A Future-Adaptable Password Scheme

Niels Provos and David Mazières
{provos,dm}@openbsd.org
The OpenBSD Project

Abstract

Many authentication schemes depend on secret passwords. Unfortunately, the length and randomness of user-chosen passwords remain fixed over time. In contrast, hardware improvements constantly give attackers increasing computational power. As a result, password schemes such as the traditional UNIX user-authentication system are failing with time.

This paper discusses ways of building systems in which password security keeps up with hardware speeds. We formalize the properties desirable in a good password system, and show that the computational cost of any secure password scheme must increase as hardware improves. We present two algorithms with adaptable cost—*eksblowfish*, a block cipher with a purposefully expensive key schedule, and *bcrypt*, a related hash function. Failing a major breakthrough in complexity theory, these algorithms should allow password-based systems to adapt to hardware improvements and remain secure well into the future.

1 Introduction

As microprocessors grow faster, so does the speed of cryptographic software. Fast cryptography opens many opportunities for making systems more secure. It renders encryption usable for a wide range of applications. It also permits larger values of tunable security parameters such as key length. Increasing security parameters makes cryptography exponentially (or at least superpolynomially) more difficult to break, dwarfing any benefit faster hardware may offer attackers. Unfortunately, one security parameter—the length and entropy of user-chosen passwords—does not scale at all with computing power. While many systems require users to choose secret passwords for authentication, few actually adapt their algorithms to preserve security in the face of increasingly powerful attackers.

One widespread use of passwords, and a good example of failure to adapt, is the UNIX password system. UNIX, a multi-user operating system, requires users to prove their identity before accessing system resources. A user typically begins a session by providing her username and secret password to a login program. This program then verifies the password using a system-wide password file. Given the importance of keeping passwords secret, UNIX does not store plaintext passwords in this file. Instead, it keeps *hashes* of passwords, using a one-way function, *crypt* [9], that can only be inverted by guessing preimages. To verify a password, the login program hashes the password and compares the result to the appropriate hash in the password file.

At the time of deployment in 1976, *crypt* could hash fewer than 4 passwords per second. Since the only known way of inverting *crypt* is to guess preimages, the algorithm made passwords very difficult to recover from their hashes—so much so, in fact, that the designers of UNIX felt comfortable leaving the password file readable by all users. Today, over 20 years later, a fast workstation with heavily optimized software can perform over 200,000 *crypt* operations per second. Attackers can now expediently discover plaintext passwords by hashing entire dictionaries of common passwords and comparing the results to entries in a password file. *crypt* nonetheless still enjoys widespread use, and legacy software even forces many sites to keep their password files readable by all users.

Today we have authentication schemes considerably more sophisticated than the UNIX password file. In practice, however, implementations of these schemes still often depend on users remembering secret passwords. There are alternatives, such as issuing special authentication hardware to users or giving them printed lists of randomly generated access codes, but these approaches generally inconvenience users or incur additional cost. Thus, passwords continue to play an important role in the vast majority of user-authentication systems.

This paper discusses ways of building systems in which password security keeps up with hardware speeds. We present two algorithms with adaptable cost—*eksblowfish*, a block cipher with a purposefully expensive key schedule, and *bcrypt*, a related hash function. Failing a major breakthrough in complexity theory, these algorithms should allow password-based systems to adapt to hardware improvements and remain secure 20 years into the future.

The rest of the paper is organized as follows. In Section 2, we discuss related work on password security. In Section 3, we explain the requirements for a good password scheme. Section 4 presents *eksblowfish*, a 64-bit block cipher that lets users tune the cost of the key schedule. Section 5 introduces the variable-cost *bcrypt* password hashing function and describes our implementation in the OpenBSD operating system. Finally, Section 6 compares *bcrypt* to two widely-used password hashing functions.

2 Related Work

Password guessing attacks can be categorized by the amount of interaction they require with an authentication system. In *on-line* attacks, the perpetrator must make use of an authentication system to check each guess of a password. In *off-line* attacks, an attacker obtains information—such as a password hash—that allows him to check password guesses on his own, with no further access to the system. On-line attacks are generally considerably slower than off-line ones. Systems can detect on-line attacks fairly easily and defend against them by slowing the rate of password checking. In contrast, once an attacker has obtained password verification information, the only protection a system has from off-line attacks is the computational cost of checking potential passwords.

Techniques for mitigating the threat of off-line password guessing generally aspire to one of two goals—limiting a system's susceptibility to off-line attacks or increasing their computational cost. As a simple example of the former, many modern UNIX systems now keep password hashes secret from users, storing them in a read-protected *shadow* password file rather than in the standard openly readable one.

Much of the work on preventing off-line password attacks has centered around communication over insecure networks. If cryptographic protocols rely on user-chosen passwords as keys, they may open themselves up to off-line guessing attacks. Gong et. al. [7] suggest several protocol design tricks to thwart password guessing by network attackers. Unfortunately, their most interesting proposals require encryption algorithms with unusual and difficult to achieve properties.

Several people have designed secure password protocols that let users authenticate themselves over insecure networks without the need to remember or certify public keys. Bellovin and Merritt [2, 3] first proposed the idea, giving several concrete protocols putatively resistant to off-line guessing attacks. Patel [11] later cryptanalyzed those protocols, but people have since continued developing and refining others in the same vein. More recent proposals such as SRP [16] show promise of being secure.

Of course, even a secure password protocol requires some server capable of validating users with correct passwords. An attacker who obtains that server's secret state can mount an off-line guessing attack. Because secure password protocols require public key cryptography [8], they do have a tunable key length parameter. However, this parameter primarily controls the difficulty of mounting off-line attacks without a server's secret state; it only indirectly affects the cost of an off-line attack given that state. Tuning key length to preserve password guessing costs would have other unintended consequences, for instance increasing message sizes and costing servers unnecessary computation. By combining a scheme like SRP with the *bcrypt* algorithm presented in this paper, however, one can vary the cost of guessing passwords independently from most other properties of the protocol.

Whatever progress occurs in preventing off-line attacks, one can never rule them out entirely. In fact, the decision to have an openly readable password file was not an oversight on the part of the UNIX system designers [9]. Rather, it was a reaction to the difficulty of keeping the password file secret in previous systems, and to the realization that a supposedly secret password file would need to resist off-line guessing anyway. This realization remains equally true today. Aside from the obvious issues of backup tape security, attackers who compromise UNIX machines routinely make off with the list of hashed passwords, whether shadowed or not.

A poor hashing algorithm not only complicates recovery from break-ins, it also endangers other machines. People often choose the same password on multiple machines. Many sites intentionally maintain identical password files on all machines for administrative convenience. While shadow password files certainly do not hurt security, the big flaw in UNIX password security is not its openly readable password file. Rather, it is the choice of a hash function that cannot adapt to a 50,000 fold increase in the speed of hardware and software. This paper presents schemes that can adapt to such improvements in efficiency.

Others have already proposed numerous schemes to increase the cost of guessing passwords. The FreeBSD operating system, for instance, introduced a replacement for *crypt* based on the MD5 [13] message digest algorithm. MD5 *crypt* takes considerably longer to compute than the original *crypt*. Yet, it still has a fixed cost and thus cannot not adapt to faster hardware. As time passes, MD5 *crypt* offers steadily decreasing protection against off-line guessing attacks. Significant optimizations have already been found to speed up the calculation of MD5 *crypt*.

Abadi et. al. [1] propose strengthening user-chosen passwords by appending random bits to them. At authentication time, software uses the known part of the password and a hash of the full password to guess the random bits. As hardware gets faster, one can easily tune this technique by increasing the number of random bits. Unfortunately, password strengthening inherently gives unauthenticated users the ability to mount off-line guessing attacks. Thus, it cannot be combined with techniques like SRP that attempt to limit the possibility of off-line attacks in the first place.

Finally, many systems rely less directly on password security for authentication. The popular ssh [17] remote login program, for example, allows users to authenticate themselves using RSA encryption. Ssh servers must have a user's RSA public key, but they need not store any information with which to verify user-chosen passwords. The catch is, of course, that users must store their private keys somewhere, and this usually means on disk, encrypted with a password. Worse yet, ssh uses simple 3-DES to encrypt private keys, making the cost of guessing ssh passwords comparable to the cost of computing *crypt*. Nonetheless, because of its flexibility, ssh's RSA authentication is a generally better approach than schemes more closely tied to passwords. For example, without modifying the core protocols, ssh could easily employ the ***eksblowfish*** algorithm proposed in this paper to improve the security of stored secret keys.

3 Design criteria for password schemes

Any algorithm that takes a user-chosen password as input should be hardened against password guessing. That means any public or long-lived output should be of minimal use in reconstructing the password. Several design criteria can help achieve this goal.

Ideally, one would like any password handling algorithm to be a strong one-way function of the password—that is, given the algorithm's output and other inputs, an attacker should have little chance of learning even partial information she could not already have guessed about the password. Unfortunately, one-way functions are defined asymptotically with respect to their input lengths—an attacker has negligible probability of inverting a one-way function on sufficiently large inputs, but exactly how large depends on the attacker. Because there is a fixed limit to the size of passwords users will tolerate, we need a different criterion for functions on passwords.

Informally, we would like a password scheme to be "as good as the passwords users choose." Given a probability distribution D on passwords, we define the *predictability* $R(D)$ of the distribution to be the highest probability $\Pr(s)$ of any single password s in D: $R(D) = \max_{s \in D} \Pr(s)$. A function of a password is secure if an attacker's probability of learning any partial information about the password is proportional to the product of the work she invests and the predictability of the password distribution.

What does it mean for an attacker to learn partial information about a password? We define partial information to be the value of any single-bit predicate on a password. Interesting predicates on passwords might include the first bit of a password, or the parity of bits in a password. An attacker can always guess certain predicates with high probability—for instance, the trivial predicate $P(s) = 1$ which returns 1 on all passwords. If a function of a password is secure, however, its output should not let an attacker guess any predicate more accurately than she could have without the function's output.

More formally, let $F(s,t)$ be a function. The argument s represents a user's secret password, which will be drawn from a probability distribution D. The argument t represents any additional non-secret inputs F might take. Let the values of t be drawn from a probability distribution T. We model an attacker as a randomized boolean circuit[1], A, that tries to guess a predicate P about a password. The cost of an attack—or the work invested by an attacker—is the number of gates in the circuit, which we denote $|A|$. We use the notation $\Pr[v_1 \leftarrow S_1,\, v_2 \leftarrow S_2,\, \ldots\,;\, B]$ to denote the probability of statement B after an experiment in which variables $v_1, v_2, \ldots$ are drawn from probability distributions $S_1, S_2, \ldots$, respectively. Now we can define what it means for a password function to resist attack. We say that function $F(s,t)$ is an *ϵ-secure password function* if the following hold:

1. Finding partial information about F's secret input is as hard as guessing passwords. Put another way, for any password distribution D and predicate P, an attacker A who guesses P based on output from F will do almost as well when F is computed on unrelated passwords:

$$\begin{aligned}
&\forall D,\, \forall P,\, \forall A,\\
&\quad \Big|\Pr\big[t_1 \leftarrow T,\, \ldots,\, t_c \leftarrow T,\, s \leftarrow D,\\
&\qquad\quad b \leftarrow A(t_1, F(s,t_1), \ldots, t_c, F(s,t_c));\\
&\qquad\quad b = P(s)\big]\\
&\quad\; - \Pr\big[t_1 \leftarrow T,\, \ldots,\, t_c \leftarrow T,\, s \leftarrow D,\\
&\qquad\quad b \leftarrow A(t_1, F(s,t_1), \ldots, t_c, F(s,t_c)),\\
&\qquad\quad s' \leftarrow D;\, b = P(s')\big]\Big|\\
&\quad < \frac{\epsilon}{2} \cdot |A| \cdot R(D)
\end{aligned}$$

2. Finding second preimages is as hard as guessing passwords. (A second preimage of an input (s,t) is a different password $s' \neq s$ for which $F(s,t) = F(s',t)$.) Here we model the attacker A as a randomized circuit with multiple output bits:

$$\begin{aligned}
&\forall D,\, \forall A,\\
&\quad \Pr\big[t \leftarrow T,\, s \leftarrow D,\, s' \leftarrow A(s,t);\\
&\qquad\quad s \neq s' \wedge F(s,t) = F(s',t)\big]\\
&\quad < \epsilon \cdot |A| \cdot R(D)
\end{aligned}$$

[1]Boolean circuits are a complexity theoretic abstraction. A boolean circuit is an acyclic collection of interconnected gates. Each gate computes a boolean function of 0, 1 or 2 single-bit inputs. A randomized boolean circuit takes a certain number of random input bits in addition to its regular inputs.

We should first note that this definition matches our intuition about a password hashing function like *crypt*. If users choose predictable enough passwords, knowing a password hash gives adversaries a large advantage—they can compare hashes of the most popular passwords to that of the password they are trying to break. If, additionally, one can guess a useful predicate without even looking at a password hash—for instance by knowing that the third character of most passwords is a lower-case letter—then clearly an adversary can guess this too.

If, however, no single password occurs with particularly high probability, an adversary should need to expend a large amount of effort (as measured in circuit gates) to discover any non-trivial information about a password. Finally, we also wish to prevent an attacker from finding other strings that hash to the same value as a password; such strings may prove equivalent to passwords during authentication. The requirement of second preimage resistance guarantees such collisions are hard to find, even with knowledge of the original password. It also ensures that F does not ignore any bits of its password input.

The definition implies that a secure password function $F(s,t)$ must make non-trivial use of its second argument, t. To see this, consider that the first bit of $F(s,0)$ is a perfectly valid predicate on passwords. An attacker could easily guess this predicate if either F ignored its second argument or the string 0 occurred in T with high probability. This point is not merely an academic one. A single-input password hashing function $F(s)$ can be inverted by a circuit large enough to encode a lookup table mapping $F(s)$ (or sufficiently many bits of $F(s)$) to s. The size of such a circuit depends only on the probability distribution of the passwords, not on the particulars of F.

As proposed by Morris and Thompson [9], however, lookup tables can be thwarted with the second input to F, which they call a *salt*. If a random salt is chosen whenever users establish new passwords, and if the salt space is large enough to ensure a negligible probability of recurrence, lookup tables offer an adversary no advantage; he may as well compute F at the time of attack. If, on the other hand, the salt space is too small, the output bits of F become useful predicates on passwords, a fact exploited by the QCrack [12] program described in Section 6.

While salted passwords defeat lookup tables, given a particular salt and hash, an adversary can still mount a brute force attack by evaluating $F(s, t)$ on every possible password. It follows that the best security one can achieve is $\epsilon \approx 1/|F|$, where $|F|$ is the cost in gates of implementing F. Usability requirements therefore effect a lower limit on ϵ—if people can only tolerate a one second delay for checking passwords, F can take at most one second to evaluate. F should not take significantly less, however, as this would unnecessarily weaken security.

The number of gates $|A|$ that an adversary can reasonably muster for an attack increases constantly as hardware improves. Fortunately, so does the speed of machines that must legitimately evaluate F. That means passwords should not be hashed by a single function F with fixed computational cost, but rather by one of a family of functions with arbitrarily high cost. Instead of repeatedly throwing out functions like *crypt* and MD5 *crypt* to start over with more expensive but incompatible ones, systems should allow the cost of any password manipulation software to scale gracefully with a tunable parameter. Thus, ϵ can decrease as fast as hardware improves and users will tolerate. Compromised password databases will then enjoy maximum security against off-line attacks.

In summary, a good password function makes extracting any partial information about passwords as difficult as guessing passwords. A concrete parameter, ϵ, should characterize this difficulty. To achieve low values of ϵ, a password function must take a second input, the salt, that prevents adversaries from benefiting from large lookup tables. The best value of ϵ is inversely proportional to the cost of evaluating a password function. This establishes a lower limit for ϵ based on the maximum tolerable cost of evaluating F during legitimate use. As hardware speeds constantly improve, a good password scheme should allow the cost of F to increase gradually so that ϵ can decrease over time.

One final criterion for a good password function is then to minimize the value $\epsilon \cdot |F|$. That means one should make any password function as efficient as possible for the setting in which it will operate. The designers of *crypt* failed to do this. They based *crypt* on DES [10], a particularly inefficient algorithm to implement in software because of many bit transpositions. They discounted hardware attacks, in part because *crypt* cannot be calculated with stock DES hardware. Unfortunately, Biham [4] later discovered a software technique known as bitslicing that eliminates the cost of bit transpositions in computing many simultaneous DES encryptions. While bitslicing won't help anyone log in faster, it offers a staggering speedup to brute force password searches.

In general, a password algorithm, whatever its cost, should execute with near optimal efficiency in any setting in which it sees legitimate use, while offering little opportunity for speedup in other contexts. It should rely heavily on a CPU's fast instructions—for instance addition, bitwise exclusive-or, shifts, and memory access to state that fits in a processor's first level cache. Ideally these operations should all be portably accessible from high-level languages like C, so as to minimize the benefit of hand-coded assembly language implementations. Conversely, the algorithm should avoid operations like bit transposition on which customized hardware enjoys a large advantage.

A password function should also not lend itself to any kind of pipelined hardware implementation. It should permit relatively little speedup from any kind of precomputation—for instance, hashing 1,000 passwords with the same salt and hashing one password under 1,000 salts should each cost 1,000 times more than hashing a single password.

4 Eksblowfish Algorithm

We now describe a cost parameterizable and salted block cipher that we call *eksblowfish* for expensive key schedule blowfish. Eksblowfish is designed to take user-chosen passwords as keys and resist attacks on those keys. As its base we use the blowfish [15] block cipher by Schneier, which is well-established and has been fairly well analyzed.

Blowfish is a 64-bit block cipher, structured as a 16-round Feistel network [14]. It uses 18 32-bit subkeys, $P_1, \ldots, P_{18}$, which it derives from the encryption key. The subkeys are known collectively as the *P-Array*.

Blowfish encrypts by splitting a 64-bit input block

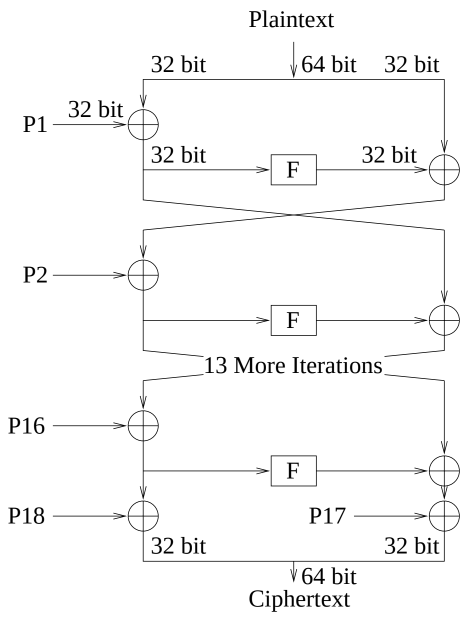

Figure 1: Blowfish Feistel network with F being the Feistel function, using only modular addition and XOR.

into two 32-bit halves, L_0 and R_0. The most-significant half, L_0, is XORed with subkey P_0, and used as input for a function F. The result of that function is XORed with the least-significant half, R_0. The two halves are then swapped, and the whole process repeated another 15 times for a total of 16 iterations. Thus, for $1 \le i \le 16$, letting $\oplus$ denote XOR:

$$\begin{aligned} R_i &= L_{i-1} \oplus P_i, \\ L_i &= R_{i-1} \oplus F(R_i). \end{aligned}$$

After 16 rounds, the two halves are swapped again (undoing the effect of the 16th swap), and each half is XORed with another 32-bit subkey:

$$\begin{aligned} R_{17} &= L_{16} \oplus P_{17}, \\ L_{17} &= R_{16} \oplus P_{18}. \end{aligned}$$

This process is illustrated graphically in Figure 1.

The function F in Blowfish uses four arrays, $S_1, \ldots, S_4$, derived from the encryption key. Each array contains 256 32-bit words. The arrays act as substitution boxes or *S-boxes*, replacing an 8-bit input with a 32-bit output. F splits its 32-bit input into four 8-bit bytes, a, b, c, and d, with a the most significant byte. It replaces each byte by the contents of an S-box, and combines the results as follows: Letting $\boxplus$ signify addition modulo 2^{32}:

$$F(a,b,c,d) = \big((S_1[a] \boxplus S_2[b]) \oplus S_3[c]\big) \boxplus S_4[d].$$

```
EksBlowfishSetup ($cost$, $salt$, $key$)
    $state \leftarrow$ InitState ()
    $state \leftarrow$ ExpandKey ($state$, $salt$, $key$)
    **repeat** ($2^{cost}$)
        $state \leftarrow$ ExpandKey ($state$, 0, $salt$)
        $state \leftarrow$ ExpandKey ($state$, 0, $key$)
    **return** $state$
```

Figure 2: Eksblowfish, expensive key schedule blowfish, is a cost parameterizable and salted variation of the blowfish block cipher.

Eksblowfish encrypts identically to Blowfish. The two differ in the functions they use to transform encryption keys into subkeys and S-boxes. Figure 2 sketches *EksBlowfishSetup*, the algorithm used by eksblowfish. *EksBlowfishSetup* has three input parameters: a cost, a salt, and the encryption key. It returns a set of subkeys and S-boxes, also known as a *key schedule*.

The cost parameter controls how expensive the key schedule is to compute. The salt is a 128-bit value that modifies the key schedule so that the same key need not always produce the same result, as motivated by Section 3. Finally, the key argument is a secret encryption key, which can be a user-chosen password of up to 56 bytes (including a terminating zero byte when the key is an ASCII string).

EksBlowfishSetup begins by calling *InitState*, a function that copies the digits of the number π first into the subkeys, then into the S-boxes.

ExpandKey(state, salt, key) modifies the P-Array and S-boxes based on the value of the 128-bit salt and the variable length key. First it XORs all the subkeys in the P-array with the encryption key. The first 32 bits of the key are XORed with P_1, the next 32 bits with P_2, and so on. The key is viewed as being cyclic; when the process reaches the end of the key, it starts reusing bits from the beginning to XOR with subkeys.

Subsequently, *ExpandKey* blowfish-encrypts the first 64 bits of its salt argument using the current state of the key schedule. The resulting ciphertext replaces subkeys P_1 and P_2. That same ciphertext is also XORed with the second 64-bits of salt, and the result encrypted with the new state of the key schedule. The output of the second encryption replaces subkeys P_3 and P_4. It is also XORed with the first 64-bits of salt and encrypted to replace P_5 and P_6. The process continues, alternating between the first and second 64 bits salt. When *ExpandKey* finishes replacing entries in the P-Array, it continues on replacing S-box entries two at a time. After replacing the last two entries of the last S-box, $S_4[254]$ and $S_4[255]$, *ExpandKey* returns the new key schedule.

In computing *ExpandKey(state, 0, key)*, a block of 128 0 bits is used instead of the salt. This is equivalent to a single iteration of the standard blowfish key schedule. The call to *ExpandKey(state, 0, salt)* simply treats the salt as a 16-byte key.

After calling *InitState* to fill a new key schedule with the digits of π, *EksBlowfishSetup* calls *ExpandKey* with the salt and key. This ensures that all subsequent state depends on both, and that no part of the algorithm can be precomputed without both salt and key. Thereafter, *ExpandKey* is alternately called with the salt and then key for 2^{cost} iterations. For all but the first invocation of *ExpandKey*, the second argument is a block of 128 0 bits. This more closely resembles the original blowfish key schedule, and also allows *EksBlowfishSetup* to be implemented more efficiently on CPU architectures with few registers.

We hope that the unpredictable and changing content of the P-array and S-Boxes will reduce the applicability of yet unknown optimizations. Additionally the *eksblowfish* S-Boxes require 4 KB of constantly accessed and modified memory. Thus, the S-Boxes cannot be shared across key schedules—separate S-Boxes must exist for every simultaneous execution. This vastly limits the usefulness of any attempts to pipeline the Feistel network in hardware.

5 Bcrypt Algorithm

The problems present in traditional UNIX password hashes led naturally to a new password scheme which we call *bcrypt*, referring to the Blowfish encryption algorithm. Bcrypt uses a 128-bit salt and encrypts a 192-bit magic value. It takes advantage of the expensive key setup in *eksblowfish*.

The *bcrypt* algorithm runs in two phases, sketched in Figure 3. In the first phase, *EksBlowfishSetup* is called with the cost, the salt, and the password, to initialize *eksblowfish*'s state. Most of bcrypt's time is spent in the expensive key schedule. Following that, the 192-bit value "OrpheanBeholderScryDoubt" is encrypted 64 times using *eksblowfish* in ECB mode with the state from the previous phase. The output is the cost and 128-bit salt concatenated with the result of the encryption loop.

```
bcrypt ($cost$, $salt$, $pwd$)
  $state$ ← EksBlowfishSetup ($cost$, $salt$, $key$)
  $ctext$ ← "OrpheanBeholderScryDoubt"
  repeat (64)
    $ctext$ ← EncryptECB ($state$, $ctext$)
  return Concatenate ($cost$, $salt$, $ctext$)
```

Figure 3: The bcrypt algorithm for hashing UNIX passwords, based on eksblowfish.

In Section 3, we derived that an ϵ-secure password function should fulfill several important criteria: second preimage-resistance, a salt space large enough to defeat precomputation attacks, and an adaptable cost. We believe that *Bcrypt* achieves all three properties, and that it can be ϵ-secure with useful values of ϵ for years to come. Though we cannot formally prove *bcrypt* ϵ-secure, any flaw would likely deal a serious blow to the well-studied blowfish encryption algorithm.

5.1 Implementation

We have implemented *bcrypt* and deployed it as part of the OpenBSD operating system. *Bcrypt* has been the default password scheme since OpenBSD 2.1.

An important requirement of any *bcrypt* implementation is that it exploit the full 128-bit salt space. OpenBSD generates the 128-bit *bcrypt* salt from an arcfour (*arc4random(3)*) key stream, seeded with random data the kernel collects from device timings.

OpenBSD lets administrators select a password hashing scheme through a special configuration file, *passwd.conf(5)*. *passwd.conf* allows detailed control over which type of password to use for a given user or group. It also permits different password schemes for local and YP passwords. For *bcrypt*, one can also specify the cost. This lets people adjust password verification time for increasing processor speed. At the time of publication, the default cost is 6 for a normal user and 8 for the superuser. Of course, whatever cost people choose should be reevaluated from time to time.

To differentiate between passwords hashed by different algorithms, every password function other than the original *crypt* prefixes its output with a version identifier. Thus a single password file can contain several types of password. In the current OpenBSD implementation, *bcrypt* passwords start with "$2a$", while MD5 *crypt* passwords with "1." Because the result of traditional *crypt* never begins with a "$", there is never any ambiguity.

6 Bcrypt Evaluation

Because *bcrypt* has adjustable cost, we cannot meaningfully evaluate the performance of the algorithm on its own. Instead, we will place it in the context of two popular password hashing functions. We describe various attacks and optimizations these functions have undergone, and discuss the applicability of the same techniques to *bcrypt*.

6.1 Comparison

In the following, we give a brief overview of two password hashing functions in widespread use today, and state their main differences from *bcrypt*.

6.1.1 Traditional crypt

Traditional *crypt(3)*'s design rationale dates back to 1976 [9]. It uses a password of up to eight characters as a key for DES [10]. The 56-bit DES key is formed by combining the low-order 7 bits of each character in the password. If the password is shorter than 8 characters, it is padded with zero bits on the right.

A 12-bit salt is used to perturb the DES algorithm, so that the same password plaintext can produce 4,096 possible password encryptions. A modification to the DES algorithm, swapping bits i and $i+24$ in the DES E-Box output when bit i is set in the salt, achieves this while also making DES encryption hardware useless for password guessing.

The 64-bit constant "0" is encrypted 25 times with the DES key. The final output is the 12-bit salt concatenated with the encrypted 64-bit value. The resulting 76-bit value is recoded into 13 printable ASCII characters.

At the time traditional *crypt* was conceived, it was fast enough for authentication but too costly for password guessing to be practical. Today, we are aware that it exhibits three serious limitations: the restricted password space, the small salt space, and the constant execution cost. In contrast, *bcrypt* allows for longer passwords, has salts large enough to be unique over all time, and has adaptable cost. These limitiations therefore do not apply to *bcrypt*.

6.1.2 MD5 crypt

MD5 *crypt* was written by Poul-Henning Kamp for FreeBSD. The main reason for using MD5 was to avoid problems with American export prohibitions on cryptographic products, and to allow for a longer password length than the 8 characters used by DES *crypt*. The password length is restricted only by MD5's maximum message size of 2^{64} bits. The salt can vary from 12 to 48 bits.

MD5 *crypt* hashes the password and salt in a number of different combinations to slow down the evaluation speed. Some steps in the algorithm make it doubtful that the scheme was designed from a cryptographic point of view—for instance, the binary representation of the password length at some point determines which data is hashed, for every zero bit the first byte of the password and for every set bit the first byte of a previous hash computation.

The output is the concatenation of the version identifier "1", the salt, a "$" separator, and the 128-bit hash output.

MD5 *crypt* places virtually no limit on the size of passwords, while *bcrypt* has a maximum of 55 bytes. We do not consider this a serious limitation of *bcrypt*, however. Not only are users unlikely to choose such long passwords, but if they did, MD5 *crypt*'s 128-bit output size would become the limiting factor in security. A brute force attacker could more easily find short strings hashing to the same value as a user's password than guess the actual password. Finally, like DES *crypt*, MD5 *crypt* has fixed cost.

6.2 Attacks and Vulnerabilities

Once an attacker has obtained a list of password hashes, passwords can be guessed by comparing the target list to a list of hashes of candidate passwords.

n	10 digits		26 lowercase		36 lowercase alphanumeric		52 mixed case		62 mixed case alphanumeric		95 keyboard characters	
4	0.04	sec	1.9	sec	7	sec	30.5	sec	61.6	sec	5.7	min
5	0.4	sec	49.5	sec	4.2	min	26.4	min	1.1	hours	9	hours
6	4.2	sec	21.5	min	2.5	hours	22.9	hours	2.7	days	35.5	days
7	41.6	sec	9.3	hours	3.8	days	49.6	days	169	days	9.2	years
8	6.9	min	10	days	136	days	7	years	28.8	years	875	years
9	1.2	hours	261	days	13.4	years	366	years	1786	years	83180	years

Table 1: Time required to test a single password against a specified password space when being able to perform 240,000 evaluations of *crypt* per second. Password spaces above the separation can be searched completely within 4 days. However this does not imply that passwords chosen from below the separation are secure against password guessing. These times are normal for traditional *crypt* nowadays.

This task is facilitated by the fact that users tend to choose predictable passwords. In the following, we will present commonly used techniques and evaluate how they affect the security of *bcrypt*.

The most common method is known as a dictionary attack. It is based on the knowledge that many users choose their passwords in a very predictable way. Often a user's password can be found in a dictionary or is the name of a close relative with small modifications, *e.g.,* "Susan1" or "neme$i$". The attacker compiles a list of common names and words. For a given salt, the words in the list are hashed with the password scheme and compared with entries of the same salt in the password file. If there is a match, the plaintext password has been found.

Commonly, lists of likely passwords contain hundreds of thousands of words. A dictionary attack is only feasible when the one-way function can be computed quickly. *Bcrypt*'s cost can be made as high as tolerable by legitimate users, rendering dictionary attacks impractically slow.

6.2.1 Salt Collisions

A *salt collision* occurs when two password encodings use the same salt. Ideally, there should be no *salt collisions*—the salts of different password encodings should be different even across password files. Because traditional *crypt* uses only 4,096 different salts, it leads to many collisions, as illustrated in Figure 4. To optimize dictionary attacks, an attacker can group encrypted passwords by salt, and hash each candidate password from a dictionary only once per salt. The resulting speedup can roughly be determined as

$$\frac{\text{number of passwords}}{\text{number of different salts}}.$$

If salts are generated with a good random number generator, the expected number of different salts for n password entries with s possible salts is

$$EV(n,s) = \sum_{i=0}^{n-1} \left(\frac{s-1}{s}\right)^i = s - (s-1)^n s^{1-n}.$$

In a 15,000 entry password file, a space of 2^{41} salts ensures with high probability that every salt will be unique. For 2^{12} possible salts, on the other hand, we can only expect about 3,991 different salts. At 2^{24} possible salts, the number becomes 14,994. In practice, however, we find that the number of salt collisions is generally higher than expected. The reason is that many operating systems generate poor random numbers.

6.2.2 Precomputing Dictionaries

Using precomputation, an attacker can build a list of the hashes of every common password under every possible salt, and store this list on mass data storage. Inverting the hash of a common password then becomes a simple lookup in a database, with little computational cost.

The 1934 edition of the Webster Dictionary contains, after truncation to 8 characters and duplicate removal, 171,395 unique entries. Using standard

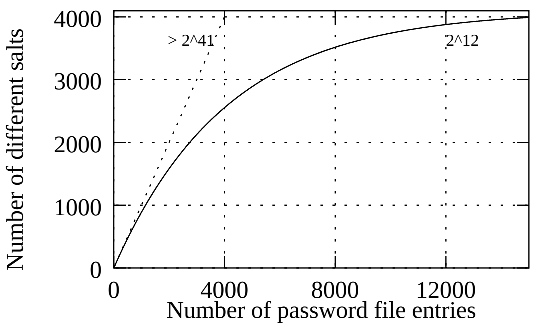

Figure 4: Distribution of expected different salts depending on the salt space against the number of entries in a password file.

crypt, the result of hashing every dictionary word under every possible 12-bit salt would fit on a single 9 GB hard disk.

One can do better, however, by storing less than the full output of *crypt* in a database. The QCrack [12] password cracking program takes exactly this approach. QCrack precomputes a database of common passwords hashed under every salt. Rather than store the full 13 character output of *crypt*, it further hashes *crypt*'s output down to a single byte. When cracking a password from the dictionary, QCrack uses the database to rule out 255 of every 256 candidate passwords without needing to compute their hashes. A QCrack database of the Webster Dictionary consumes only 670 MB. QCrack could store hashes of approximately $2,350,000$ words on a 9 GB hard disk.

Bcrypt has a large enough salt space to make storing even one bit of information per salt completely intractable. Moreover, the algorithm makes immediate use of the password and salt from the very beginning. Thus, before knowing a target password's salt, there is not even an intermediary state of the algorithm that can be usefully precomputed.

6.2.3 Algorithm Optimization

Since a guessing attack on a password function involves repeated evaluation of the function, any optimization of the function will reduce the computational cost of an attack, making the attack more practical.

Biham recently discovered a notable software optimization of DES which he called *bitslicing* [4]. By replacing DES's S-Boxes with a logic gate circuit, one can reduced DES to a set of bit operations. One can then treat a 64-bit processor as 64 parallel one-bit processors, each implementing the circuit.

On a 300MHz Alpha 8400 processor, Biham gained about a factor of 5 speedup using bitsliced DES. His implementation encrypted 137 Mb/sec on average, compared to Eric Young's libdes, which achieved only 28 Mb/sec.

For MD5 *crypt* the situation is similar. In "John the Ripper" [5] a considerable speedup was made by simplifying MD5 *crypt*'s central computing loop.

Bitslicing relies on the fact that DES's S-boxes are fixed and well known. In contrast, *Bcrypt*'s S-boxes change constantly over the course of the algorithm, and take on different values for every combination of password and salt. Bitslicing cannot be applied to *bcrypt*.

6.2.4 Hardware Improvements

In 1977 on a VAX-11/780, *crypt* could be evaluated about 3.6 times per second. In the last 20 years, machine speed has increased dramatically and the algorithm has been optimized in various ways.

The Electronic Frontier Foundation built a DES cracker in 1998 and was able to crack a 56-bit key in 56 hours with an average search rate of about $88 \cdot 10^9$ keys per second [6]. While the EFF DES cracker cannot be used for password guessing, a comparable machine could crack traditional *crypt* by brute force in 22 days, compared to 875 years on the fastest alpha processor to which we had access.

The impact of increasing processor speed and better optimization of the password hashing algorithm is shown in Figure 5.

Both traditional and MD5 *crypt* operate with a fixed number of rounds. On a modern Alpha processor, traditional *crypt* can already be computed fast enough to render it unusable with readable password files. When using specialized DES hardware, the computing time can be reduced again by several orders of magnitude.

Neither traditional nor MD5 *crypt* support a vari-

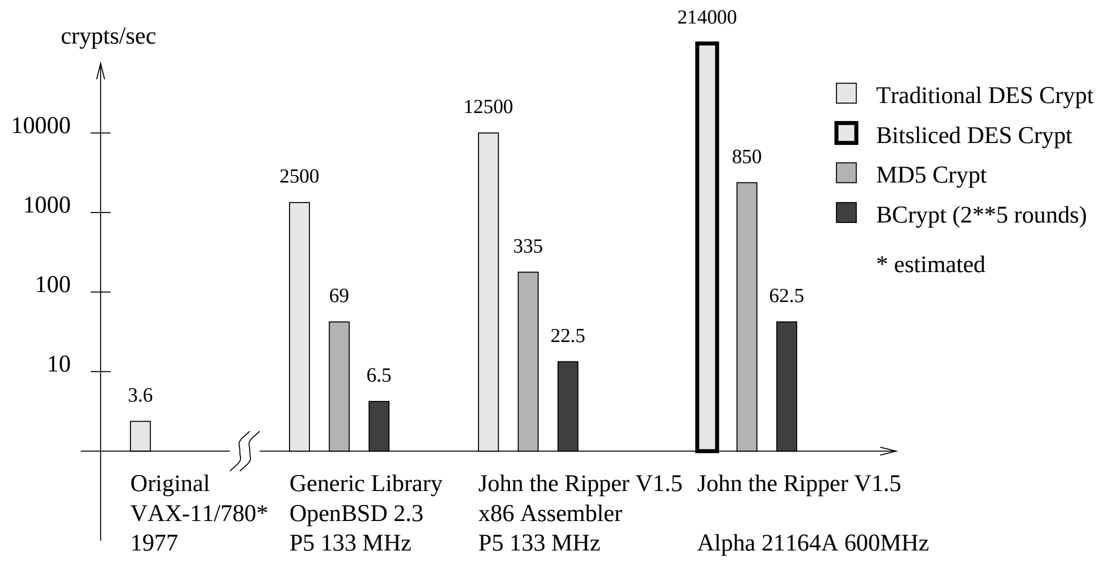

Figure 5: Impact of Algorithm Optimization and Advance in Processors

able number of rounds. With increasing processing power, these functions become steadily easier to compute. In contrast, *bcrypt* will adapt to more powerful attackers. Moreover, its inner loop relies exclusively on operations that are efficient on general-purpose CPUs, leaving little opportunity for specialized hardware to achieve dramatic improvements.

7 Conclusion

Many authentication schemes depend on secret passwords. Unfortunately, the length and entropy of the passwords users choose remain fixed over time. In contrast, hardware constantly improves, giving attackers increasing computational power. As a result, password schemes (including the traditional UNIX user-authentication system) are failing to withstand off-line password guessing attacks.

In this paper, we formalize the notion of a password scheme "as good as the passwords users choose," and show that the computational cost of such a scheme must necessarily increase with the speed of hardware. We propose two algorithms of parameterizable cost for use with passwords. *Eksblowfish*, a block cipher, lets one safely store encrypted private keys on disk. *Bcrypt*, a hash function, can replace the UNIX password hashing function or serve as a front-end to secure password protocols like SRP. We have deployed *bcrypt* as part of the OpenBSD operating system's password authentication. So far, it compares favorably to the two previous hashing algorithms. No surprise optimizations have yet turned up. As hardware speeds increase, OpenBSD lets one preserve the cost of off-line password cracking by tuning a simple configuration file.

8 Acknowledgments

We thank Solar Designer for helpful discussions on optimization and flaws of password schemes. We further thank Angelos D. Keromytis, Peter Honeyman, Robert T. Morris, and Frans Kaashoek for remarks and suggestions.

References

[1] Martín Abadi, T. Mark A. Lomas, and Roger Needham. Strengthening passwords. Technical note 1997-033, DEC Systems Research Center, September 1997.

[2] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1992.

[3] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 224–250, Oakland, CA, November 1993.

[4] Eli Biham. A Fast New DES Implementation in Software. In *Fast Software Encryption, 4th In-*

ternational Workshop Proceedings, pages 260–271. Springer-Verlag, 1997.

[5] Solar Designer. John the Ripper. `http://www.false.com/security/john`.

[6] Electronic Frontier Foundation. *Cracking DES*. O'Reilly and Associates, 1998.

[7] Li Gong, T. Mark A. Lomas, Roger M. Needham, and Jerome H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, June 1993.

[8] Shai Halevi and Hugo Krawczyk. Public-key cryptography and password protocols. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, 1998.

[9] Robert Morris and Ken Thompson. Password Security: A Case History. *Communications of the ACM*, 22(11):594–597, November 1979.

[10] National Bureau of Standards. Data Encryption Standard, January 1977. FIPS Publication 46.

[11] Sarvar Patel. Number theoretic attacks on secure password schemes. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 236–247, Oakland, CA, May 1997.

[12] QCrack. `ftp://chaos.infospace.com/pub/qcrack/qcrack-1.02.tar.gz`.

[13] R. L. Rivest. The MD5 Message Digest Algorithm. RFC 1321, Apr 1992.

[14] Michael Ruby. *Pseudorandomness and Cryptographic Applications*. Princeton Computer Science Notes, 1996.

[15] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.

[16] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[17] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, July 1996.