In Dependable Computing for Critical Applications, eds. A. Avizienis and J. Laprie. Springer, 1991, pp. 283-300

IMPLEMENTING DISTRIBUTED CAPABILITIES WITHOUT A TRUSTED KERNEL

Maurice P. HERLIHY [1] *, J. D. TYGAR*
Computer Science Dept., Carnegie Mellon University
Pittsburgh, PA 15213 - USA

Abstract

Capabilities are well-known to be a simple and efficient technique for implementing protection in centralized systems. In decentralized distributed systems, however, implementing capabilities can be considerably more difficult. Two problems stand out: (1) how to communicate information about capabilities across an insecure communication network, and (2) how to revoke capabilities in the presence of failures such as message delays, crashes, and network partitions. This paper describes a new scheme for managing capabilities in a distributed system that incorporates novel solutions to both problems. The communication problem is addressed by a new and efficient protocol that exploits recent developments in "zero-knowledge" authentication protocols. The revocation problem is solved by new protocols that rely on approximately synchronized real-time clocks to create the illusion that revocation occurs instantaneously, even in the presence of failures.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order Numbers 4976, monitored by the Air Force Avionics Laboratory under Contracts F33615-84-K-1520. J.D. Tygar received additional support from National Science Foundation Presidential Young Investigator Grant CCR-8858087.

---

[1] Author's current address: Digital Equipment Corporation, Cambridge Research Laboratory, One Kendall Square, Cambridge, MA 02139

1. Introduction

Capabilities are well-known to be a simple and efficient technique for implementing protection in centralized systems. In decentralized distributed systems, however, implementing capabilities can be considerably more difficult. Two problems stand out: (1) how to communicate information about capabilities across an insecure communication network, and (2) how to revoke capabilities in the presence of failures such as message delays, crashes, and network partitions. This paper describes a new scheme for managing capabilities in a distributed system that incorporates novel solutions to both problems. The communication problem is addressed by a new and efficient protocol that exploits recent developments in "zero-knowledge" authentication protocols. The revocation problem is solved by new protocols that rely on approximately synchronized real-time clocks to create the illusion that revocation occurs instantaneously, even in the presence of failures.

2. Model

A *distributed system* consists of a collection of computers, called sites, that are geographically distributed and connected by a communications network. A distributed program is organized as a collection of modules, each of which resides at a single site. Modules communicate through messages, acting as clients and as servers. A *client* makes use of services provided by other modules, while a *server* provides services to others by encapsulating a resource, providing synchronization, protection, and crash recovery. The client/server model is hierarchical: a particular module may be both a client and a server.

Not every client is allowed access to every server. Each client has a set of rights, where a right is permission to make use of services provided by certain servers. A client may delegate those rights to other clients, and rights may be revoked. A well-known way to implement such protection is to associate each right with a *capability*, a token which is produced by a client having that right. Certain priviledged servers are allowed to create capabilities and to distribute them to deserving clients. The server that created the capabilities is called their *owner*. In addition, clients may delegate rights by exchanging capabilities among themselves. A capability's owner may revoke clients' rights by rendering that capability invalid. If client A has transferred a capability to client B, then if A's right to use that capability is revoked, so is B's. It has long been recognized that capability systems are as powerful as full access control matrices for protection

[17]. For this reason, numerous operating systems have used capability mechanisms as the basic means of implementing protection [36, 6, 8, 18, 35].

In a distributed environment, capability systems have typically been implemented by storing capability based information in a distributed kernel, or by having trusted kernels exchange information about capabilities [37, 8] . By contrast, in this paper, we propose a capability scheme where each client stores its own authentication information. When a client wishes to begin a transaction with a server, the client must first present the security information to the server. We call this approach *self-securing*, since each client and server is responsible for monitoring security concerns, and not the underlying operating system kernel. In this paper, we do not presume that the data network is secure; neither do we presume that the base operating system provides security other than address space protection. (We do not, however, attempt to address denial of service problems here. For some theoretical contributions to this problem, see [13].) This paper addresses three problems: (1) What form do capabilities take and how are they distributed? (2) How can we guarantee that individual clients will not forge or steal capabilities? (3) How can capabilities be revoked?

To illustrate the issues involved, consider the following unsatisfactory technique for implementing capabilities in a distributed environment. Each capability is associated with a password. Each site keeps a public list of capabilities, where each capability is associated with a password encrypted by a one-way function (as in [24]). A client who wishes access to a server transmits the password corresponding to that capability. Access is permitted only if the encrypted version of the capability password matches the version in the published list. Clients exchange capabilities simply by transferring passwords. This simplistic protocol has two problems:

- Since messages are not presumed to be secret, an eavesdropper could steal the password, and hence the capability.
- There is no provision for revocation.

Nonetheless, this example suggests that an authentication method, if it could be made secure and revocable, could act as the foundation for constructing self-securing programs in a capability based system. This paper presents a new class of efficient authentication methods which do not suffer from drawbacks existing authentication schemes have, as well as an examination of several revocation schemes.

2.1. Zero knowledge authentication

Authentication is at the heart of the security system for any loosely-coupled distributed operating system. How can client A and server B prove their identities to one another? The problem is difficult because A and B may reside on different sites, thus they must communicate by exchanging messages across a potentially vulnerable communications network. Since messages transmitted over the network may be intercepted by a third party, C, A and B must be able to prove their identities without revealing information which would allow C to successfully feign an identity as A or as B.

How well do existing authentication methods accomplish this goal? In practice, not very well. For example, Rivest, Shamir, and Adleman proposed an authentication method based on the RSA public-key signature methods [32]. In their protocol, values are encrypted according to a public-key encryption e function $E(m)=m^e \bmod n$, where m is a message, e is an encryption key, and n is the product of two large primes p and q. Decryption is accomplished through the function $D(c)=c^d \bmod n$, where C is the ciphertext, d is chosen so that $ed=1 \bmod (p-1)(q-1)$. It is true there is no published method for quickly decrypting messages given only e and n, and not the factorization of n. Nevertheless, this method leaks information. For example, Lipton points out that the well known Legendre function L satisfies the relation $L(m,n)=L(E(m),n)$ [19]. Indeed, the problem is much worse. Alexi, Chor, Goldreich, and Schnorr recently proved that if an adversary can find the low order bit of m $50\%+\varepsilon$ of the time given $E(m)$, she can invert arbitrary RSA encryptions [1, 5]. A corollary to this result is that the usual query-response methods for encryption, such as the family of protocols described in [23], an adversary can 2 emulate another client or server after engaging in $O((\log n)^2)$ authentications.

Needham and Schroeder have suggested an authentication method which uses private-key cryptographic methods [26]. Needham and Schroeder's work presupposes a secure key distribution method and a private-key cryptosystem. Recent work by Luby and Rackoff suggests that authentication methods depending on DES [25] are vulnerable to a "low-bit" attack similar to the one mentioned above for the RSA cryptosystem [20]. For example, the second author has found a method to subvert the authentication scheme used by the Andrew File System VICE [34, 33], which uses a strategy similar to Needham and Schroeder's.

To give users confidence in a system, we would like to be able to prove that an authentication method does not leak information. Several researchers have independently proposed protocols, termed *zero-knowledge protocols*, which satisfy this constraint, given a complexity assumption that P ≠ NP [9, 3, 10, 7]. To convey a flavor of arguments used, we summarize a zero-knowledge protocol below by which A can prove to B that some graph G with n vertices known to both A and B contains a k-clique (that is, a set Q of k vertices such that between every two vertices in Q there exists an edge). (This version of the proof is due to M. Blum.) Let G be a graph with n vertices known to both A and B. Suppose that A knows a k-clique in G. Since the problem of finding a k-clique in an arbitrary graph is NP-complete, B can not in general find the k-clique. This protocol will allow A to prove to B that G has a k-clique without revealing any information about the vertices in Q.

1. A secretly labels each vertex of G with random unique integer from 1 to n.

2. A prepares $n(n-1)/2$ envelopes labeled uniquely with a pair of integers $\langle i,j \rangle$, $i<j$. A puts "Yes" in the envelope labeled $\langle i,j \rangle$ if an edge exists between the vertices labeled i and j, and "No" otherwise.

3. A seals the envelopes and presents them to B. B flips a coin and reports its value to A. If the value is heads, A must open all the envelopes and show the numbering of the vertices of G. B then verifies that the descriptions are correct. On the other hand, if the value is tails, A must then open only the envelopes which are labeled $\langle i,j \rangle$ where i and j belong to Q. B then verifies that all envelopes contain "Yes".

4. The above protocol is repeated t times (with an independent random numbering assigned each time in step 1). If A successfully responds B's queries, the probability that A does know a proof is 2^{-t}.

Clearly, if A knows a k-clique and correctly follows the above protocol, A will succeed. On the other hand, suppose C is trying to masquerade as A. Since C does not know a k-clique, C has two choices: it can correctly perform step 2 (in which case it is caught whenever B gets tails) or it can put false values in the envelopes (in which case it is caught whenever B gets heads). Hence in each of the t iterations of the protocol the probability that C's ignorance revealed is 1/2. After t iterations, C will be caught with probability $1-2^{-t}$. Finally, notice that B acquires no information about the location of the clique. If B could find information about the clique from the above protocol, it could generate the same

information by flipping a coin and generating a random numbering of the graph when it gets heads and a random numbering of a complete graph on k vertices when it gets tails.

Notice that since any problem lying in NP can be reduced to the NP-complete problem k-clique [14], A could use this protocol to prove to B that it had a proof or disproof of, for example, Fermat's Last Theorem. At the end of this protocol, B would be convinced that A did, in fact, have a proof or disproof without having any idea which way the problem was resolved, much less any idea of the technique used to solve the problem. In principle, this protocol could be used to generate authentication proofs: A would publish the graph G in a public white pages directory. To prove its identity, A would give a zero-knowledge proof of the existence of a k-clique in the graph.

In practice, however, this protocol would not work well. First, A would have to find a graph G in which it was computationally intractable to find a k-clique. While it is true that our complexity hypothesis guarantees that such graphs must exist, most random graphs with k-cliques, those cliques can be found through efficient heuristics. Second, A and B would have to develop a good cryptographic scheme for implementing "envelope exchange". For even a modest security level, the size of data involved here is on the order of 10^{200} bytes. Using the highest bandwidth transmission techniques available today, execution of this protocol would exceed the time remaining before the heat death of the universe.

2.2. An efficient protocol

In research described in [38], we have developed a family of zero-knowledge protocols which are efficient for real use in applications. That paper also gives timing figures for those algorithms. Below we give a simplified (and slightly less efficient) version of the protocol.

The protocol we use can depend on one of two complexity assumptions: that factoring large integers can not be done in polynomial time, or that it is hard to invert messages encrypted by random keys under DES. Other similar complexity assumptions may be used instead. The protocol described below depends on the complexity of factoring integers. We recall the following lemma by Rabin [28]:

> If there exists a polynomial time algorithm for finding square-roots modulo $n=pq$, where p and q are large primes, then we can factor pq in polynomial time.

Rabin observed that we can take a random integer r between 1 and pq-1; check that GCD(r,n) $\neq$ 1 (if this value is p or q, then we have factored n). Calculate $x=r^2$ mod n and find a square-root s, so that $s^2=x=r^2$ mod n. A simple number-theoretic argument demonstrates that x has four square-roots modulo n, including r and -r. Since r is chosen at random, there is a 50% chance that $s \neq \pm r$ mod n. If $s = \pm r$ mod n, then we can pick a new r and repeat the algorithm. If $s \neq \pm r$ mod n, then it is the case that GCD(r+s,n)=p or q. Hence finding square-roots is equivalent to factoring.

In this protocol, we assume that the system manager publishes a product of two large primes $n=pq$, keeping the factorization secret. This n can be used for all authentication protocols, and no one need ever know its factorization. To 2 initialize its puzzle, A picks a random r and publishes $x=r^2$ in the white pages. I will prove it knows a square-root of x without revealing any information about the value.

Here is the protocol:

1. A computes t temporary random values, v_1, v_2,...,v_t, where each v_i satisfies $1 \leq v_i \leq pq$-1. A sends to B the vector $\langle v_1^2 \bmod n, v_2^2 \bmod n, ..., v_t^2 \bmod n \rangle$.

2. *B* flips t independent coins and sends back a vector of t random bits $\langle b_1,...,b_t \rangle$ to A.

3. For $1 \leq i \leq t$ A computes:

$$z_i = \begin{cases} v_i & \text{if } b_i=0 \\ rv_i \text{ mo } n & \text{otherwise} \end{cases}$$

   A transmits the vector $\langle z_1 ,...,z_t \rangle$.

4. *B* verifies that for $1 \leq i \leq t$, that:

$$z_i^2 = \begin{cases} v_i^2 \bmod n & \text{if } b_i=0 \\ xv_i^2 \text{ mo n)} & \text{otherwise} \end{cases}$$

   If the equalities hold, A has authenticated its identity to B with probability $1-2^t$.

Once again note that if A knows a value r such that $r^2=x$ mod n, then it can easily follow the above protocol. Suppose C is trying to masquerade as A. Since C

doesn't know such a value r, it can not know both v_i and $rv_i \bmod n$, since $r=rv_i\,(v_i)^{-1} \bmod n$. Finally, all that B sees is a series of random values of the form $\langle z_i, z_i^2 \bmod n\rangle$. If B could find any information about r from the above protocol, it could do so by generating a set of t random values and squaring them modulo n, and thus factoring n.

The above protocol uses only an expected $3t$ multiplications to generate security of $1\text{-}2^{-t}$. Our improved protocol uses only expected $1.5t$ multiplications to achieve the same level of security.

2.3. Self-securing programs

As mentioned above, our algorithm presumes communication networks which are potentially vulnerable. A and B need to use methods to protect the privacy and integrity of their data. If we extend an authentication algorithm to support key exchange, A and B can transmit their messages though highly secure private-key encryption methods. We have adapted our algorithms to also perform this operation -- A can send a temporary key e_A and B can send a temporary key e_B. Both parties can then use a trusted private-key encryption method, such as DES with the key $e=e_A \oplus e_B$, where $\oplus$ is bit-wise exclusive-or. Hence, if A and B have protected address spaces, we can make all messages transmitted in the system public, since no observer can find the encryption key used. Also, sending messages encrypted by the key removes the need to re-authenticate until either party decides to establish a new temporary key. This approach yields a *self-securing program* which requires only a minimal amount of security in our base operating system.

Indeed, our algorithm obviates the need to ever use public-key cryptography. If A wishes to transmit a message to B without having a shared private key, A can simply authenticate itself to B exchanging a temporary encryption key. All further communications are protected by encryption. The signature functions of public-key cryptography can be performed by the fingerprinting algorithm described in the next Section.

3. Fingerprinting

Karp and Rabin introduced an algorithm which computes a *cryptographic checksum* [30, 15]. Their algorithm takes a bit string s of arbitrary length and secret key k of d bits (where d-1 is prime) and returns a *fingerprint* sequence of

d-1 bits $\Phi_k(s)$. Each key k defines a *fingerprint* function, and if the keys are k chosen with uniform distribution, the family of fingerprint functions Φ_k can be viewed as a provably good random hash function in the style of [4, 31]. Without the secret key, computing a fingerprint given a string of bits is intractable. On the other hand, if the secret key is known, it is easy to compute the fingerprint.

Given the fingerprint algorithm, the problem of protecting the integrity of data from alteration becomes much simpler. For example, to protect a file F, we could store $\langle F,\Phi_k(F)\rangle$. If an adversary attempts to alter the file by k replacing it with F', he will need to calculate $\Phi_k(F')$. But since the k adversary does not know k, he can not compute the fingerprint of F'. Even if the adversary attempts to find a F' with the same fingerprint $\Phi_k(F)=\Phi_k(F')$, he will be thwarted, since the problem of finding an k k input which generates a given fingerprint is intractable without the key value k.

The fingerprinting algorithm views a sequence of bits s as a polynomial $f_s(x)$ over the integers modulo 2. For example, the bit sequence s= "100101001" is taken to be the polynomial $f(x)=x^8+x^5+x^3+1$. The secret key for this algorithm corresponds to a random irreducible polynomial $g(x)$ of degree d-1 over the integers modulo 2. It is extremely easy to generate these polynomials (several approaches are outlined in [29, 30]) and we have implemented two different efficient routines for doing so. Compute $r(x) = f_s(x) \bmod g(x)$. $r(x)$ is a polynomial over the integers modulo 2 of degree at most d-1. Both the polynomial $r(x)$ and the key k can be represented as a string of d bits. The bits produced by the algorithm define the Φ function.

4. Revocation

At any time, a capability can be revoked by its owner, rendering it invalid for all clients to which it has been transferred. Ideally, revocation should take effect instantaneously, ensuring that any subsequent attempt to use that capability will be refused. In a decentralized distributed system, however, it is difficult to make such a guarantee in the presence of variable message delays, site crashes and communication failures. Instead, we guarantee that revocation appears to be instantaneous to each individual client or server in the system, in the sense that no one should be able to observe that a capability has been used after it has been revoked.

To illustrate the meaning of this guarantee, consider the following simple example. A professor issues a homework assignment to each student in her class, together with a capability for turning in the completed assignment. The professor will accept late homeworks up until the time she gets around to publishing the sample solution. To keep students from resubmittting the published solution, the professor simply revokes the capability before publishing the sample. If revocation appears to be instantaneous, a student who has read the published solution no longer has the right to turn in that assignment.

As discussed in Section 5, our revocation scheme is an adaptation of an algorithm originally developed for a very different purpose: managing orphans in distributed transaction systems [22].

4.1. The basic mechanism

Each server has a clock. Our revocation method relies on the assumption that clocks can be kept approximately synchronized: the maximum skew between the clocks at two servers protected by the same capability is always less than some constant Δ. It is important that no adversary can trick two servers' clocks into diverging by more than Δ, thus if clock synchronization is maintained by periodic exchange of messages (e.g., [21]), then all such messages must be authenticated.

Each server protected by a capability maintains a *quiesce time* and a later *refusal time*. Subject to constraints described below, different servers may have different quiesce and refusal times for the same capability. If an invocation arrives at a server before the capability's local quiesce time has elapsed, the capability is considered valid, and the operation is allowed to proceed. If it arrives after the refusal time has elapsed, however, it is considered invalid, and the operation is refused. If the invocation arrives between the quiesce and refusal time, then the capability's status is indeterminate. The server postpones a decision to accept or reject the operation until either (1) the capability's lifetime is extended, in which case the operation is permitted, or (2) the refusal time elapses, in which case the operation is refused.

Let *First(Refusal(c))* denote the earliest refusal time for c at any server protected by *c*, and let *Last(Quiesce(c))* denote its latest quiesce time. The capability's quiesce and refusal times are subject to the following *revocation invariant*:

$$\text{Last(Quiesce(c))} - \text{First(Refusal(c))} > \Delta$$

The difference between a capability's latest quiesce time and its earliest refusal time must exceed Δ, the maximum clock skew. This invariant ensures that there is always some moment of real time, call it t_0, at which the 0 capability's quiesce time has elapsed at every server but its refusal time has not yet arrived. This t_0 is the moment at which revocation appears to take 0 effect.

We assume that the owner knows a lower bound for First(Refusal(c)) and an upper bound for Last(Quiesce(c)). In the next two sections, we describe a *revocation protocol*, a fault-tolerant protocol for revoking capabilities, and a *refresh protocol*, a fault-tolerant protocol for extending capabilities' lifetimes.

4.2. The revocation protocol

The simplest way for an owner to revoke a capability is just to wait for the earliest refusal time to elapse, at which time the capability will no longer be considered valid anywhere in the system. Nevertheless, if the refusal time is far in the future, it will be desirable to hasten revocation by active means. This section describes a *revocation protocol* that can be used to adjust the refusal time without violating the revocation invariant.

The revocation protocol is a two-phase protocol similar to the two-phase commit protocol [11]. For brevity, we use "all servers" as shorthand for "all servers protected by the capability in question". In the first phase, the owner sends a message to all servers, directing each one to reset its local quiesce time to the present. Each server returns a confirmation. The first phase is successful if the owner receives confirmations from all servers. If the first phase is successful, the owner sends a second message, directing all servers to reset their refusal times to the present. The owner can then resume execution. Although no confirmation messages are needed for phase two, they might reduce the likelihood of unnecessary delay at servers that missed the phase two message. If the first phase fails, revocation is blocked until the earliest refusal time elapses.

This protocol is *fail-safe*. By adjusting the quiesce and refusal times in two phases, we ensure that the revocation invariant is preserved at all times, even in the presence of site failures, lost messages, and network partitions. Such failures would adversely affect performance, but they cannot affect the correctness of revocation. In the presence of failures such as network partitions, it is difficult to imagine how one could make a stronger guarantee.

4.3. The refresh protocol

A disadvantage of the scheme as described so far is that capabilities have fixed lifetimes. If a server's quiesce time arrives before a legitimate client has finished, the client must request a new capability from the owner. To avoid this difficulty, an owner may periodically undertake a *refresh* protocol to advance each server's quiesce and refusal times. The interval between a site's current time and the quiesce time for any server is the *quiesce interval*, and the interval between the quiesce and refusal times is the *refusal interval*. The interval between refresh protocols is the *refresh interval*. These terms are illustrated in Figure 1. Unnecessary revocation will be unlikely if clocks are closely synchronized and if the refresh interval is significantly less than half the quiesce interval.

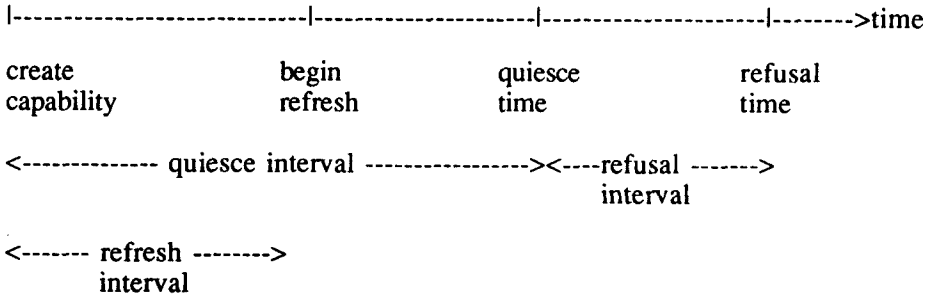

Figure 1. Quiesce, Refusal, and Refresh Intervals

The refresh protocol is a two-phase protocol similar to the revocation protocol In the first phase, the owner attempts to advance the capability's refusal time at all servers. If the first phase is successful, i.e., if all server confirm they have done so, the owner attempts to advance the capability's quiesce time at all servers. Here, too, the two phases preserve the revocation invariant, and hence the protocol is fail-safe in the presence of arbitrary site failures and network partitions. As before, confirmation messages are not needed for the second phase, but they might enhance performance by reducing the likelihood of spurious delays. In practice, the refresh and quiesce intervals may have to be tuned to incorporate such factors as lost refresh messages and the retransmission rate.

4.4. Crash recovery

A simple way to preserve the revocation invariant across site crashes is to keep quiesce and refusal times in non-volatile storage, perhaps in a small "stable cache". If this technique is impractical, an alternative technique is to set a system-wide maximum value for the *quiesce interval*, the duration between a server's current clock value and the quiesce time for any capability (see Figure 1.). When a server recovers, it reinitializes its clock, and refuses all operation invocations until the maximum quiesce interval (plus Δ) has elapsed at every server, ensuring that all capabilities valid at the time of the crash have quiesced.

4.5. Lazy revocation

An interesting "lazy" variant of this revocation method arises if, instead of using clocks to drive revocation, we use revocation to drive clocks. Real-time clocks are replaced by logical clocks [16] satisfying the following properties:

1. Each server's clock generates successively increasing timestamps.
2. When a message is sent from one server to another, the time at which it is received (by the receiver's clock) is later than the time at which it was sent (by the sender's clock).

The second property is ensured if, whenever one server sends a message to another, the sender includes its current logical time, and the recipient advances its own logical clock beyond the observed value.

It must be emphasized, however, that properties 1 and 2 can be guaranteed only if all sites communicate via trusted network interface modules that can be relied upon to affix valid logical timestamps to outgoing messages. The lazy scheme is not secure in systems where an adversary can transmit messages with forged logical timestamps.

As before, each capability has a quiesce and refusal time at each server, satisfying a slightly different revocation invariant.

$$\text{Last}(\text{Quiesce}(c)) < \text{First}(\text{Refusal}(c))$$

This invariant ensures that there exists some logical *time* before which the capability has quiesced at all servers, and after which it will be refused. As before, the capability is considered valid before its quiesce time has elapsed; it is

considered invalid after its refusal time has elapsed, and otherwise its status is indeterminate. The difference here is that all times are logical times, not physical times.

The major advantage of lazy revocation is that it is "demand-driven" rather than "time-driven". To revoke a capability, an owner simply advances its local logical clock to its refusal time, and resumes execution. It is never necessary to wait for a capability's refusal time to elapse, nor is it necessary to undertake a revocation protocol, simply because a server's logical clock can be advanced instantaneously. Similarly, it is not necessary to undertake refresh protocols at fixed intervals. Instead, refresh protocols are triggered by revocations. When an owner revokes a capability, it advances its local clock, which eventually advances other clocks as messages propagate through the system. If a server's quiesce time suddenly elapses, it may send a message to the owner asking it to initiate a refresh protocol.

Whether the eager scheme's combination of periodic refresh protocols with delays is more cost-effective than the lazy scheme's demand-driven refresh protocols without delays depends on the expected frequency of revocations and the relative costs of delay and of message traffic. Of course, one disadvantage of the lazy scheme is that it provides no real-time guarantees about revocation. A related disadvantage is that this scheme is vulnerable to information flowing through "covert channels". In the homework example, if the professor's machine is partitioned from the machine where the homework is to be handed in, a student could read the posted solution in one partition, walk over to a machine in the other partition, and submit the stolen solution.

5. Discussion

How should quiesce and refusal times be chosen in practice? Naturally, this choice depends on the expected rates of failures and the expected frequency of revocation. The message traffic associated with refresh protocols can be kept to a minimum by choosing quiesce and refusal times relatively far in the future. Although it would then take a long time for the earliest refusal time to elapse, most revocation would be accomplished by the two-phase revocation protocol. An owner would wait for the earliest refusal time to elapse only as a fall-back measure when network failures cause the revocation protocol to fail, presumably a rare occurrence.

Our revocation scheme is easily extended to allow the set of servers protected by a capability to change dynamically. To create a new server protected by an existing capability, it is enough to leave a "forwarding address" at an existing server. The revocation and refresh protocols are extended by having the older server forward the phase one message to the newer server, and making its own phase one confirmation conditional on the newer server's confirmation.

The revocation method proposed here is an adaptation of an algorithm originally developed to detect and eliminate *orphans* [22] in distributed transaction systems. An orphan is an activity executing on behalf of an aborted transaction. Orphans are undesirable because they waste system resources, and because they may observe inconsistent data. At first, it may seem curious that revocation and orphan elimination are so closely related, but the connection becomes clearer if we think of an orphan as a transaction whose "right to exist" is revoked when it is aborted.

It is natural to ask whether other orphan elimination algorithms can also be adapted for revocation. Orphan elimination schemes for non-transactional systems [27] do not seem appropriate, since they do not provide any consistency guarantees. The algorithm used by Argus [12] requires that each process include all the orphan information it "knows" in each message, relying on various optimizations to keep message sizes under control. Unfortunately, the Argus scheme, like the "lazy" scheme described above, requires the cooperation of all processes in the system, and hence it is vulnerable to an adversary who neglects to pass on the required information. Our scheme circumvents this difficulty by relying on approximately synchronized real-time clocks; in effect, using the passage of time as a secure message channel.

6. Conclusions

We have seen how we can build a capability system in a distributed environment without placing extensive support code in the kernel. While this paper primarily addresses systems that implement the client-server model, we believe that our algorithms may also benefit other approaches to distributed systems. We have presented a new zero-knowledge authentication protocol and a new revocation protocol, each of which can be provided through library routines in individual programs running on a loosely-coupled system. Although we have not attempted to solve the denial of service problem, recent work has shown that distributed client/server systems which use replication to achieve fault tolerance can be

efficiently integrated with secure systems. We intend to pursue research in this direction, to see if self-securing programs can protect themselves against denial of service attacks through the use of replication.

Our self-securing capability system depends on some assumptions. We assume that address spaces are protected, and that certain problems, such as factoring large integers or inverting text enciphered by DES, are intractible. Our self-securing approach has several important implications. It implies that we can build systems with only minimal security support, knowing that we can retrofit the systems later. It means that we can verify and validate software which can meet arbitrarily high standards of security without using the traditional nested security approach, in which a distributed secure application relies on a distributed secure library of programs which in turn rely on a distributed secure file system, which in turn relies on a distributed secure kernel, etc. [2]. Instead we can move security out of the kernel and into user space. This approach may help tame the computer security problem for distributed systems and reach high levels of security with relatively small amounts of development effort.

References

[1] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr, RSA and Rabin Functions "Certain Parts are as Hard as the Whole", in *Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science*, November 1984. To appear in SIAM J. on Computing.

[2] J. P. Anderson, "Computer Security Technology Planning Study", Technical Report ESD-TR-73-51, USAF Electronic Systems Division, October 1972.

[3] Laszlo Babai, "Trading Group Theory for Randomness", in *Proceedings of the 17th ACM Symp. on Theory of Computing*, pp. 421-429, May 1985.

[4] J. Carter and M. Wegman, "Universal Classes of Hash Functions", in *Proceedings of the 17th IEEE Foundations of Computer Science*, pp. 106-112, May 1976.

[5] Ben-Zion Chor, *ACM Distinguished Dissertations: Two Issues in Public Key Cryptography: RSA Bit Security and a New Knapsack Type System*, MIT Press, 1986.

[6] E. Cohen and D. Jefferson, "Protection in the Hydra Operating System", in *Proceedings 5th Symp. on Operating Systems Principles*, pp. 141-160, November 1975.

[7] Uriel Feige, Amos Fiat, and Adi Shamir, "Zero Knowledge Proofs of Identity", in *Proceedings of the 19th ACM Symp. on Theory of Computing*, pp. 210-217, May 1987.

[8] E. F. Gehringer and R. J. Chansler, Jr., "Star OS User and System Structure Manual", Technical Report, Carnegie Mellon University, 1981.

[9] Shafi Goldwasser, Silvio Micali, and Charles Rackoff, "The Knowledge Complexity of Interactive Proof Systems", in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, May 1985.

[10] S. Goldwasser and M. Sipser, "Arther Merlin Games versus Zero Interactive Proof Systems", in *Proceedings of the 17th ACM Symp. on Theory of Computing*, pp. 59-68, May 1985.

[11] J.N. Gray, *Notes on Database Operating Systems*, Lecture Notes in Computer Science 60, Springer-Verlag, Berlin, 1978, pp. 393-481.

[12] M.P. Herlihy, N.A. Lynch, M. Merritt, and W.E. Weihl, "On the correctness of orphan elimination algorithms", in *17th Symposium on Fault-Tolerant Computer Systems (FTCS)*, July 1987. Abbreviated version of MIT/LCS/TM-329.

[13] Maurice P. Herlihy and J. D. Tygar, "How to Make Replicated Data Secure", in *Advances in Cryptology, CRYPTO-87*, Springer-Verlag, August 1987.

[14] R. M. Karp, "Reducibility among Combinatorial Problems", *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-103.

[15] Richard M. Karp, 1985 Turing Award Lecture, "Combinatorics, Complexity, and Randomness", *Communications of the ACM* 29(2): 98-109, February 1986.

[16] L. Lamport, "Time, clocks and the ordering of events in a distributed system", *Communications of the ACM* 21(7): 558-565, July 1978.

[17] B. W. Lampson, "Protection", *ACM Operating Systems Review* 8(1): 18-24, January 1974.

[18] Henry M. Levy, *Capability-Based Computer Systems*, Digital Press, 1984.

[19] R. Lipton, Personal communication.

[20] Michael Luby and Charles Rackoff, "Pseudo-random Permutation Generators and Cryptographic Composition", in *Proceedings of the 18th ACM Symp. on Theory of Computing*, pp. 356-363. May 1986.

[21] K. Marzullo and S. Owicki, "Maintaining time in a distributed system", in *Proceedings of the second ACM Symposium on Principles of Distributed Computing*, pp. 295-305, August 1983.

[22] M.S. McKendry and M.P. Herlihy, "Time-driven orphan elimination," in *Fifth Symposium on Reliability in Distributed Software and Database Systems*, January 1986. Also CMU-CS-85-138.

[23] C. Meyer and S. Matyas, *Cryptography*, Wiley, 1982.

[24] R. Morris and K. Thompson, "Unix Password Security", *Communications of the ACM* 22(12): 594-596, December 1979.

[25] NBS Standard, "Data Encryption Standard", Technical Report FIPS Publication 46, National Bureau of Standards, January 1977.

[26] Roger M. Needham, Michael D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers", *Communications of the ACM* 21(12): 993-999, December 1978. Also Xerox Research Report, CSL-78-4, Xerox Research Center, Palo Alto, CA.

[27] B. Nelson, "Remote Procedure Call", Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1981.

[28] Michael Rabin, "Digitalized Signatures and Public-Key Functions as Intractable as Factorization", Laboratory for Computer Science Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, January 1979.

[29] Michael O. Rabin, "Probabilistic Algorithms in Finite Fields", *SIAM Journal on Computing* 9:273-280, 1980.

[30] Michael Rabin, "Fingerprinting by Random Polynomials", Center for Research in Computing Technology, Aiken Laboratory TR-81-15, Harvard, May 1981.

[31] J. Reif and J. D. Tygar, "Efficient Parallel Pseudo-Random Number Generation", in *Advances in Cryptology: CRYPTO-85*, pp. 433-446, Springer-Verlag, August 1985. To appear in SIAM J. on Computing.

[32] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM* 21(2): 120-126, February 1978.

[33] M. Satyanarayanan, "Integrating Security in a Large Distributed Environment", Technical Report CMU-CS-87-179, Carnegie-Mellon University, November 1987.

[34] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, Michael J. West, "The ITC Distributed File System: Principles and Design", in *Proceedings of the Tenth Symposium on Operating System Principles*, pp. 35-50, ACM, December 1985. Also available as Carnegie-Mellon Report CMU-ITC-039, April 1985.

[35] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and Verification of the UCLA Unix Security Kernel", *Communications of the ACM* 23(2): 118-131, February 1980.

[36] M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System*, North-Holland, 1987.

[37] W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill, 1981.

[38] Bennet S. Yee, J. D. Tygar, and Alfred Z. Spector, "A Self-Securing Protection System for Distributed Programs", Technical Report CMU-CS-87-184, Carnegie-Mellon University, December 1987.