = \alpha^* \in (1, \infty)$, hence the optimal $[\alpha]$ is the pointwise distribution centered on $\alpha^*$. Asymptotically, the numerical values of the optimal attack coefficient $\alpha^*$ and the corresponding minimal penalty $\pi_0^*$ are given by:

$$\alpha^* = \arg\min_\alpha \frac{\alpha+1}{\ln(\alpha)}\ , \qquad \pi_0^* = \frac{\alpha^*+1}{\ln(\alpha^*)}\ , \qquad \pi_0^* = \alpha^* \approx 3.59112147666862\ . \tag{3}$$

To implement the optimal strategy, a rational attacker $\mathcal{A}$ would fix $\alpha = \alpha^*$ from Equation (3), and start the search schedule from some random $t_1 = t_{\text{lo}}\gamma$ where $\gamma$ is distributed as in Equation (1). No matter how cleverly $\mathcal{C}$ chooses $t$, the expected penalty incurred by $\mathcal{A}$ is $\pi(t) = \pi_0^* \approx 3.5911215$. (We mention that the same constant, 3.59112..., arises in the context of the cow-path problem [23], which is a hidden search problem with a related structure and also with scale-invariance properties.)

Reciprocally, we determine the optimal Pareto exponent $-\beta^*$ that a rational $\mathcal{C}$ should choose to oppose $\mathcal{A}$. Straightforward calculations yield $-\beta^* = -1$, *i.e.*, the "inverse law" $\frac{d}{dx}\Pr(t < x) \propto \frac{1}{x}$.

## 3.4 Justifying the Inverse-Law Hypothesis

We have shown that (for the stated objective of maximizing the expected security gap) the optimal distribution $[t]$ is a power law of exponent $-\beta^* = -1$ over some fixed and fairly wide interval of interest. The question is whether this is a reasonable assumption to make for the behavior of $\mathcal{C}$:

*Would a typical user not always program the same key derivation delay?*

The first answer to this question depends on the user's psychology, and his or her understanding of the benefits provided by HKDFs. In fact, it is sufficient that the attacker *believes* that the user has a good reason to use very long delays on occasion (*e.g.*, to protect a particularly sensitive ciphertext, or to shield a long-term backup password that will only be used as a last resort, as we already discussed in the introduction). Of course, if the attacker does not believe such a thing, but the user acts on it nonetheless, it will be entirely to the attacker's detriment.

The second answer is a phenomenological one. Pareto or Zipf distributions (of law $\propto n^{-\beta}$) have been noted to occur ubiquitously in the heavy tail of empirical distributions in a variety of contexts, ranging from physics and geology with cosmic-ray energies and oil-field reserves, to linguistics with the relative frequency of words in literary pieces, to economics regarding distribution of income and normalized returns of securities, and even to anthropology with the size of human population centers (see [34] for a list of these phenomena). Hence, it seems justifiable to assume that for large ensembles of users and/or ciphertexts, the induced iteration count $t$ would abide by a power law. This hypothesis draws further credence from a pattern in natural and human sciences [31, 25] that the most frequently seen empirical exponents $-\beta \approx -(1+\epsilon)$ mirror our own ideal $-\beta^* = -1$.

### 3.5 Quantifying the HKDF Security Gain

The practical HKDF-to-KDF security boost against realistic attackers is thus, with ideal hashing:

**Theorem 8.** *Security gain:* Under the reasonable hypothesis that $[t]$ has scale-invariant law $\sim t^{-1}$, and that to an offline dictionary attacker the available memory grows more slowly than time, then, asymptotically, HKDFs increase the " effective entropy" of all passwords, over regular KDFs, by:

$$\log_2(\pi_0^*) \approx 1.84443445579378 \text{ bits} .$$

## 4 Real-world Implementation

We believe that the case is strong for dropping KDFs in favor of HKDFs wherever possible, and to make it even stronger we discuss two compelling real-world applications.

We present two implementations of HKDFs on GNU/*Linux* systems, which we intend to release as open-source portable (POSIX) C code. Our first prototype is as a stand-alone command-line tool to be used in conjuction with programs such as GNU `gpg` [15] or Ruusu's `aespipe` [1] to assemble strong password-based encryption pipelines. Our second prototype is a patch for the `truecrypt` [40] "plausibly deniable" disk encryption software, which dramatically increases its resistance to offline dictionary attacks, and thus plausible deniability by implication.

We will see that HKDFs are much more secure in practice than the KDFs they replace, at the cost of little tweaks to the UI, minimal impact on the user behavior, and no change to the hardware.

### 4.1 *TrueCrypt* Disk Encryption

*TrueCrypt* [40] is a password-based disk encryption software of modern design, developed for *Windows* and subsequently ported to *Linux*, and available under a permissive open-source license. *TrueCrypt* is aimed at local storage encryption underneath the filesystem. It provides *plausible deniability*, meaning that a `truecrypt`-encrypted disk should be indistinguishable from a `shred`-ded disk to anyone who lacks the password. Free-space ciphertext *and plaintext* are designed to look random: this allows a nested volume to be hidden in a container volume's free space. This is perhaps the central feature of the design, and `truecrypt` is able to avoid clobbering the hidden volume when writing on the container, as long as both volumes are mounted.

The cryptographic design is otherwise fairly standard. A password-based KDF-encrypted header holds a randomly generated key, needed for encrypting the bulk of the data (*i.e.*, the disk sectors). One peculiarity is that the KDF iteration count cannot be recorded because the encrypted volume *including the header* must appear random, and so it is burned into the `truecrypt` binary.

**Plausible Deniability from HKDF.** As it turns out, HKDFs are quite salutary for deniability:

1. Unlike KDFs, HKDFs have no distinguishable headers. This makes it possible to vary the iteration count on a per-ciphertext basis, without giving away the presence of encryption.
2. Ultimately, plausible deniability hinges on the inquirer's inability to crack the user's password. Since this is harder if HKDFs are used, plausible deniability is correspondingly enhanced.

**Implementation and Interface.** *TrueCrypt*'s KDF is PKCS#5 with a user-selected hash (SHA1, RipeMD160, or WhirlPool) and a hard-coded iteration count (2000, 2000, and 1000 respectively). Since the hash selection cannot be recorded in the volume any more than the iteration count, `truecrypt` simply tries each of the three derivation functions in sequence until one works.

We shall add an HKDF as a fourth option, per the template of Section 2.2 with SHA1 hashing. It should appear last in the sequence, and have a manual interruption facility via the user interface. The *TrueCrypt* volume format allocates 64 bytes for the random salt; we reclaim 40 of those for the HKDF public string $v$ (which *is* random; see Section 2), and pad the remaining 24 at random.

We begin with a clean HKDF programming interface:

```
ulong                                                   /* return: final value of t */
HKDF_prepare( ulong                          tmax ,     /* input: maximum t, or 0   */
              uchar const * w           , uint w_sz ,   /* input: password w        */
              uchar const * r           , uint r_sz ,   /* input: randomness r      */
              uchar       * v           , uint v_sz ,   /* output: public string v  */
              uchar       * k           , uint k_sz ,   /* output: derived key k    */
              int (* ui)( ulong, void *) , void * ob );  /* callback: user approval  */

ulong                                                   /* return: final value of t */
HKDF_extract( ulong                          tmax ,     /* input: maximum t, or 0   */
              uchar const * w           , uint w_sz ,   /* input: password w        */
              uchar const * v           , uint v_sz ,   /* input: public string v   */
              uchar       * k           , uint k_sz ,   /* output: derived key k    */
              int (* ui)( ulong, void *) , void * ob );  /* callback: user annulment */
```

We build a modified version of *TrueCrypt*, called `hkdf-tc`, that invokes `HKDF_prepare()` when asked to create a new volume with the HKDF flag turned on, and defers to `HKDF_extract()` when asked to mount a volume with an undecipherable header. Although both functions take a parameter `tmax` that sets the admissible upper bound for $t$, the actual selection of $t$ is designed to be interactive via the callback predicate `(*ui)()`, which supplies the outer loop iterator $i$ to the user interface:

*When creating a new volume,* `hkdf-tc` asks the user to enter the same password twice, and to choose a number of options. If the HKDF option is selected, `HKDF_prepare()` will invite the user to press a key after any—short or long—delay, explaining that the same delay will be incurred every time the volume is mounted as a defense against password guessers.

*When mounting an existing volume,* `hkdf-tc` queries the password and tries the built-in KDFs. If these fail, `HKDF_extract()` is invoked, and the user instructed to press a key if it is taking too long, for the program cannot distinguish a wrong password from one with a longer delay.

In both cases, computations proceed in the background, until the user signal is caught by the callback `(*ui)()`, which polls the keyboard via a non-blocking system call at every iteration of the main loop. The low polling frequency $\sim$ 1–30 Hz makes this solution responsive but not wasteful.

With a graphical UI, the preferred approach is to add a button to the password entry dialog, greyed out at first, and becoming clickable once the user has entered a password: its label as commissioned by `HKDF_prepare()` could be [wrap up]; or [give up] when commissioned by `HKDF_extract()`. A graphical user interface could also include a busy indicator, progress bar, iteration counter, *etc.*

## 4.2 Command-line HKDF Tool

Our second implementation is a small command-line tool, called `hkdf`, whose usage is as follows:

```
1.  hkdf -p [-r|-s] [-t MAX]       prompts for a passphrase, and prints a public string
    hkdf -k [-r|-s] [-t MAX] FILE  instead, writes public str to FILE and prints the key
2.  hkdf [-d] [-t MAX]             reads public str, asks for a passphr, and prints a key
```

```
Arguments:  -p|-k   PREPARE mode --- once running, press the * key to finish
            -d      EXTRACT mode (the default) --- press Control-C to cancel
            -r      reads randomness from stdin (instead of /dev/random)
            -s      reads passphrase from stdin (instead of user prompt)
            -t MAX  triggers auto-finish or auto-cancel at iteration MAX
```

Each of the following commands creates a random public string $v$, saves it to the file `public.v`, and prints the corresponding key $k$ on standard output, based on the user's passphrase. The second command asks for the passphrase twice (on behalf of `hkdf -p` and `hkdf -d`, in unspecified order), and re-derives $k$ on-the-fly to provide end-to-end verification without committing any secret to disk.

```
# hkdf -k public.v
# hkdf -p | tee public.v | hkdf -d
```

The user must press the `*` key at some time after entering the passphrases(s) (or use the `-t` option) to set the key derivation delay. To recover $k$ from `public.v` at a later time, we use:

```
# hkdf < public.v
```

which prompts for the passphrase once.

**Encryption with *AESpipe* and *GnuPG*.** We can combine `hkdf` with `aespipe` [1] to assemble a (randomized) password-based AES encryptor with HKDF resistance to dictionary attacks. The plaintext is a file `plain.bin` and the ciphertext will consist of two files `crypt.v` and `crypt.aes`. To encrypt:

```
# aespipe -p 4 4<<<`hkdf -k crypt.v` < plain.bin > crypt.aes
```

In the Bourne shell (`/bin/sh`), the string `` 4<<<`...` `` causes the command between the backquotes to be executed in a sub-shell, and its output redirected to the parent's unused file descriptor #4; meanwhile, the parameter `-p 4` instructs `aespipe` to fetch its key from the same. To decrypt:

```
# aespipe -d -p 4 4<<<`hkdf < crypt.v` < crypt.aes > decrypted.bin
```

This command works similarly. If the passphrase is good, `hkdf` will feed the right key to `aespipe`; otherwise, it will run forever until interrupted by Control-C.

The `hkdf` tool is even easier to interface with other programs, *e.g.*, `gpg` [15]:

```
# hkdf -k crypt.v | gpg --passphrase-fd 0 -o crypt.gpg -c plain.bin
# hkdf < crypt.v | gpg --passphrase-fd 0 -o decrypted.bin crypt.gpg
```

This is merely suggestive; more sophisticated scripts could merge the ciphertext into a single file.

***GnuPG* Key-rings.** Since the user passphrase is the Achilles' heel of the system, an excellent use of the `hkdf/gpg` synergy is to replace `gpg`'s default key-ring encryption with something stronger. To quote the `gpg(1)` manual page:

> *WARNINGS*
>
> *Use a *good* password for your user account and a *good* passphrase to protect your secret key. This passphrase is the weakest part of the whole system. Programs to do dictionary attacks on your secret keyring are very easy to write and so you should protect your "~/.gnupg/" directory very well.*

HKDFs are an excellent way to add protection with or without changing the passphrase. Our `hkdf` tool and a small script to bind it to `gpg` are all that is needed.

### 4.3 Concrete Security Gains

We now quantify the security gained by upgrading *TrueCrypt* and *GnuPG* from KDF to HKDF. Our test platform is a 1.5 GHz single-core x86 laptop running *Debian Linux*.

**Baseline Measurements.** First we clock the various built-in KDFs to establish the benchmark:

| Software | Digest function | Normalized speed | Fixed multiplier | Time per password (as measured) |
|---|---|---|---|---|
| `truecrypt` | HMAC-SHA1 | 25200 #/s | 2000 # | 79 ms |
| | HMAC-RIPEMD160 | 20400 #/s | 2000 # | 98 ms |
| | HMAC-WHIRLPOOL | 9700 #/s | 1000 # | 101 ms |
| `gpg` | MD5 | 30.0 MB/s | 65536 B | 2.2 ms |
| | SHA1 (default) | 28.0 MB/s | 65536 B | 2.3 ms |
| | SHA256 | 15.2 MB/s | 65536 B | 4.3 ms |
| | SHA512 | 9.9 MB/s | 65536 B | 6.6 ms |

These timings were obtained by instrumenting the relevant sections of code, in order to suppress overheads and obtain an accurate indication of the amount of work needed for a brute-force attack.

**HKDF Performance.** Next, we measure the performance of the HKDF implementation, and the rate at which the size of the state is increased:

| $H$ algorithm for $\mathsf{HKDF}^H$ | Hash width $\ell$ | $\mathsf{HKDF}$ throughput | Time resolution and Memory rate (@1 CPU) ($q = 57600$) | | ($q = 230400$) | |
|---|---|---|---|---|---|---|
| SHA1 | 160 | 25.1 MB/s | 11.0 Hz | 220 B/s | 2.8 Hz | 56 B/s |
| WHIRLPOOL | 512 | 19.7 MB/s | 2.7 Hz | 173 B/s | 0.7 Hz | 45 B/s |

As we would expect, the raw throughput is very close to but slightly less than a "pure" implementation of the corresponding hash function (*e.g.*, compare the SHA1 instantiation with `gpg` above). The discrepancy is caused by the modular reduction in the inner loop of the HKDF algorithm.

**Attainable Security Gains.** We now find the actual key derivation complexity (time and space) for several user-programmed delays, and what this entails for an optimal attacker. We fix $q = 57600$:

| Program | $H$ for $\mathsf{HKDF}^H$ | Time & Memory (per password) : Programmed | | Adversarial | | Security gain *vs.* built-in KDF |
|---|---|---|---|---|---|---|
| `hkdf-tc` (*vs.* `truecrypt`) | WHIRLPOOL | 3 sec. | < 1 kB | 11 sec. | 2 kB | $10^2\times$ ($\sim$ 7 bits) |
| | | 4 min. | 41 kB | 14 min. | 147 kB | $10^4\times$ ($\sim$ 13 bits) |
| | | 45 min. | 0.5 MB | 3 hours | 1.7 MB | $10^5\times$ ($\sim$ 17 bits) |
| `hkdf/gpg` (*vs.* `gpg`) | SHA1 | 1 sec. | < 1 kB | 4 sec. | 1 kB | $10^3\times$ ($\sim$ 10 bits) |
| | | 10 min. | 131 kB | 36 min. | 469 kB | $10^6\times$ ($\sim$ 20 bits) |
| | | 2 hours | 1.6 MB | 7 hours | 5.5 MB | $10^7\times$ ($\sim$ 23 bits) |

The last column shows the actual security gain provied by HKDFs in comparison to the benchmarks. For the most casual uses (where the HKDF preparation is finished without deliberate delay), we expect a steady security gain of about $\sim 7$ bits over *TrueCrypt*, and about $\sim 11$ bits over *GnuPG*. For more sensitive uses, gains of $\sim$ 15–20 bits can be attained with a few minutes of patience. For long-term backups where two-hour waits can be justified, the gain reaches $\sim 23$ bits over *GnuPG*. The security gain further increases by $\sim \log_2(N)$ bits in all cases if the user's machine has $N$ CPUs.

To give a very concrete example, a *GnuPG* secret key file will be equally well protected with an 11-letter all-lowercase password ($\sim 51$ bits of entropy) by `gpg` itself, as by our `hkdf` system with a 6-letter password ($\sim 28$ bits of entropy) plus a two-hour wait—or eight-minute on a sixteen-core machine. An infrastructure the scale of Google ($\sim 10^5$ CPUs) would take two years to crack either.

**Attack Times.** Our last table compares the times to crack one password in *GnuPG*, *TrueCrypt*, and various HKDF use cases, in function of the password strentgh (40 and 60 bits of entropy, or $\sim 9$ and $\sim 13$ random lowercase letters, respectively), against a spectrum of opponents:

| Opponent | # CPUs | *GnuPG* | *TrueCrypt* | HKDF | | | |
|---|---|---|---|---|---|---|---|
| | | | | 1-core | | 32-core | |
| 40-bit secret | | | | 1 s | 10 m | 1 s | 1 h |
| Individual | $10^1$ | 7.7 y | 275 y | 12.5 k y | 7.5 M y | 401 k y | 1.4 G y |
| Corporation | $10^4$ | 67 h | 101 d | 13 y | 7.5 k y | 401 y | 1.4 M y |
| Huge botnet | $10^7$ | 242 s | 2.4 h | (31 h)† | 7.5 y | (41 d)† | 1.4 k y |
| "The World" | $10^{10}$ | 242 ms | 8.6 s | (110 s)† | (18 h)† | (59 m)† | (147 d)† |
| 60-bit secret | | | | | | | |
| Government | $10^6$ | 80 y | 2.9 k y | 131 k y | 79 M y | 4.2 M y | 15 G y |
| "The World" | $10^{10}$ | 70 h | 105 d | 13 y | 7.9 k y | 420 y | 1.5 M y |

†The flagged figures relate to a *persistent* attack, feasible for these parameters if the opponent has 1 GiB per CPU.

Even with "instantaneous" user delays ($\sim 1$ s), the security gains are substantial and may suffice to turn a successful attack into a successful defense. Larger delays ($> 1$ min.–1 hr.) are surprisingly secure with the inherent benefits of HKDFs; they are justified for last-resort disaster-recovery backups, which must remain secure, and their passwords not forgotten, over long cryptoperiods.

## References


[1] AESpipe - AES encrypting or decrypting pipe. `http://loop-aes.sourceforge.net/`.

[2] A. Back. Hashcash. Technical report, 1997. `http://www.cypherspace.org/hashcash/`.

[3] E. Barkan, E. Biham, and A. Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In *Advances in Cryptology—CRYPTO 2006*, volume 4117 of *LNCS*. Springer-Verlag, 2006.

[4] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology—EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–55. Springer-Verlag, 2000.

[5] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy—SP 1992*, pages 72–84. IEEE Press, 1992.

[6] D. R. L. Brown. Prompted user retrieval of secret entropy: The passmaze protocol. Cryptology ePrint Archive, Report 2005/434, 2005. `http://eprint.iacr.org/`.

[7] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format. RFC 2440, November 1998. `http://www.ietf.org/rfc/rfc2440.txt`.

[8] R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. MacKenzie. Universally composable password-based key exchange. In *Advances in Cryptology—EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–21. Springer-Verlag, 2005.

[9] R. Canetti, S. Halevi, and M. Steiner. Mitigating dictionary attacks on password-protected local storage. In *Advances in Cryptology—CRYPTO 2006*, volume 4117 of *LNCS*. Springer-Verlag, 2006. Full version: Cryptology ePrint Archive, Report 2006/276. `http://eprint.iacr.org/`.

[10] CryptoCard. `http://www.cryptocard.com/`.

[11] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *USENIX Security Symposium—SECURITY 2001*, 2001. `http://www.usenix.org/events/sec01/full_papers/dean/dean.pdf`.

[12] DigiPass. `http://www.vasco.com/`.

[13] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *Advances in Cryptology—CRYPTO 2003*, volume 2729 of *LNCS*. Springer-Verlag, 2003.

[14] C. Dwork and M. Naor. Pricing via processing or combating junk mail. In *Advances in Cryptology—CRYPTO 1992*, volume 740 of *LNCS*, pages 139–47. Springer-Verlag, 1992.

[15] GnuPG - the GNU privacy guard. `http://www.gnupg.org/`.

[16] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *Proceedings of WWW 2005*, 2005.

[17] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. In *ACM Conference on Computer and Communications Security—CCS 1998*, pages 122–31. ACM Press, 1998.

[18] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Information Theory*, 26(4):401–6, 1980.

[19] IEEE P1363.2: Password-based public-key cryptography. `http://grouper.ieee.org/groups/1363/`.

[20] D. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 1996.

[21] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *Proceedings of NDSS 1999*, pages 151–65, 1999.

[22] B. Kaliski. PKCS #5: Password-based cryptography specification, version 2.0. RFC 2898, September 2000. `http://www.ietf.org/rfc/rfc2898.txt`.

[23] M.-Y. Kao, J. H. Reif, and S. R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. In *ACM-SIAM Symposium on Discrete Algorithms—SODA 1993*, 1993.

[24] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, February 1997. `http://www.ietf.org/rfc/rfc2104.txt`.

[25] J. Laherrere and D. Sornette. Stretched exponential distributions in nature and economy: 'fat tails' with characteristic scales. *European Physical Journals*, B2:525–39, 1998. `http://xxx.lanl.gov/abs/cond-mat/9801293`.

[26] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–93, 2001.

[27] P. MacKenzie, T. Shrimpton, and M. Jakobsson. Threshold password-authenticated key exchange. *Journal of Cryptology*, 19(1):27–66, 2006.

[28] W. Mao. Send message into a definite future. In *Proceedings of ICICS 1999*, volume 1726 of *LNCS*, pages 244–51. Springer-Verlag, 1999.

[29] M. Naor and B. Pinkas. Visual authentication and identification. In *Advances in Cryptology—CRYPTO 1997*, volume 1294 of *LNCS*, pages 322–36. Springer-Verlag, 1997.

[30] P. Oechslin. Making a faster cryptanalytical time-memory trade-off. In *Advances in Cryptology—CRYPTO 2003*, volume 2729 of *LNCS*, pages 617–30. Springer-Verlag, 2003.

[31] R. Perline. Zipf's law, the central limit theorem, and the random division of the unit interval. *Physical Review E*, 54(1):220–3, 1996.

[32] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *ACM Conference on Computer and Communications Security—CCS 2002*, pages 161–70. ACM Press, 2002.

[33] N. Provos and D. Mazières. A future-adaptable password scheme. In *USENIX Technical Conference—FREENIX Track 1999*, 1999. `http://www.usenix.org/events/usenix99/provos/provos.pdf`.

[34] W. J. Reed. The Pareto, Zipf and other power laws. *Economics Letters*, 74(1):15–9, 2001.

[35] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report MIT-LCS-TR-684, MIT, 1985. `http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-684.pdf`.

[36] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security Symposium—SECURITY 2005*, 2005.

[37] RSA-Laboratories. PKCS #5: Password-based encryption standard, version 1.5, November 1993. See also [22].

[38] SecurID. `http://www.rsasecurity.com/`.

[39] A. Stubblefield and D. Simon. Inkblot authentication. Tech. report MSR-TR-2004-85, Microsoft Research, 1985.

[40] TrueCrypt - free open-source on-the-fly disk encryption software. `http://www.truecrypt.org/`.

[41] J. Viega, T. Kohno, and R. Housley. Patent-free, parallelizable MACing. Crypto Forum Research Group, December 2002. `http://www1.ietf.org/mail-archive/web/cfrg/current/msg00126.html`.

[42] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Advances in Cryptology—CRYPTO 2003*, volume 2656 of *LNCS*. Springer-Verlag, 2003.

[43] J. Yan, A. Blackwell, R. Anderson, and A. Grant. The memorability and security of passwords - some empirical results. *IEEE Security and Privacy*, 2(5):25–31, 2004.

[44] K.-P. Yee and K. Sitaker. Passpet: Convenient password management and phishing protection. In *Symposium On Usable Privacy and Security—SOUPS 2006*. ACM Press, 2006.