Time-lock puzzles and timed-release Crypto

Ronald L. Rivest*, Adi Shamir**, and David A. Wagner***

Revised March 10, 1996

*MIT Laboratory for Computer Science
545 Technology Square, Cambridge, Mass. 02139

**Weizmann Institute of Science
Applied Mathematics Department
Rehovot, Israel

***Computer Science Department
U.C. Berkeley
Berkeley, California 94720

{rivest,shamir}@theory.lcs.mit.edu, daw@cs.berkeley.edu

1 Introduction

Our motivation is the notion of "timed-release crypto," where the goal is to encrypt a message so that it can not be decrypted by anyone, not even the sender, until a pre-determined amount of time has passed. The goal is to "send information into the future." This problem was first discussed by Timothy May [6].

What are the applications of "timed-release crypto"? Here are a few possibilities (some due to May):

- A bidder in an auction wants to seal his bid so that it can only be opened after the bidding period is closed.
- A homeowner wants to give his mortgage holder a series of encrypted mortgage payments. These might be encrypted digital cash with different decryption dates, so that one payment becomes decryptable (and thus usable by the bank) at the beginning of each successive month.
- An individual wants to encrypt his diaries so that they are only decryptable after fifty years.

- A key-escrow scheme can be based on timed-release crypto, so that the government can get the message keys, but only after a fixed period (say one year).

There are presumably many other applications.

There are two natural approaches to implementing timed-release crypto:

- Use "time-lock puzzles"–computational problems that can not be solved without running a computer continuously for at least a certain amount of time.
- Use trusted agents who promise not to reveal certain information until a specified date.

Using trusted agents has the obvious problem of ensuring that the agents are trustworthy; secret-sharing approaches can be used to alleviate this concern. Using time-lock puzzles has the problem that the CPU time required to solve a problem can depend on the amount and nature of the hardware used to solve the problem, as well as the parallelizability of the computational problem being solved.

In this note we explore both approaches. (We note that Tim May has suggested an approach based on the use of trusted agents.)

2 Time-lock puzzles

We first explore an approach based on computational complexity: we study the problem of creating computational puzzles, called "time-lock puzzles," that require a precise amount of time to solve. The solution to the puzzle reveals a key that can be used to decrypt the encrypted information. This approach has the obvious problem of trying to make "CPU time" and "real time" agree as closely as possible, but is nonetheless interesting.

The major difficulty to be overcome, as noted above, is that those with more computational resources might be able to solve the time-lock puzzle more quickly, by using large parallel computers, for example. Our goal is thus to design time-lock puzzles that, to the greatest extent possible, are "instrinsically sequential" in nature, and can not be solved substantially faster with large investments in hardware. In particular, we want our puzzles to have the property that putting computers to work together in parallel doesn't speed up finding the solution. (Solving the puzzle should be like having a baby: two women can't have a baby in 4.5 months.) We propose an approach to building puzzles that appears to be intrisically sequential in the desired manner.

Of course, our approach yields puzzles with a solution time that is only *approximately* controllable, since different computers work at different speeds. For example, the underlying technology may be different: gallium arsenide gates are faster than silicon gates. If precise timing of the information release is essential, an approach based on the use of trusted agents is preferable.

We also note that with our approach, the puzzle doesn't automatically become solvable at a given time; rather, a computer needs work continuously on the puzzle until it is solved. A ten-year puzzle needs some dedicated workstation working away for ten years to solve it. If the computing doesn't start until five years after the puzzle was made, then the solution won't be found until ten years after that (perhaps a bit less if technology has improved in the meantime). Our approach therefore requires much more in the way of computational resources than an approach based on trusted agents, and thus may be best suited for relatively simple puzzles (with time-to-solution under a month, say). Nonetheless, we feel that our approach has sufficient utility to merit this exposition.

An unworkable approach

We begin by presenting an approach that *doesn't* work well. Let M denote the information to be encrypted for a period of time. Let S denote the speed of a workstation measured in decryptions per second. Then to encrypt M to be decryptable after T seconds, we choose a conventional cryptosystem (say RC5 [9]) with a key size of approximately $k = \lg(2ST)$ bits and encrypt M with a k-bit key. We save the ciphertext and throw away the key. By using exhaustive search of the key space, a workstation will take about T seconds, on the average, to find the key.

We note that Merkle [7] was the first to suggest this method of designing puzzles, and was also the first to introduce the notion of a "puzzle," in research that ultimately led to the invention of the concept of public-key cryptography.

There are two problems with this way of building a time-lock puzzle by encrypting M with a conventional cipher:

- A brute-force key-search is trivially parallelizable, so that N computers make the computation run N times faster.
- The computation time estimate of T seconds is only an expected running time; the actual running time could be signicantly larger or smaller, depending on the order in which the keys are examined.

These problems are fixed in the proposal given next.

2.1 Creating a time-lock puzzle

We now show a method for creating time-lock puzzles based on repeated squaring. Our approach can also be viewed as an application of the "random-access" property of the Blum-Blum-Shub "$x^2 \bmod n$" pseudo-random number generator [3]. (We actually propose a scheme that is a variation on the $x^2 \bmod n$ generator, but the differences are nonessential, and the original scheme could have been used as well here.) An early version of our paper suggested a different approach based on superencryption in RSA [10, 12, 8, 13, 2]; the current approach is considerably simpler.

Here is our approach. Suppose Alice has a message M that she wants to encrypt with a time-lock puzzle for a period of T seconds.

- She generates a composite modulus

  $$n = pq \tag{1}$$

  as the product of two large randomly-chosen secret primes p and q. She also computes

  $$\phi(n) = (p-1)(q-1) \; . \tag{2}$$

- She computes
$$t = TS \ , \tag{3}$$
where S is the number of squarings modulo n per second that can be performed by the solver.

- She generates a random key K for a conventional cryptosystem, such as RC5. This key is long enough (say 160 bits or more) that searching for it is infeasible, even with the advances in computing power expected during the lifetime of the puzzle.

- She encrypts M with key K and encryption algorithm RC5, to obtain the ciphertext
$$C_M = RC5(K, M) \ . \tag{4}$$

- She picks a random a modulo n (with $1 < a < n$), and encrypts K as
$$C_K = K + a^{2^t} \pmod{n} \ . \tag{5}$$
To do this efficiently, she first computes
$$e = 2^t \pmod{\phi(n)} \ , \tag{6}$$
and then computes
$$b = a^e \pmod{n} \ . \tag{7}$$

- She produces as output the time-lock puzzle (n, a, t, C_K, C_M), and erases any other variables (such as p, q) created during this computation.

(We add as a technical footnote here the remark that p, q, and a can be chosen carefully, so that 2 is guaranteed to have a large order modulo $\phi(n)$, and so that a is guaranteed to have a large order modulo n. See Blum, Blum, and Shub [3] for some relevant discussion. However, choosing p, q, and a randomly should give the desired level of difficulty with overwhelming probability, so that these precautions are not expected to be necessary in practice. Indeed, in practice choosing a fixed value $a = 2$ should be safe with high probability. Since there are other risks in the whole approach (e.g. an adversary could just guess K), aiming for perfection in the number-theory is probably overkill.)

2.2 Solving the puzzle

By design, searching for the RC5 key K directly is infeasible, so the fastest known approach to solving the puzzle is to determine
$$b = a^{2^t} \pmod{n} \tag{8}$$
somehow. Knowing $\phi(n)$ enables 2^t to be reduced efficiently to e, modulo $\phi(n)$, so that b can be computed efficiently by equation (7). However, computing $\phi(n)$ from n is provably as hard as factoring n, so that once Alice publishes the puzzle and throws away the key (throws

away the factors p and q), there seems to be no faster way of computing b than to start with a and perform t squarings sequentially (each time squaring the previous result).

While factoring n is certainly an alternative attack for solving the puzzle, when p and q are large enough the factoring approach is far less efficient than repeated squaring.

The number t of squarings required to solve the puzzle can be exactly controlled. Thus, we can create puzzles of various desired levels of difficulty.

More importantly, repeated squaring seems to be an "intrinsically sequential" process. We know of no obvious way to parallelize it to any large degree. (A small amount of parallelization may be possible *within* each squaring.) Having many computers is no better than having one. (But having one fast computer is better than one slow one.) The degree of variation in how long it might take to solve the puzzle depends on the variation in the speed of single computers, and not on one's total budget. Since the speed of hardware available to individual consumers is within a small constant factor of what is available to large intelligence organizations, the difference in time to solution is reasonably controllable. (We admit that more control here might be desirable, but with a complexity-based approach such as this one there is not much that can be done to compensate for different gate speeds.)

3 Using trusted agents

A natural approach is to use a trusted agent to store the message M until its desired release time t. As an extension of this idea, the message M could be shared among several agents (using standard secret-sharing techniques, such as the one proposed by Shamir [11]) who all agree to release their shares at time t. The message M can then be reconstructed from those shares. As a further refinement, the agents can be asked to store shares of a cryptographic key K instead of shares of M. This reduces the storage demands on the agents. Then the encryption $C = E(K, M)$ of M with key K can be kept in some publicly available location. At time t, the key K can be reconstructed and C decrypted to yield M. These ideas are discussed briefly by May. Related work on time-lock puzzles and "verifiable partial-key escrow" has been developed by Bellare and Goldwasser [4, 1].

We suggest here an alternative, but related, approach that has the following properties and implementation:

- The agents are not "escrow agents" as they are in May's proposal: they do not have to store *any* information that is given to them by the user. The amount of storage required for an agent is fixed and bounded, independent of the number of timed-release user secrets that he has been asked to help out with.

- The main task of an agent is to periodically (say at the beginning of each hour) publish a previously secret value. We let s_{it} denote the secret published by agent i at time t. The agent will digitally sign all secrets s_{it} he publishes, using some standard digital signature scheme.

- The only other task that an agent must perform is to respond to requests of the form: "Here are values for y and t; please return $E(s_{it}, y)$, the encryption of y under the secret key s_{it} that you will reveal at future time t." The agent will only perform encryptions (never decryptions). It is assumed that the encryption algorithm is secure against chosen-message attacks, so that an adversary can obtain many encryptions of various y's with some future s_{it} and will not be able to thereby deduce s_{it}. Having received the request, the agent will return an encrypted digitally signed copy of the message

$$(i, t, t_0, E(s_{it}, y))$$

where i is the index of the agent, t is the future time requested, t_0 is the current time (by the agent's clock), and $E(s_{it}, y)$ is the requested ciphertext. The message is encrypted with the public key of the requestor and then signed with the agent's private key. The agent need not require that $t > t_0$, although this will be the normal case.

- Anyone can set himself up in business as a trusted agent, without requiring coordination between himself and other agents. More precisely, the sequence of secrets published by one agent is independent of the sequence of secrets published by any other agent.

- The sequence of secrets published by each agent has the property that from s_{it} one can easily compute $s_{it'}$ for all $t' \leq t$. The secret the agent reveals at time t can be used to compute all of his previously published secrets. Thus, it suffices to ask an agent for his latest secret in order to learn all of his previously published secrets. This can be easily implemented by having the secrets satisfy a recurrence such as:

$$s_{i(t-1)} = f(s_{it}) \tag{9}$$

for some suitable (but otherwise arbitrary) one-way function f. Because f is one-way, publishing s_{it} does not reveal any future secrets $s_{it''}$ for $t'' > t$. (The agent might precompute his sequence of secrets, beginning with a randomly chosen secret for some point in the distant future and working backwards, or he might chose f as a trap-door one-way function, so that only he can compute s_{it} from $s_{i(t-1)}$.)

- The message M to be released at time t is encrypted with a randomly chosen key K and a conventional encryption algorithm, to yield a ciphertext $C = E(K, M)$. The user picks some number d of agents i_1, i_2, …, i_d, and publishes

$$(C, i_1, i_2, \ldots, i_d, r_1, r_2, \ldots, r_d) \tag{10}$$

where r_1, r_2, …, r_d are d "timed-release shares" of the key K that will allow K to be reconstructed once time t is reached and the agents publish their secrets for time t.

- The user may pick a threshold θ (where $0 < \theta \leq d$) such that one can reconstruct K given θ or more time-release shares and the corresponding agents' secrets for time t. To accomplish this, the user splits K into d shares

$$y_1, y_2, \ldots, y_d \tag{11}$$

according some standard secret-sharing scheme with threshold θ, and then asks agent i_j (for $1 \leq j \leq d$) to produce the value

$$r_j = E(s_{i_j t}, y_j) \ , \tag{12}$$

the encryption of share y_j of K with the secret $s_{i_j t}$ of agent i_j that will be revealed at time t. This request should be encrypted with the public key of the agent, and the reply should be encrypted and signed as described earlier.

The agents in this scheme are extremely simple: they only need

- to produce an unpredictable sequence of secrets satisfying equation (9)),
- to decrypt a message of the form $(y, t, (e, n))$ encrypted with the public key of the agent,
- to encrypt values y under the secret s_{it} to be revealed by the agent at time t,
- to return the resulting ciphertext, signed by the agent and then encrypted with the public key (e, n) of the requestor, and
- to publish a signed version of s_{it} at time t.

Since such a simple agent could be built into a small tamper-proof device quite easily, one can produce implementations of such agents that are highly secure.

The fact that the scheme is based on secret-sharing with a threshold gives robustness, both against the possible corruption of one or more agents (who might sell future values of their secrets) or the death or disappearance of one or more agents. As long as θ agents are still around at time t, the message M will be reconstructable at time t (and at any later time). As long as fewer than θ agents have been corrupted, the message M will not be revealed before time t.

This scheme is not "verifiable" in the sense that an observer who sees the published material of equation (10) can not verify that it is the proper encryption of anything particular. Only when the secrets of time t are published can he decrypt the shares r_j to obtain the corresponding y_j values that allow him to reconstruct K, and thus obtain M. Standard "verifiable" secret-sharing techniques aren't particularly applicable here, since the message M could be junk, even if K was verifiably shared. (We note that in principle, it is possible, albeit difficult, to prove certain properties of M to a verifier without having to reveal K or M.)

Because the agent includes the current time t_0 in his signed reply to an encryption request, he acts as a simple "time-stamping" service (e.g. [5]). A user can give the agent the cryptographic hash value $h(M)$ of some message M, and ask the agent to sign and encrypt it with s_{it} for some value of t. The signed hash value becomes decryptable at time t, thus proving (assuming that the agent is trustworthy) that the document M existed at time t_0. Normally one might have $t = t_0$, but a user might choose $t > t_0$ in some cases. For example, in an auction it may be required that the bids be submitted before some time t', and that they be opened at time t''. The user would submit (the encryption key K for) his bid at time $t_0 < t'$, and ask for it to be encrypted with s_{it} where $t = t''$.

3.1 An off-line version

The previous protocol can be converted to an offline protocol, as follows. Each trusted agent constructs a public/private keypair $E_{i,t}, D_{i,t}$ for each future time t. The public key

$E_{i,t}$ is published immediately, and the private key $D_{i,t}$ is published at time t. (Of course, a trusted agent always digitally signs the published E's and D's under his master public key, to eliminate would-be imposters.)

The E's and D's directly replace the s's: now the user can perform the encryption of the y's himself, without needing to invoke the trusted agent. The trusted agent can now be entirely offline, except for the periodic publication of the D's.

On the other hand, in this offline formulation, it seems hard to encode any structure into the agent's keys, so it seems to require more storage to store the list of public keys for the future and the private keys revealed for the past. At 200 bytes per key, storing one key for each day of the next fifty years requires about 3.6 megabytes.

Another disadvantage of this off-line approach is that the agents are no longer usable or available as "time-stamping" agents.

4 Conclusions

We have suggested a way to create "time-lock puzzles," which require (approximately) a certain amount of time (real time, not total CPU time) to solve. We have also discussed a way to use trusted agents to efficiently enable timed-release crypto.

References

[1] Mihir Bellare and Shafi Goldwasser. Verifiable partial key escrow. Technical Report CS95-447, Dept. of Computer Science and Engineering, U.C. San Diego, October 1995.

[2] Shimshon Berkovits. Factoring via superencryption. *Cryptologia*, 6(3):229–237, July 1982.

[3] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM J. Computing*, 15(2):364–383, May 1986.

[4] Shafi Goldwasser, 1996. Personal communication.

[5] S. Haber and W.S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3:99–111, 1991.

[6] Timothy C. May. Timed-release crypto, February 1993. `http://www.hks.net/cpunks/cpunks-0/1460.html`.

[7] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21:294–299, April 1978.

[8] Ronald L. Rivest. Remarks on a proposed cryptanalytic attack of the M.I.T. public-key cryptosystem. *Cryptologia*, 2(1):62–65, January 1978.

[9] Ronald L. Rivest. The RC5 encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption*, pages 86–96. Springer, 1995. (Proceedings Second International Workshop, Dec. 1994, Leuven, Belgium).

[10] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[11] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

[12] Gustavus J. Simmons and Michael J. Norris. Preliminary comments on the MIT public-key cryptosystem. *Cryptologia*, 1(4):406–414, October 1977.

[13] H. C. Williams and B. Schmid. Some remarks concerning the MIT public-key cryptosystem. *BIT*, 19:525–538, 1979.