Resilient Self-Organizing Overlay Networks for Security Update Delivery

Jun Li, *Member, IEEE*, Peter L. Reiher, *Member, IEEE*, and Gerald J. Popek

***Abstract*—Rapid and widespread dissemination of security updates throughout the Internet will be invaluable for many purposes, including sending early-warning signals, updating certificate revocation lists, distributing new virus signatures, etc. Notifying a large number of machines securely, quickly, and reliably is challenging. Such a system must outpace the propagation of threats, handle complexities in a large-scale environment, deal with interruption attacks on dissemination, and also secure itself.**

***Revere* addresses these problems by building a large-scale, self-organizing, and resilient overlay network on top of the Internet. We discuss how to secure the dissemination procedure and the overlay network, considering possible attacks and countermeasures. We present experimental measurements of a prototype implementation of Revere gathered using a large-scale-oriented approach. These measurements suggest that Revere can deliver security updates at the required scale, speed and resiliency for a reasonable cost.**

***Index Terms*—Network security, overlay network, overloading-based measurement, resiliency, security update.**

I. Introduction

THERE is often an urgent need for sending early warning signals, distributing firewall or intrusion detection system updates, invoking extensive certificate revocation, disseminating new virus signatures, and delivering many other security updates. Is it feasible to deliver security updates to most of the connected nodes of an Internet-scale computer network very rapidly, resiliently, and securely? Can it be done without huge, powerful server systems? How rapidly can it be done? Within seconds, for example?

A. Challenges

Any system that attempts to deliver rapid security updates at high scale must overcome several difficult challenges.

1) Speed: Security updates must be delivered faster than attacks. Recent studies show that attacks can spread in minutes or even tens of seconds [1]. Early this year the slammer worm infected more than 90% of vulnerable hosts worldwide in less than 10 min [2]. If a node cannot receive the most recent security updates, it becomes highly susceptible to potential threats.

2) Scalability: With potentially millions of participants, it is daunting for a single machine, or even dozens of machines, to store up-to-date global knowledge concerning all participants. Any centralized management is difficult, if not impossible. Further, high scale ensures that significant numbers of nodes will be disconnected at the moment a security update is being disseminated, so any solution must handle node disconnection as a norm, rather than an exception.

3) High Dissemination Assurance: Nodes assisting in dissemination may be compromised, resulting in dropped, misdirected or damaged security updates. Encryption, authentication, and digital signatures do not help ensure message delivery at all. Authenticated acknowledgment help, but do not scale well, and typically retransmitted messages are still subject to interruption.

4) Security: Last, but not least, the system itself must be secure. A system handling millions of machines is a tempting target. If the system is corrupted, not only will the machines in the system be broken, but even larger numbers of machines. Furthermore, a sound security solution must support large-scale heterogeneous nodes, where each could enforce a very different set of security schemes.

B. Possible Approaches

One approach is to require a user to pull information from a dissemination center either manually or according to a specific schedule. However, this pulling-based approach results in a dilemma: not pulling frequently will leave a user's machine not instantaneously updated, whereas attacks may come at any moment [1], [2]; pulling frequently will incur high bandwidth cost, both at each participant and throughout the network, and it can create a flash crowd at the center, probably slowing down the center or even making it inaccessible.

Viewed in the most general context, security update delivery fits within the broad scope of information distribution. The simplest approach is to unicast, but it is not scalable to unicast security updates to millions of nodes from a dissemination center, one by one. Another approach is to broadcast, but broadcast is primarily meant for a subnet or a small collection of subnets. Still another approach is to use Internet protocol (IP) multicast, but IP multicast still faces many problems for deployment at a large scale and cannot distribute to all recipients unless they are all connected simultaneously. Reliable multicast (probably at the application layer for easy deployment) is better, but it mainly handles packet loss caused by transmission errors, not loss caused by attacks such as interruption threats; on a reliable multicast tree, all the descendents of a compromised node are cut off.

At a higher layer, protocols such as smtp, nntp, ftp and http all provide certain distribution capabilities, but it is difficult to tailor these capabilities to meet the challenges of providing a successful security update dissemination service. For example, none of these provide a resilient network to

Manuscript received November 15, 2002; revised June 1, 2003.

J. Li is with the Department of Computer and Information Science, University of Oregon, Eugene, OR 97403 USA (e-mail: lijun@cs.uoregon.edu).

P. Reiher and G. J. Popek are with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: reiher@cs.ucla.edu; jpopek@corp.untd.com).

Digital Object Identifier 10.1109/JSAC.2003.818808

address man-in-the-middle delivery threats, none consider both connected nodes and disconnected nodes for best large-scale delivery coverage, and none fully address security. Event notification services, which usually adopt a centralized approach, focus on different issues and often view the mapping between event subscribers and event publishers as a key issue [3], [4]. Much research has also been done on content delivery networks (CDN), using distributed caching or overlay techniques. Whereas security updates are usually of small size and low frequency, CDN usually handles large blocks of data.

When considering special-purpose applications, virus signature distribution is probably most similar to security update delivery service. The pulling-based approach has been widely used in this context; recognizing its drawbacks mentioned earlier, some groups set up central servers to automatically broadcast new virus signatures to every individual user, but difficulty in managing user records at the central servers grew quickly as more users participated in a dynamic way. Peer-to-peer technology has been used to address some of these problems, where information can be forwarded along a chain of recipients [5]; however, the design technology to handle disconnected nodes, strengthen security (including combating interruption threats), and maintain the chains has not been reported.

C. Revere Overview

To address these challenges, Revere builds a large-scale, self-organizing and resilient overlay network on top of the Internet at the application level. This overlay approach provides flexibility, while requiring no changes to existing network infrastructure (Revere is currently implemented as a Java application on participating nodes). Individual nodes can join and leave a Revere overlay network; once joined, nodes on an overlay will receive security updates and every nonleaf node will also forward updates-nodes can also query and pull updates if needed.

While various overlay networks have been proposed in the past [6]–[11], the special requirements and challenges of disseminating security updates requires that Revere builds its own overlay. Although Revere allows a node to join or leave a Revere overlay network at its own discretion, as do many other overlay networks, Revere's overlay network is built and maintained differently.

To combat attempts to interrupt dissemination, Revere uses a redundant overlay network for a more resilient delivery. Since Revere is designed to handle a low volume of relatively small but highly important messages, the redundancy makes great sense. A Revere overlay also handles an Internet-scale number of participants, and equally important, is self-organized. Furthermore, Revere enforces stringent security for both the dissemination process through a Revere overlay network and the overlay itself. An interesting tradeoff employed by Revere is to support a relatively heavy overlay management (construction, management, security, etc.) in order to support a simple, lightweight dissemination process.

D. Paper Outline

Section II discusses the Revere overlay network, emphasizing its self-organization, resiliency and scalability. The dissemination procedure is discussed in Section III. We identify security issues and discuss our approaches in Section IV. Section V reports on measurement results. Section VI summarizes related work. Section VII is on future work and we conclude the paper in Section VIII.

II. RBone: A Self-Organized Resilient Overlay Network

A Revere overlay network, also called an *RBone*, organizes itself. Using a straightforward user interface, Revere allows individual nodes to join or leave an RBone with no further human intervention. Through a simple but effective three-way-handshake protocol, a node can attach itself as a child of other existing Revere nodes to become part of an RBone. In particular, parent selection allows a node to select multiple parents to achieve superior resiliency, as well as efficiency. Revere can also detect problematic nodes and handle broken links, causing nodes to reattach themselves as required.

An RBone can contain millions of nodes, so scalability requires that all RBone management operations be simple and rely only on a small amount of partial knowledge at each node. In Revere, each node only keeps information about its parents, its children, and the dissemination center.

For convenience, we assume that a different RBone rooted at a specific dissemination center will be built for every different type of security update. Sharing a common RBone for different types of security updates and/or different centers is possible, but it leads to complexities not addressed in this paper.

In this section, we assume all Revere nodes are benign (not corrupted). RBone security will be discussed in Section IV.

A. Three-Way-Handshake Protocol

To join an RBone, a new node needs to locate existing Revere nodes first. Various methods can be employed, such as using configured knowledge (for example, the address of the dissemination center or a local designated Revere node), contacting a directory service, applying a multicast-based expanding-ring, or expanding-wheel search [12], etc. In fact, the discovery of Revere nodes falls into a more generalized problem of scalable resource discovery in a large-scale network and any approach to resource discovery can be adopted by an individual Revere node to locate other Revere nodes at its own discretion. A node discovery mechanism is not hardwired into Revere and is not a core component of Revere per se (note that a nonscalable resource discovery mechanism, if adopted, could affect Revere's scalability).

The new node then can negotiate with those existing nodes to attach itself to some of them as a new child. The negotiation between a potential child and a potential parent is a reciprocal selection procedure. An existing node needs to determine whether it wants to add the new node as a child. The new node, on the other hand, needs to determine whether it wants the existing node to be its parent.

The negotiation applies a three-way-handshake protocol (Fig. 1). A potential child sends an attach request to a potential parent. The potential parent decides whether to adopt the applicant as a new child, and sends back an acknowledgment. The child adoption decision is machine-specific: some machines

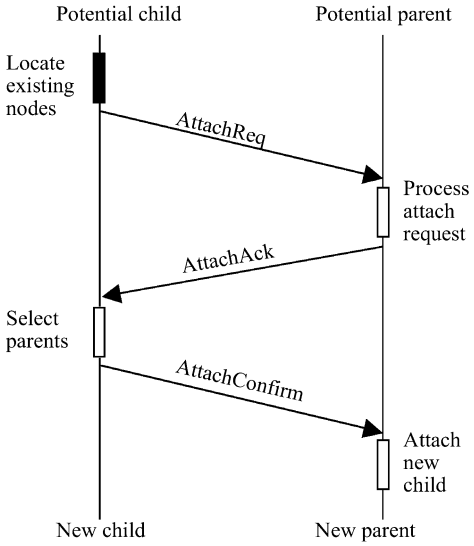

Fig. 1. Three-way-handshake protocol.

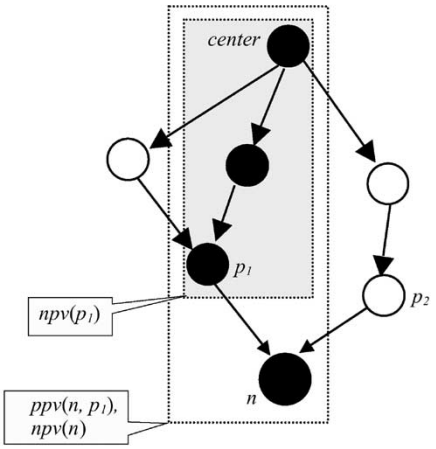

Fig. 2. Path vector at node n. Node n has two parents: p_1 and p_2. According to the definition in Section II-B-I, $\mathrm{ppv}(n, p_1)$ includes all nodes on $\mathrm{npv}(p_1)$ and n itself. Furthermore, because $\mathrm{ppv}(n, p_1)$ represents the fastest delivery path for n (in this particular example, $\mathrm{ppv}(n, p_1)$ is assumed faster than $\mathrm{ppv}(n, p_2)$), $\mathrm{npv}(n)$ is $\mathrm{ppv}(n, p_1)$.

may only check to see if they have reached the maximum number of children that can be accommodated; some machines may require more information. Revere supports pluggable machine-specific child-adoption modules. For example, because a mobile node is often disconnected, it may choose to be only a leaf node and not accept attaching requests. Or a multicast-capable node may prefer nodes that can hear multicast messages, allowing it to reach all children with a single multicast message. If the potential parent accepts the new child, it adds the applicant as a pending child and replies with a positive acknowledgment to indicate approval (otherwise, it will send back a negative acknowledgment).

Upon receipt of a positive acknowledgment, the new node decides whether to accept this potential parent (the decision procedure will be discussed in Section II-B). If yes, it will send back a confirmation. The parent, upon the receipt of the confirmation, will convert this pending child to a regular child.

A timer-based error recovery is also designed. After initiating a request, a potential child will set up a timer to await an acknowledgment from its potential parent. If the request or the acknowledgment is lost, or the potential parent refuses to respond, this timer will expire and the potential child will retransmit another request, this time possibly toward a different potential parent. Similarly, after sending a positive acknowledgment toward a pending child, a potential parent will also bind a timer with that child when waiting for possible confirmation. In case the acknowledgment or the confirmation is lost, or the potential child does not confirm, this timer will also go off and the pending child will be discarded. Note that after sending the confirmation, the child will enter the RBone maintenance procedure (to be described in Section II-C) since the child has formally added a new parent. More analysis on three-way-handshake robustness is presented in [13].

A new node typically needs multiple parents. Therefore, the new node needs to continuously search for candidate parents until it has attached itself to some predefined minimum number of parents.

During the join procedure, the transmission mechanism that the new parent uses to forward security updates can also be negotiated. The (positive) acknowledgment can contain an ordered list of transmission mechanisms preferred by the potential parent and the confirmation message can carry the transmission mechanism selected by the child.

B. Parent Selection

In this section, we discuss how to select parents to achieve the best possible efficiency and resiliency. Every Revere node, at its own discretion, can select more than one parent, typically with one of the parents providing the fastest security update delivery and the rest delivering copies along paths as disjoint as possible. A node will only miss security updates if all its paths are broken. (Because Revere works at the application level, the RBone built by Revere achieves resiliency at the same level. Achieving hardware-level disjoint paths is a topic of future research.)

1) Path Vector: We introduce *path vector* to describe a potential path for delivering security updates from a dissemination center to a node. While many properties may be defined, a path vector has two important parameters: a latency value and an ordered list of nodes to cross. Note that a path vector includes both the center and the destination node.

We further introduce two different types of path vectors.

- Parent path vector (PPV). A PPV is associated with a particular parent of a node. A node n's PPV for parent p, denoted as $\mathrm{ppv}(n, p)$, corresponds to the fastest path along which p must be the previous hop before reaching n.
- Node path vector (NPV). A node n's NPV, denoted as $\mathrm{npv}(n)$, is the fastest PPV among all node n's parents.

Clearly, $\mathrm{ppv}(n, p)$ is the concatenation of $\mathrm{npv}(p)$ and the link connecting p to n, as shown in Fig. 2.

2) Parent Resiliency Comparison: When selecting parents, a node will choose the fastest parent first, and then choose other parents primarily based on their contributions to the node's resiliency. For the former, the node can simply select a parent that maps to the fastest PPV. For the latter, however, to compare the

resiliency contribution that every parent provides is somewhat tricky. We adopt the following approach.

Step 1) Using the fastest parent as the reference, a node compares every other parent's PPV with the fastest parent's PPV (which is also the node's NPV) in terms of the number of overlapping intermediate nodes between the two paths. Here, a higher overlapping degree is assumed to lead to a weaker resiliency.

Step 2) For parents that end up with the same resiliency level in Step 1, further comparison is needed. Now, the node uses the most resilient parent obtained in Step 1 as a new reference (if there are more than one eligible to be the reference, choose the fastest), and compares the PPV of each of those parents in question with the PPV of this new reference parent, using the same procedure as in Step 1.

Step 3) Repeat Step 2 until all parents are ordered according to their resiliency contributions.

Strictly speaking, a higher overlapping degree does not always leads to a weaker resiliency. Other factors may also affect the resiliency of a path, such as the number of hops, the stability or connection quality of each node on the path, the probability that a node is compromised, etc. But a "perfect" solution would require a path vector to carry much more information in order to consider every factor that might affect the resiliency, some of which is hardly possible to obtain. On balance, we believe the heuristics we adopt in the above steps are reasonable in terms of their effect, and inexpensive to compute. Others can be substituted without altering the substance of the design or its results.

3) Parent Selection Procedure: A child c selects a parent as follows.

A potential parent x includes its NPV $\mathrm{npv}(x)$ in the positive acknowledgment that it sends to node c.

c evaluates the latency from x to itself. To do this, c can contact an existing service (such as [14]). Or, with the attaching request and the positive acknowledgment timestamped, c can estimate the round-trip time between x and itself, and use half of that value as the approximate latency from x to c (which will be further refined during RBone maintenance).

Combining $\mathrm{npv}(x)$ and the latency from x to c, node c derives $\mathrm{ppv}(c, x)$.

Given $\mathrm{ppv}(c, x)$, node c determines whether adding x as a parent improves its efficiency or resiliency, as depicted in Fig. 3.

```
Function boolean selectParent (ppv(c, x)) on node c:
   whether to select node x as a new parent.
   npv(c): current node path vector of node c.
1  if (npv(c) does not exist)  {/* c has no parent yet */
2     npv(c) ← ppv(c, x)
3     return true
4  }  else if (ppv(c, x) is faster than  npv(c))  {
      /* x improves efficiency */
5     npv(c) ← ppv(c, x)
6     return true
7  } else if  (c has not obtained the required number of parents ) {
8     return true
9  } else if  (∃ a parent m of node c, such that x, if accepted
                  as a parent, would have better resiliency
                  contribution than m according to procedure
                  in Section II.B.2)) {
      /* x improves resiliency if replacing m*/
10    return true  /* m will be removed */
11 } else {
      /* x improves neither efficiency nor resiliency */
12    return false
13 }
```

Fig. 3. Parent selection based on path vector.

C. Adaptive RBone Management

The changes to an RBone must be detected and quickly dealt with. Changes happen when a new node joins, when an existing node crashes or leaves, when a parent or a child wants to untie the connection, when the characteristics of a parent-child connection change, when a parent is detected as corrupted, when a better path is detected, or for any similar reason.

Managing an RBone is a distributed task. While an RBone can be comprised of a large number of nodes, a change may only be detected by a few. Moreover, because of the large scale of an RBone, every node only has partial knowledge of the whole RBone, mostly about its neighbors. Each node has to respond to changes autonomously, thus usually asynchronously, based on its limited knowledge.

Revere supports two different mechanisms for detecting changes: explicit notification and implicit detection. With explicit notification, a node can send a teardown message to a parent (or a child), and remove that parent (or that child) from its records. With implicit detection, a node relies on heartbeat messages to detect if its parents and children are still alive. Each parent periodically sends heartbeats to its children and each child periodically sends heartbeats to its parents. Lack of heartbeats from a parent (or a child) will eventually lead to the removal of that parent (or that child).

Parent removal causes a node to adjust its data structures, particularly its path vectors. The PPV corresponding to a removed parent will be discarded; if this PPV is the node's NPV, a new NPV must be determined.

The explicit teardown messages are not guaranteed reliable. If a teardown message is lost, the heartbeat mechanism can help. For example, if a teardown notification from a parent P to a child C is lost, although C will regard P as its parent for some period, lack of heartbeats from P will cause C to remove P.

Heartbeat messages carry other useful information as well. They carry timestamps to estimate the round-trip time between a node and a parent. Once the NPV of a parent is changed, its heartbeat toward the node will also carry the new NPV. In both cases, the node will update the parent's PPV and the node's NPV if this is the fastest parent. If the node's NPV becomes slower than one of its other PPV's, this node replaces its current NPV with its currently fastest PPV. Clearly, this NPV adjustment may further propagate.

III. Dissemination Procedure

Revere implements a dual mechanism to disseminate security updates. *Pushing* is the main method used to broadcast security updates from a dissemination center across an RBone to all nodes currently attached. *Pulling*, as a supplementary method,

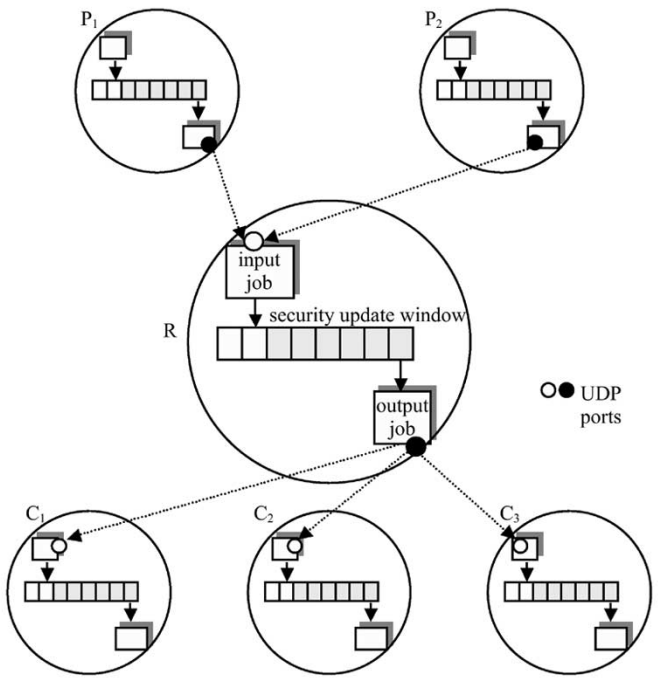

Fig. 4. UDP-based pushing operation. Here, node R has two parents: P_1 and P_2, and three children: C_1, C_2, and C_3.

allows an individual node to pull missed security updates from one or more selected repositories.

Revere delegates the reliability provision to every individual node instead of enforcing a strict global reliability. It is up to individual nodes themselves to determine how many delivery paths to obtain and maintain, to select proper transmission mechanisms, to verify updates, and to retrieve missed ones.

A. Pushing Security Updates

To begin pushing a security update, a dissemination center adds a timestamp and a sequence number, signs the message with a digital signature, and forward it toward all its children. Every node processes the update and forward it to its own children. While of critical importance, security updates are usually small and infrequent, and Revere can afford to deliver a copy of an update from every parent to every child (assuming no failures).

A simple store-and-forward mechanism at each node uses two types of jobs (input jobs and output jobs) and one main data structure (a security update window) to handle updates. An input job is responsible for receiving incoming updates from parents, processing them, and buffering them into the security update window. An output job fetches updates from the window and delivers them to local applications or its children. The transmission mechanism between a parent and a child can be negotiated during the three-way-handshake to tailor to local conditions or configurations. Fig. 4 shows those dissemination jobs and the security update window when UDP is used for transmission.

One part of processing a security update is *duplicate checking*. Because of the redundancy built into an RBone, nodes typically receive duplicate copies of security updates. Duplicate copies are identified by the sequence numbers carried in security updates and will be dropped. In addition to preventing local reuse and retransmission to children, this mechanism avoids dissemination loops. Another important part of the processing is authenticating the update (covered in Section IV).

B. Pulling Security Updates

During a pushing session, some nodes may not be connected or may be temporarily turned off. When they regain connectivity, they will want to receive the missed security updates; however, parents generally bear no responsibility for keeping all missed updates, and the retransmission from the center does not scale because every reconnected node can have a different set of missed updates. Reliable transmission mechanisms [such as transmission control protocol (TCP)] can help, but only for a short disconnection. A more general solution is to have disconnected nodes inquire about security updates that occurred during disconnections. In Revere, repository servers are used to store old security updates and respond to inquiries.

1) Repository Selection: Revere employs a dynamic repository selection, maintenance and notification mechanism. First, every node on an RBone can nominate itself as a repository to be selected or rejected. Second, an existing repository may fail (or decide to degrade itself into a normal Revere node), which will be detected. Third, whenever there is a change to the set of repositories, Revere nodes will be notified of the change.

In detail, when a node nominates itself as a repository candidate, it will add itself to the repository candidate list and piggyback that in the heartbeat messages toward every parent. In turn, when a parent receives repository candidate lists from its children, it will aggregate those lists to generate a new candidate list and piggyback the new list on its own heartbeat message toward its own parent. This repeats until the dissemination center receives a final list of all repository candidates.

If there are millions of nodes, each nominating itself to be a repository, then the piggybacked lists will be huge, especially when they get closer and closer to the dissemination center. To address this scalability issue, every Revere node can filter the piggybacked list if the list is longer than a threshold value. For example, for the longest IP address prefix that contains multiple self-nominees, only select one of them randomly; this can repeat recursively until the list length is below threshold (this procedure can help uniformly distribute repositories within the IP address space).

The center checks the repository candidate list and selects some candidates to be the repositories (we omit the selection details in this paper). The center then propagates the new results through heartbeat messages toward its children. Every child will record the list of new repositories and forward the list toward its own children (again through heartbeat messages).

Revere handles repository failures. In addition to the repository candidate list, a heartbeat from a child can also carry the identities of current repository servers. Similarly, a parent can aggregate such information from all of its children and piggyback the aggregated result on its own heartbeat message toward its own parent. Since no heartbeat will be reported from a failed repository, the center will receive a final list of repositories still alive. Changes, if any, will be notified similarly as above.

This approach is made scalable by piggybacking repository-related information in heartbeats and ensuring that every node,

including the center, only listens to its parents or children. The drawback is that low-frequency heartbeats may result in a relatively slow maintenance speed, but given that a node only needs to pick a few repositories to query, the node can tolerate relatively inaccurate knowledge of the whereabouts of all current repositories.

2) Contacting Repository Servers: Since every node keeps a local list of available repositories, it then can inquire about missed security updates from one of those repositories.

Repositories bring other benefits as well. If a node has received updates n, but not update $n-1$ yet, it can always query a repository. If a node has not received any security updates from its parents in a suspiciously long time, it can also check with repositories, offering protection against the possibility that all of a node's parents were corrupted.

An issue here is the security of a pulling operation, since a repository might very well be subverted itself. We defer this discussion to Section IV.

IV. Security

Given that some Revere nodes can be compromised, Revere must secure itself. After introducing possible threats to Revere, in this section, we address Revere's security from two aspects: 1) **security of a dissemination procedure** by adopting a public key cryptography-based approach, as well as handling key corruption at a dissemination center and 2) **security of an RBone** by enforcing trust management and discretionary authentication, where for the latter we focus on authentication scheme negotiation and pluggable security boxes.

A. Threats

In this paper, we consider the following threats.

- *Security update interception threat at Revere nodes*. This includes dropping, misdirecting, delaying, damaging, forging, or replaying security updates. For example, a replayed security update, if not recognized, can cause a node to incur extra CPU processing overhead; worse, a replayed security update may be further propagated to thousands of other nodes, sometimes circling around indefinitely. Depending on what action is taken on receipt of a security update (which is, strictly speaking, outside of Revere's scope), delivering a sufficiently stale replayed update could have arbitrarily bad consequences, such as undoing a more recent security update to a piece of software. Designing systems to handle security updates delivered by Revere will be much easier if they can be built with a high assurance that the update Revere gives them has not been replayed.
- *Corruption of a repository*. A compromised repository can provide tampered or incomplete updates.
- *Key theft at a dissemination center*. If the key that a center uses to sign security updates is stolen, an attacker can impersonate the center.
- *RBone attack*. A malicious node may provide false RBone information, replay previous control messages, or impersonate another node. One typical attack is to form a malfunctioning RBone, such as a Sybil attack, where a few malicious nodes tries to become parents of a great deal of benign nodes.

B. Dissemination Security

1) Security Update Protection: Revere protects security updates by ensuring their integrity and authenticity, strengthening their availability, and preventing replays. (Note that in light of Revere's free subscription model, secrecy protection is not needed.)

Revere adopts a public key cryptography-based approach. A dissemination center has a public key and a private key, and signs security updates with its private key. Every node uses the public key of the dissemination center to verify that those security updates have not been modified and are indeed distributed by the center. (The public key of the center is assumed to be well known.) This approach prevents those attacks that forge or damage security updates.

Revere strengthens security update availability by ensuring that an RBone is in a sound structure (whose protection is addressed in Section IV-C) and every node can choose to have multiple resilient delivery paths (as described in Section II). Attacks such as dropping, misdirecting, or seriously delaying security updates have to compromise *all* delivery paths in order to succeed.

The duplicate check at every node clearly prevents attacks that replay security updates (Section III-A).

2) Subverted Repository: Although the security update authentication mechanism in Section IV-B1 ensures that a subverted repository cannot forge false updates (since an update still carries the signature of the center), the repository could still easily fail to deliver some of the updates it had received.

Revere again employs redundancy to achieve high certitude that *all* missed security updates have been retrieved: a node can contact more than one repository. As an optimization, instead of literally pulling security update copies from each contacted repository, a node can just pull security updates from only one of them—a "master" repository, and contact other "slave" repositories to check whether the master repository provided a complete set of missed updates, typically done by comparing the range of sequence numbers of recently disseminated updates. For example, after retrieving missed updates with sequence numbers 10–15 from the first repository, if the second repository reports that it has updates up to 17, the node can then try to pull security updates 16 and 17 from the second repository (and discover whether the first repository was incomplete or the second one was cheating).

3) Key Corruption Management: The private key of a dissemination center must be carefully protected. If, despite such care, the private key of the center is compromised, disastrous attacks can be launched since attackers can now impersonate the center. Four issues must be addressed: impersonation detection (how can Revere detect that an attacker is impersonating a center and sending forged updates?), key invalidation (how can the current broken public key be revoked?), key switch (how can nodes switch to the next public key of the center?), and redelivery (should old security updates be redelivered?).

a) *Impersonation Detection*: Whereas out-of-band knowledge can be used for this difficult problem, we are also investigating a reverse traversal mechanism. If a node is still suspicious after verifying an update, the node can report this update to all its parents. Every parent will verify the reported update and further forward to its own parent (only one copy of an update will be reported). This repeats until the reported update reaches the center, which then can diagnose the authenticity of the update.

b) *Key Invalidation*: The dissemination center itself cannot distribute a new public key to replace the old one, since an attacker could then easily impersonate the center. Instead, the center sends out a key invalidation message to declare that its current public key should be invalidated. The invalidation message is signed using the broken private key and disseminated in the same way as a normal security update.

Here, the key invalidation message is really simple—it does not contain any extra information. The reason is that any extra information cannot be trusted by a node at all, since the attacker who has the private key can easily fabricate or change those fields. The key invalidation message is designed to only pass a single fact to Revere nodes—a public key must be invalidated, but nothing more than that.

If an attacker who has compromised the private key creates its own invalidation message, it would destroy any benefit the attacker received from cracking the key. The attacker may try to suppress the invalidation message, but an RBone is already a resilient network with a built-in redundancy.

A repository keeps all key invalidation messages. Upon receipt of a pulling request for missed updates from a reconnected node, the repository will determine whether a key invalidation message should also be returned, as well as missed updates.

c) *Key Switch*: After key invalidation, a new pair of center public and private keys must now be made current. In our existing implementation, every node preinstalls a series of center public keys and can now switch to the next version public key in the series. This switching mechanism is also useful if every public/private key pair has a lifetime.

d) *Redelivery*: During the period between key compromise and key invalidation, the security updates received at a node or repository, even though verified as authentic, can be either forgeries or valid updates. The forgeries should be purged and the valid updates should be replaced with new versions that are signed by the new private key. To deal with this, the center will conservatively estimate the key corruption time and resend those updates that were sent during estimated key corruption time and key invalidation, signed using the new private key.

C. RBone Security

The goal of securing an RBone is to ensure a sound RBone structure so that security updates can be delivered to a large percentage of nodes, if not all. Attackers, as described in Section IV-A, can try to compromise an RBone through false or replayed RBone control messages, probably impersonating another node. To address this problem, we believe the following requirements must be met: 1) a node can choose to only rely on those it trusts; 2) a node can authenticate whether another node is a trusted node as claimed; and 3) a node can verify messages incurred during RBone operations. (Note that even if a node can authenticate the identity of another node, it does not mean that the former trusts the latter.) In this section, we discuss trust management in Section IV-C-I, and node authentication and RBone message authentication in Section IV-C-II.

1) Trust Management: The following trust relationships need to be considered:

a) whether a node trusts another node to be its parent;
b) whether a node trusts another node to be its child;
c) whether a node trusts another node to be its ancestor;
d) whether a node trusts an entity involved in key management (such as a certificate authority).

Revere adopts a decentralized node-to-node trust management to handle these trust relationships. This allows each individual node to set up its own rules for trust judgment, avoiding the performance bottleneck and a single point of failure in centralized control.

This approach can also support richer trust functionalities than can a centralized trust management. A node's trust of another node can vary from *complete trust*, *selective trust*, and *no trust*. (In the selective trust relationship, a node is only trusted by another at certain level and/or when performing certain operations.) Second, a node can determine its trust of another node based on certain rules. With a direct-trust rule, a node only trusts another if it is specifically configured to do so. With an indirect trust rule, a node can either deduce its trust of another node based on its trust of third parties (for example, a chain-of-trust rule dictates that if A trusts B and B trusts C, then A will trust C), or contact a trust authority regarding whether or not a particular node should be trusted. The trust authority is analogous to certificate authority and can also be organized in a hierarchy, but the former certifies the trustworthiness of nodes and helps trust management, and the latter certifies the authenticity of nodes and helps node identity authentication.

A Revere node must only select a trusted node to be its parent or its child. Furthermore, nodes on the parent's node path vector also must be trusted. If this node applies direct trust or decides its trust by querying a trust authority, it checks every node on the path vector. If a node applies the chain-of-trust rule, it checks the parent node only (since the parent trusts its own parent, and this repeats recursively).

Good trust management makes attacks harder. By only allowing trusted nodes to be parents (or ancestors), it will be hard for a few malicious nodes to become parents of numerous benign nodes (as in Sybil attacks). By only allowing trusted nodes to be children, it is also hard for an attacker to launch an attack through infinite joins, in which malicious nodes try to occupy all the children slots of every existing node, thus leaving new nodes no choice except attaching to those malicious nodes.

Trust management toward key management entities is also necessary since they participate in the node authentication

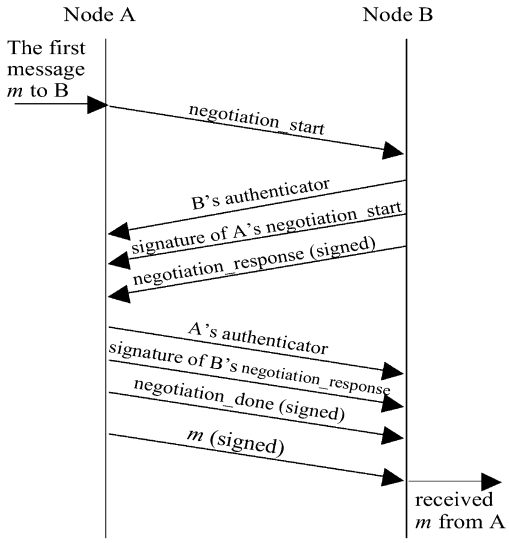

Fig. 5. Parent selection negotiation.

process (see Section IV-C-II). Similar trust management can be applied. Using certificate authorities as an example, a node can only accept certificates from a trusted certificate authority.

2) Discretionary Authentication: While a node n can determine whether it trusts another node x, node n still must be able to: 1) authenticate x's identity and 2) verify messages claimed to be from x. Both node authentication and message authentication are key to RBone security. However, due to the large scale of an RBone, we do not assume that there is a ubiquitous authentication scheme for all nodes. Every node may implement a different set of authentication schemes, perhaps with different orders of preference. In this section, we describe how Revere achieves discretionary authentication at each individual node with two key techniques: node-to-node authentication scheme negotiation and pluggable security boxes.

a) Node-to-node authentication scheme negotiation: A node must select a supported authentication scheme for exchanging messages with another node. If necessary, different schemes can be used for sending to and receiving from a given node. Choosing the appropriate schemes requires a secure negotiation. If the negotiation succeeds, proper authentication schemes can be imposed on messages exchanged between the nodes.

Authentication scheme negotiation is triggered when a node wants to send another node a message, but finds that no authentication scheme has been chosen to protect this message. Fig. 5 illustrates a negotiation procedure between nodes A and B, initiated by node A. The following is a stepwise explanation of the negotiation.

Step 1) Node A first sends a *negotiation_start* message to B in plaintext, indicating an ordered list of A's preferred authentication schemes for messages from B. Note that this can only be plaintext since A does not know what scheme B requires.

Step 2) Node B selects a scheme from A's list that B supports. Node B creates an authenticator for itself using this scheme and sends it to A. Using the signing algorithm of the selected scheme, B also sends to A a signature of A's *negotiation_start* and a signed *negotiation_response* message. The *negotiation_response* message contains the scheme that B selected and an ordered list of B's preferred authentication schemes for messages from A.

Step 3) Node A authenticates B, verifies the signature of its initial *negotiation_start* message to ensure it has not been tampered with and verifies the *negotiation_response* message. If all are verified, A chooses a scheme from B's list to protect its messages toward B. Node A sends an authenticator toward B, using the scheme that was just selected. To assure B that its response was not tampered with, A sends back a signature of the message. Node A also sends a signed *negotiation_done* message to B, indicating the scheme that A selected and ending the negotiation.

If any of these steps fail, the negotiation will fail, and no authentication scheme will be selected for communication between the two nodes. For example, if B cannot select a scheme successfully, B will not respond to A's negotiation request and A will finally time out and give up. If all steps succeed, the negotiation succeeds, and messages can begin to be forwarded from A to B (such as message m in Fig. 5), or *vice versa*, protected by using the selected authentication schemes.

During authentication scheme negotiation, a compromised node may try to trick a benign node into using a weaker scheme to verify the messages from the compromised node. This cannot succeed because the compromised node, whether the initiator or not, must use one of the schemes already specified by the benign node to authenticate itself and sign its response.

b) Pluggable security boxes: Revere implements an extensible architecture to support various authentication schemes. As in [15], each authentication scheme can be added by plugging in a corresponding security box.

A security box can be viewed as a security monitor that is responsible for node authentication and protection of RBone activities such as the join procedure, repository selection or RBone maintenance. A security box allows a node to authenticate other nodes or authenticate itself to another node, and ensures that only authentic messages will be used.

All control messages must pass through the security box. Incoming messages are accepted or rejected based on trust and authenticity. Outgoing ones are inspected and stamped with authentication information. Every control message, including heartbeats and those used in three-way-handshakes, is signed by its sender's security box and verified by its receiver's security box. (Security updates do not pass through the security boxes, since they are authenticated by a uniform method for all nodes.)

Many security box implementations are possible, each providing a different level of node authentication, message verification, replay prevention, and possibly secrecy. The level of protection provided depends entirely on the particular security box implementation.

c) A security box example: One example of a security box is based on a hierarchical infrastructure of public key certificate authorities (CA), where recursively the CA at one level (the parent) produces certificates for the next level down (the child). The public key for the CA at the root of the hierarchy is universally known.

Here, the verification of a node's public key is straightforward. Further, other nodes can authenticate the messages from this node using its public key if the messages are signed using this node's private key.

Message replay can be prevented as well. The signed portion of a message can include a random number chosen by the recipient, a standard solution to such problems.

V. Measurement

The goal of Revere is to provide a service for disseminating security updates. Dissemination speed describes the basic behavior of Revere, but we must also determine dissemination quality in the face of broken nodes to understand Revere's resiliency. In addition, understanding RBone formation and maintenance is important. As discussed in Section II, an RBone is gradually formed by a series of join procedures, which are also employed when a node needs to adjust its position during RBone maintenance. As a result, performance data on the join procedure is also key to the assessment of Revere.

A. Metrics

The following metrics are important to evaluating Revere.

- *Join latency*. The time that a new node spends becoming a participant in Revere (finding all parents).
- *Join bandwidth*. The bandwidth spent to join Revere.
- *Dissemination latency*. The latency for a security update to reach an individual Revere node. Also relevant is the time needed to reach a certain percentage of all Revere nodes.
- *RBone resiliency*. The percentage of Revere nodes that still receive security updates, given that every node has a particular probability of failure.
- *Dissemination bandwidth*. The bandwidth spent to disseminate security updates.
- *Maintenance bandwidth*. The bandwidth spent to maintain an RBone.

The last two metrics are easy to evaluate. In a single round of dissemination, the inbound dissemination bandwidth per Revere node is the size of the security update multiplied by the number of parents (under normal conditions). The RBone maintenance bandwidth per Revere node is mainly the size of heartbeat messages during each period. Both are of acceptable size.

B. Overloading Methodology

Revere is designed for large scale. Given that it is prohibitive to run empirical measurements with more than a few hundred machines, we adopted an overloading technique to measure Revere, by which a physical machine can be overloaded with multiple logical nodes, each still running the real Revere code. Using multiple machines can help achieve even larger scale measurement.

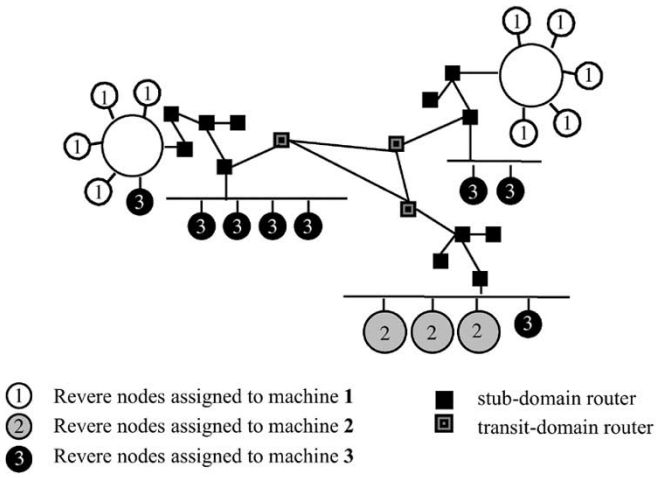

Fig. 6. A virtual topology with revere nodes.

However, this approach raises two key questions: 1) topology construction—since logical Revere nodes are overloaded on physical machines, they will have a different topology than they would in the real world and (2) resource contention—logical nodes on the same physical node must share both processor and memory, thus affecting (lengthening) the processing time of individual nodes performing particular tasks.

1) Topology Construction: A virtual topology can be employed to solve the topology problem. Each node can be viewed as attached to a particular location in a virtual topology, communicating through this virtual topology with another node in the same virtual topology. Fig. 6 is an example of 20 logical nodes (assigned to three physical machines) communicating across multiple routers in a virtual topology.

Using a virtual topology, a distributed Revere system can be created. After generating a virtual topology, each logical node in this topology is treated as an individual Revere node. For each logical node, a Revere instance is run on top of a physical machine, where multiple instances of Revere software may be invoked on the same machine.

Many results obtained in a virtual topology will not differ from those obtained by running on top of a real topology with the same structure. For example, whether the underlying topology is real or virtual, the storage cost or bandwidth cost incurred at an individual Revere node will typically be the same.

The characteristics of the communication paths between any two Revere nodes can be determined based on the specification of a virtual topology. For instance, if the length of every link in a virtual topology is known, the shortest path between any two nodes on the virtual topology can be calculated using the Dijkstra algorithm [16], instead of being measured.

2) Resource Contention: Three approaches can be taken to handling resource contention. The first approach is to remove the resource contention. If only a single node is allowed to proceed with full use of resources at a given time, the time spent by this node on a task should incur approximately the same amount of time as it would in the real world. A Revere node may have to wait for access to the resources to perform a particular task. The second approach uses a divide-and-conquer method, dividing

the task to be measured into several disjoint subtasks that are easier to measure. Here, 1) subtasks must be independent; 2) subtasks must not overlap in terms of processing latency; and 3) the sum of all subtasks must be the total processing latency. The third approach is to calculate a slow-down factor and apply that to the measured processing latency. This approach is not used in measuring Revere.

C. Measurement Procedure

Our measurements used a testbed that consisted of ten machines, overloaded with up to 3000 Revere nodes. Every machine was equipped with an AMD Thunderbird 1.333 GHz CPU, 1.5GB SDRAM, and a 100 Mb/s Ethernet interface.

Every virtual topology was created as follows. We used GT-ITM [17] to generate a router-level topology, then assigned certain numbers of Revere nodes (hosts) to each stub-domain router on that topology. Finally, a topology server assigned the same number of Revere nodes to every testbed machine.

The following configurations are used: 1) every Revere node must have **2** parents and no more than **10** children; 2) UDP is used for security update forwarding from parent to child; and 3) both security updates and RBone messages are protected using RSA-based public key cryptography with a three-level certificate authority hierarchy. Note that such a configuration is purely for providing insights on Revere performance and may not be applicable in all real situations.

We artificially divided the lifetime of Revere into three phases: the join phase, the dissemination phase, and the resiliency test phase. In real use, these three phases would overlap, but measuring them separately captures most costs appropriately. During the join phase, nodes sequentially join Revere and gradually form an RBone. The system then advances into the dissemination phase, during which the center disseminates security updates through the RBone to individual nodes for ten rounds. Finally, in the resiliency test phase, dissemination is tested in the face of broken nodes.

During the join phase, when every physical machine is overloaded with several Revere nodes, join bandwidth should be unaffected, but join latency will be artificially increased. Using the first approach from Section V-B2), we applied a token-controlled mechanism to ensure that at any time during the joining phase only one node will be in the join procedure, thus evaluating a particular scenario where all nodes join sequentially. Other nodes may be temporarily activated when requested to interact with the joining node. The results should be approximately the same as the real cost of a single-node joining.

During the dissemination phase, each node behaves in a store-and-forward manner. However, because many Revere nodes are running on a physical machine, simply measuring the interval between sending an update and receiving it cannot reflect the true dissemination latency. Given the artificially heavy load on the physical machine, both the processing delay and the kernel-space-crossing delay[1] will be lengthened.

[1]The time from invocation of sending a message from Revere at application level to the departure of the message from the node and the time from the receipt of the same message at a recipient node to the delivery of the message to Revere at application level.

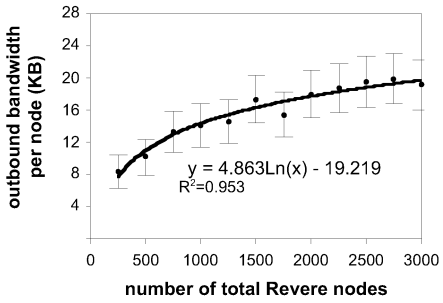

Fig. 7. Outbound bandwidth per node in joining phase (confidence level: 95%).

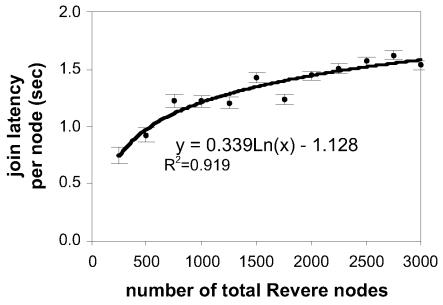

Fig. 8. Join latency per node in joining phase (confidence level: 95%).

We solved this problem using the divide-and-conquer method. The latency of disseminating an update is divided into three parts: the processing delay at each hop, the transmission delay of crossing the virtual topology, and the kernel-space-crossing delay. Each part is evaluated separately. The processing delay per hop can be measured in a separate experiment without overloading a physical node. The kernel-space-crossing delay per hop can be measured in the same way. The communication latency can be calculated using the Dijkstra algorithm over the virtual topology graph. Note that with a given RBone structure, the hops that an update travels to reach a node are invariant, no matter how many nodes are simultaneously running on the same physical node. By multiplying the processing delay per hop and kernel-space-crossing delay per hop and adding the communication latency, we can obtain a good approximation of the dissemination latency in large-scale scenarios.

During the resiliency test phase, each node on the overlay was assigned a uniform probability of failure to test how many nodes are still reachable during dissemination. The divide-and-conquer method was again used to evaluate the latency of disseminating updates toward the remaining nodes.

D. Results and Analysis

1) Join Latency and Bandwidth: Fig. 7 shows the outbound join bandwidth incurred by a node for various sizes of RBones. This bandwidth cost includes the messages that a node sends when joining an RBone and the messages sent in response to the join requests of others. Fig. 8 shows the latency experienced by a node joining RBones of various sizes. Each node completes

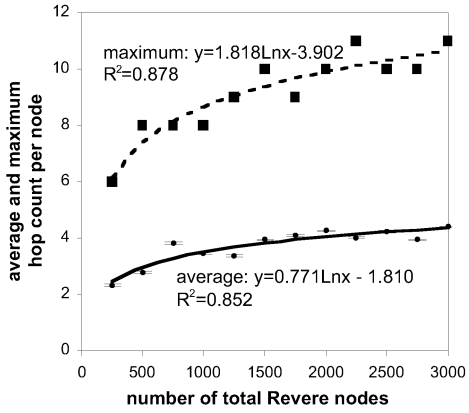

Fig. 9. Average and maximum hop count of security update dissemination (confidence level for average hop count: 99.9%).

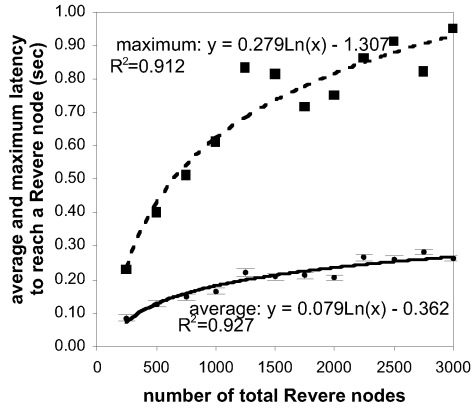

Fig. 10. Average and maximum latency of security update dissemination (confidence level for average latency: 99.9%).

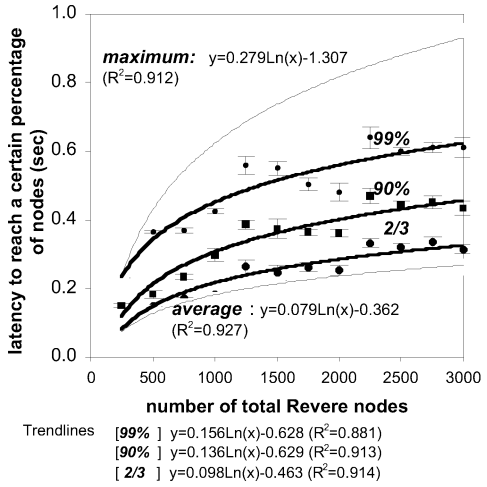

Fig. 11. The latency to reach 99%, 90%, and 2/3 of nodes in an RBone, compared with the maximum and average latency to reach a node (confidence level: 95%).

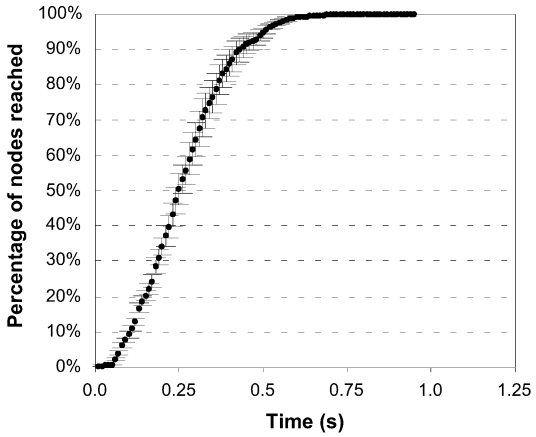

Fig. 12. Security update dissemination coverage for a 3000-node dissemination (confidence level of coverage: 99%).

the join procedure after successfully attaching itself to two acceptable Revere nodes.

The costs of both join bandwidth and join latency are acceptable, and basically follow logarithmic trends as the numbers of nodes grow. This is due to the worst-case method we adopted for searching new parents in our experiments-a top-down recursive search after a new node fails to find a local potential parent. The new node will run the three-way-handshake with the dissemination center first; if the center has no space, the new node will then negotiate with one of the children of the center; if that child is also full, it then repeats the negotiation procedure recursively.

2) Dissemination Speed: Fig. 9 shows the average and maximum hop count for disseminating security updates, Fig. 10 shows the average and maximum latency to reach a node in the various sizes of RBones, and Fig. 11 shows the latency needed to reach a certain percentage of nodes in an RBone.

Those results are based on the dissemination latency of every individual node in an RBone. Since no failure or security attacks occurred during the dissemination phase, every node used the fastest delivery path to receive the first authentic copy and this path is the one that was measured. The fastest paths for all nodes in an RBone form a tree, rooted at the dissemination center. This explains why all trendlines in Figs. 9–11 closely follow logarithmic trends when the total number of RBone nodes varies.

If one assumes that the trendlines in Figs. 9–11 continue at larger scale, in a 100-million-node RBone, it will take approximately 12 hops on average to reach a node (with a maximum of 30 hops), 1.10 s on average to reach a node, 1.34 s to reach 67% of nodes, 1.88 s to reach 90%, 2.25 s to reach 99%, and 3.83 s to reach all. In the real world, because Revere nodes are heterogeneous and every node may be configured differently, extending those trendlines smoothly would be unrealistic. Nevertheless, these results suggest a Revere-like approach has the potential to be fast and scale well.

Fig. 11 also shows that after *most* nodes have been reached, it still takes a relatively long time to reach *all* RBone nodes. This is better illustrated in Fig. 12, which depicts the dissemination coverage over time of a 3000-node RBone. Here, the "tail" of the graph corresponds to a relatively long delay for reaching all nodes after a high-percentage coverage. In practice, that pattern would be true in any case because a small percentage of nodes would be far away, turned off, or physically inaccessible.

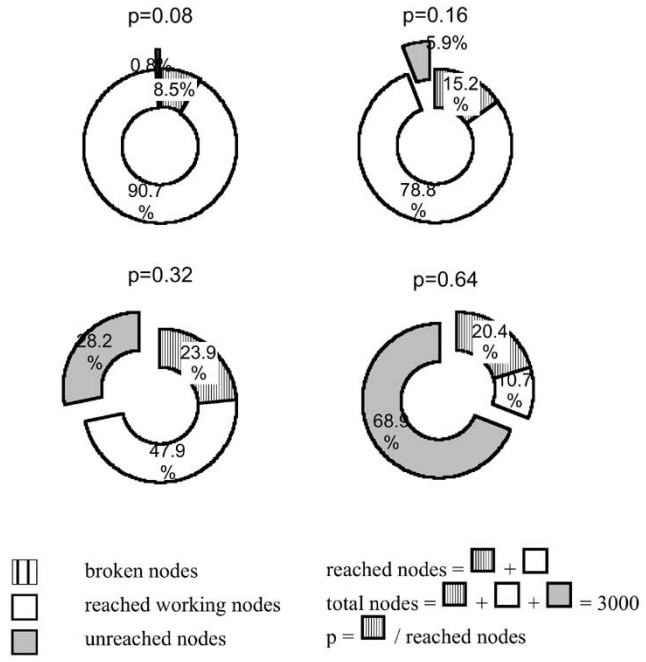

Fig. 13. RBone resiliency test with different node broken probability on a 3000-node RBone, where each figure shows the percentage of three different kinds of nodes.

3) Dissemination Resiliency: During the resiliency test phase, an RBone's resiliency was tested by assigning each node a uniform probability of being broken. Whether or not a particular node is broken is discovered at run time when an update is delivered. Upon the receipt of an update, a node queries a common random boolean server to see whether it is emulating a broken node: the server responds yes if a newly generated random number $r(0 < r < 1)$ is less than the given broken probability. If, and only if, a node is not broken will it forward the update. Some nodes thus may not be reachable.

Therefore, we have three types of nodes: (reached) *broken nodes*, *reached working nodes*, and *unreached nodes*. (The ratio of broken nodes over the total number of reached nodes should be approximately the same as the assigned broken probability.)

We measured a 3000-node RBone. Based on multiple rounds of dissemination for a given broken probability, measurement shows that the RBone is resilient to small node broken probabilities (with node broken probability lower than 2%, 100% of the rest of the nodes can still be reached). Fig. 13 shows resiliency test results for four higher failure probabilities. A comparison of the percentage of reached working nodes and the percentage of unreached nodes in Fig. 13 showcases a very resilient RBone.

Recall that in the measurement, every node has two parents. We could test a different number of parents, such as one or three. However, in the single-parent case, an RBone will be exactly a tree, whose resiliency can simply be analyzed. The three-parent case will certainly be more resilient than the two-parent case; but would be more relevant to a study of how many parents are optimal, which is not a topic this paper addresses. Here, we simply show the value of multiple parents.

4) Evaluation of Larger Scale: How would Revere perform at a scale larger than 3000 nodes? We address this question in three viewpoints. First, Revere is specifically designed for an Internet-scale environment. Revere's design should scale to a much larger network than the one we actually measured. Second, measuring the performance of any systems in an Internet-scale environment with millions of nodes is unfortunately still a very daunting task for everyone. We could try a larger-scale network, but unless we can try the largest scale, which is unlikely, we will always face the question: "What about an even larger scale?" We have to stop at some point, and 3000 was the level we could achieve. Third, we considered using simulation to evaluate Revere for an arbitrary scale, but there are a number of difficulties or disadvantages: 1) simulation would employ different software from the real code, raising questions about accuracy of the rendition; 2) simulation might not expose hidden costs and subtle timing effects; 3) simulation is expensive to develop; and 4) simulation is only worthwhile if it is validated against a real system and validating a Revere simulation (especially at high scale) would be difficult. We ourselves are not completely satisfied with the degree to which Revere scaling has been demonstrated, but we believe we have provided sufficient evidence to make a good case for Revere's scalability.

VI. Related Work

Revere's RBone overlay network is comparable to various self-organizing overlay networks that are also composed of Internet end hosts, including those used for application-layer multicast. *Yoid*, for example, tries to build a general architecture for information distribution, including a tree topology for content distribution and a mesh topology for control information distribution [8]. Revere instead relies on a single topology for both purposes, enables multipath delivery and enforces security with a different presumption of open membership. *ALMI* builds a small-scale minimum spanning tree among end hosts and relies on a central controller for tree management [10]. *End system multicast* also targets small-scale tree-structured overlay networks, but it first builds a mesh of nodes and then constructs a shortest-path tree out of the mesh [7]. *Scattercast* adopts a similar approach to *end system multicast*, while it emphasizes infrastructural support and proxy-based multicast [6]. *Bayeux* [11] uses *Tapestry* [18], an application-level routing protocol, to organize receivers into a distribution tree. *Overcast* focuses on optimizing network bandwidth when building its overlay distribution tree [9]. **A fundamental difference between RBone and these overlay networks is that RBone is not a tree-like structure**. Instead, every Revere node can choose to have multiple as-disjoint-as-possible paths to receive security updates. Also, in addition to the pushing mechanism, Revere allows each node to pull missed security updates from repositories.

In terms of building resiliency into an overlay network, Revere shares some commonalities with *RON* [19]. Instead of targeting another distribution service, RON inserts a new layer of resilient overlay network between the routing substrate below and network applications above, thus providing faster routing failure recovery and application-specific routing. One useful discovery from RON is that a failed router or physical link can be avoided if a message is routed through a different node on the RON overlay.

Multipath routing is similar to Revere's multipath message delivery [20]–[22]. However, these systems are primarily meant for load balancing or congestion avoidance and do not fully consider the disjointedness between different paths. It is also hard for these systems to address security issues (such as key distribution, replay prevention, etc.) at router level. They also face deployment problems.

Peer-to-peer computing is developing rapidly and gaining prominence as an important service [23], [24]. In some respects, the relationship between Revere nodes is also peer-to-peer and results from peer-to-peer research can be leveraged to improve the Revere overlay network.

VII. Future Work

Certainly more work is required to refine Revere's technical approach and demonstrate its feasibility. A delivery system of such scale and speed raises a number of interesting questions:

- *Adaptive redundancy*. How should a node adjust its redundancy degree for receiving security updates? Would two delivery paths be enough, for example?
- *Delivery path quality at physical level*. This question warrants study if we cannot assume that routers are fully trustworthy. In particular, there is no guarantee that if two delivery paths are disjoint at the application level, they will also be disjoint at the physical level. If not, how much overlap will there be, how can that effect be recognized and perhaps minimized?
- *Repository server selection*. Among many repositories, which ones should a node choose to query for missing security updates?
- *Security update integrity protection other than using digital signature*. While digital signature based on public key cryptography has been widely used and is also employed in Revere, could other integrity protection techniques under study benefit Revere better?
- *Secure dissemination process monitoring*. How should Revere securely monitor the dissemination process in real time? How should every individual node provide feedback?
- *Performance understanding at larger scale*. It is difficult but desirable to understand a system at very large scale. How can one deduce or extrapolate performance of this system from smaller-scale results, taking into account pragmatic deployment issues?
- *Revere in wireless environment*. With some or all nodes wireless, how does the appropriate solution change? For instance, when nodes become mobile, delivery paths will become volatile. Meanwhile, are there any elements of the wireless environment that are helpful? For instance, will node location information as reported from GPS be helpful (and critical) in determining multiple physically disjoint delivery paths?
- *What will be done with the updates once delivered?* In some cases, the answer is simple and obvious, such as installing new virus signatures into a virus detection database. In other cases, there are greater challenges. For example, system administrators today often lack confidence in automated patch installation. Reference [25] reports that with at least four vaguely defined patch installation mechanisms, Microsoft's Windows Update caused the automated scanning service to mismanage patches. In one extreme case, a patch for a customer actually removed a previous hot fix, causing that machine to be vulnerable to the Nimda virus.
- *Can an RBone be theoretically analyzed?* Singh [26] proposes a way to evaluate the global reliability of a communication network. Unfortunately, his method requires knowledge of the global topology of the network. Is there a distributed version of the algorithm where every node only has partial knowledge of the whole system?

VIII. Conclusion

Using secure, resilient, and self-organizing overlay networks, this research offers a sound solution to rapid security update delivery at Internet scale. Without relying on huge, powerful server farms, nodes in such a network not only can quickly receive pushed updates once they become available, but also can query and pull updates at any time.

Different from other overlay designs, a Revere overlay network allows a node to select multiple least-overlapping delivery paths and achieve best resiliency using a path vector concept. Also, instead of enforcing a closed membership, a Revere overlay supports open subscription. Facing challenges brought by such differences, Revere employs a self-organizing capability to cope with complexities in a dynamic large-scale environment.

Revere protects both the delivery procedure and the delivery structure. For the former, digital signature in security updates, redundancy in both push and pull, and the key invalidation mechanism allow a node to receive authentic updates once they are available. For the latter, discretionary authentication mechanisms (node-to-node authentication scheme negotiation and pluggable security box) and trust management together ensure the whole delivery structure is robust.

As a service that delivers information at application level, Revere demonstrates that an application-level Revere-like service is feasible and can be made effective without changing underlying hardware, operating systems, or network infrastructures. Further, Revere shows an interesting phenomenon in its incremental deployment: not only can Revere be easily deployed (every node can simply run Revere software to become a Revere node), but Revere also offers more attractive benefits to potential participants as more nodes exist in the system and form a larger information pool.

Acknowledgment

The authors would like to thank their colleagues at the UCLA Laboratory for Advanced Systems Research, and the reviewers and editor for many valuable comments. The authors also thank Kluwer Academic Publishers who published our earlier work on this topic.

REFERENCES

[1] S. Staniford, V. paxson, and N. Weaver, "How to own the Internet in your spare time," presented at the 11th USENIX Security Symp., San Francisco, CA, Aug. 2002.

[2] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 33–39, July–Aug. 2003.

[3] R. Gruber, B. Krishnamurthy, and E. Panagos, "The architecture of the READY event notification service," in *Proc. 19th IEEE Int. Conf. Distributed Computing Systems*, Austin, TX, May 1999, pp. 108–113.

[4] B. Krishnamurthy and D. Rosenblum, "Yeast: A general purpose event-action system," *IEEE Trans. Software Engineering*, vol. 21, pp. 845–857, Oct. 1995.

[5] McAfee ASaP's Rumor technology. [Online]. Available: http://www.mcafeeasap.com/content/virusscan_asap/rumor.asp

[6] Y. Chawathe, "Scattercast: An architecture for Internet broadcast distribution as an infrastructure service," Ph.D. dissertation, Elec. Eng. Comput. Sci. Dept., Univ. California, Berkeley, CA, Dec. 2000.

[7] Y. Chu, S. Rao, and H. Zhang, "A case for end system multicast," in *Proc. ACM Sigmetrics*, June 2000, pp. 1–12.

[8] P. Francis. (2000) Yoid: Your Own Internet Distribution. [Online]. Available: http://www.aciri.org/yoid

[9] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole Jr, "Overcast: Reliable multicasting with an overlay network," in OSDI, 2000, pp. 197–212.

[10] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, "ALMI: An application level multicast infrastructure," in Proc. 3rd Usenix Symp. Internet Technologies & Systems (USITS), Mar. 2001.

[11] S. Zhuang, B. Zhao, A. Joseph, R. Kata, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in Proc. NOSSDAV, 2001.

[12] A. Rosenstein, J. Li, and S. Tong, "MASH: The multicasting archie server hierarchy," *Computer Communication Review*, vol. 27, no. 3, pp. 5–13, July 1997.

[13] J. Li, P. Reiher, and G. Popek, *Disseminating Security Updates at Internet Scale*. Norwell, MA: Kluwer, 2002.

[14] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang, "IDMaps: A global internet host distance estimation service," *IEEE/ACM Trans. Networking*, vol. 9, pp. 525–540, Oct. 2001.

[15] J. Li, M. Yarvis, and P. Reiher, "Securing distributed adaptation," *Comput. Networks, Special Issue on Programmable Networks*, vol. 38, no. 3, pp. 347–371, 2002.

[16] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[17] K. Calvert, M. Doar, and E. Zegura, "Modeling Internet topology," *IEEE Comm. Magazine*, vol. 35, pp. 160–163, June 1997.

[18] Tapestry: Fault-resilient wide-area location and routing. [Online]. Available: http://www.cs.berkeley.edu~ravenben/tapestry

[19] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris, "Resilient overlay networks," in SOSP, 2001.

[20] J. Chen, P. Druschel, and D. Subramanian, "An efficient multipath forwarding method," in *Proc. INFOCOM*, San Francisco, CA, Mar.–Apr. 1998, pp. 1418–1425.

[21] S. Murthy and J. J. Garcia-Luna-Aceves, "Congestion-oriented shortest multipath routing," in *Proc. INFOCOM*, San Francisco, CA, Mar. 1996, pp. 1028–1036.

[22] W. T. Zaumen and J. J. Garcia-Luna-Aceves, "Loop-free multipath routing using generalized diffusing computations," in *Proc. IEEE INFOCOM*, San Francisco, CA, Mar.–Apr. 1998, pp. 1408–1417.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *ACM SIGCOMM 2001*, Aug. 2001, pp. 161–172.

[24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in SIGCOMM 2001, Aug. 2001, pp. 149–160.

[25] D. Fisher, "Security tool leaves holes," *Eweek-the Enterprise Newsweekly*, vol. 19, no. 16, April 2002.

[26] B. Singh, "A global reliability evaluation: Cutset approach," *IETE Tech. Rev.*, vol. 12, no. 4, pp. 275–278, July–Aug. 1995.

Jun Li (S'02–M'04) was born in Changzhi, Shanxi, China, in 1970. He received the B.S. degree in computer science from Peking University, Beijing, China, in 1992, the M.E. degree in computer software from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 1995, and the Ph.D. degree in computer science from the University of California (UCLA), Los Angeles, in 2002.

He is now an Assistant Professor in the Department of Computer and Information Science, University of Oregon, Eugene. He coauthored *Disseminating Security Updates at Internet Scale* (Norwell, MA: Kluwer, 2002), and several published papers, including "Securing Distributed Adaptation" in Computer Networks (Special Issue on Programmable Networks, 2002), and "SAVE: Source Address Validity Enforcement Protocol" (IEEE INFOCOM 2002). His research interests include network security, Internet protocols and applications, network performance analysis, and distributed systems.

Dr. Li received the Outstanding Doctor of Philosophy in Computer Science Award from UCLA in 2002. He has served on several committees and panels, including the program committee for the New Security Paradigm Workshop in 2000 and 2001, and the National Science Foundation (NSF) information technology research program in 2003.

Peter L. Reiher (M'02) was born at Ft. Ord, CA, in 1957. He received the B.S. degree in electrical engineering from the University of Notre Dame, South Bend, IN, and the M.S. and Ph.D. degrees in computer science from the University of California (UCLA), Los Angeles, in 1984 and 1987, respectively.

He is currently an Adjunct Associate Professor in the Computer Science Department, UCLA. Prior to that, he was Chief Designer for the Time Warp Operating System at the Jet Propulsion Laboratory, Pasadena, CA. He is the coauthor of two books on computer security and distributed systems, and has authored or coauthored over 70 papers. His research interests include network security, distributed systems, advanced operating systems, and parallel discrete event simulation.

Dr. Reiher has been the principal investigator on several National Science Foundation (NSF) and Defense Advanced Research Projects Agency (DARPA) projects involving computer network security, and has participated in many relevant conferences and programs, such as the 13th Wireless Network Security Cluster Group, the DARPA SUMOWIN program, and the FAA secure baggage handling and tracking program.

Gerald J. Popek was born in Passaic, NJ, in 1946. He received the B.S. degree in nuclear engineering from New York University, New York, in 1968, and the M.S. and Ph.D. degrees in applied mathematics from Harvard University, Cambridge, MA, in 1970 and 1973, respectively.

He is currently an Adjunct Professor in the Computer Science Department, University of California (UCLA), Los Angeles, and also Chief Technology Officer for United Online, a major Internet service provider. Prior positions included CTO of Platinum Technology, Inc., and founder and Chairman of Locus Computing Corporation. He is the coauthor of several books, including *The LOCUS Distributed System Architecture* (Cambridge, MA: MIT Press, 1985) and over 100 professional articles. His research interests are in the areas of computer security, system software, and computer architectures. He is responsible for the creation of the Locus Operating System—the core system software for Hewlett Packard's telecommunications product line and used by several telephone companies to manage large-scale operations. He has been a member of the board of directors of a number of technology companies and a consultant to several major technical organizations.