FirePatch: Secure and Time-Critical Dissemination of Software Patches*

Håvard Johansen[1], Dag Johansen[1], and Robbert van Renesse[2]

[1] University of Tromsø, Norway. `haavardj|dag@cs.uit.no`
[2] Cornell University, USA. `rvr@cs.cornell.edu`

Abstract. Because software security patches contain information about vulnerabilities, they can be reverse engineered into exploits. Tools for doing this already exist. As a result, there is a race between hackers and end-users to obtain patches first. In this paper we present and evaluate *FirePatch*, an intrusion-tolerant dissemination mechanism that combines encryption, replication, and sandboxing such that end-users are able to win the security patch race.

1 Introduction

Automatic software updates for bug fixes are essential for Internet applications. It is particularly important when a software update fixes a security hole. Software vendors, for fear of liability, release patches for security holes as soon as possible. They do so without publicizing what the bug is, for fear that hackers will exploit the vulnerability before end-users have an opportunity to install the patch.

In practice the time between when a patch is released to the time that it is installed is long and typically measured in days [1, 6]. A counterintuitive observation is that a long patching cycle is worse than no patching cycle at all. This paradox stems from the fact that a security patch can be reverse-engineered to reveal the vulnerable code. In other words, if the software vendor cannot provide the mechanism to distribute *and* install a patch quickly, the end user might be better of if the patch is not released at all.

Even if users are notified about a vulnerability and are able to download a patch in time, installing a patch is an inconvenience and might lead to downtime of critical services. Patches might also contain bugs that break system configuration or introduce new vulnerabilities. It has even been suggested that patch installation should be delayed until the risk of penetration is greater than the risk of installing a broken patch [2].

Fortunately, protection against security vulnerabilities can be done in the network layer by installing stateful packet filters like Shields [14], Self-Certifying Alerts [4], or vulnerability-specific predicates [9] that inspect and modify incoming packets. Such patches do not interrupt the execution of applications and

* This work is supported in part by the Research Council of Norway (IKT 2010 program), AFOSR, AFRL/IFSE, NSF, and Intel Corporation.

are a viable intermediate solution until the user is able to install a permanent fix to the software. Also, automatic patching infrastructures have emerged that greatly reduce the time software is left vulnerable. For instance, a recent study on the Microsoft Windows Update mechanism [6] shows that the automation of notification, downloading, and installation of patches ensures that as much as 80% of the end-clients are updated within one day of patch release.

This still gives a malicious agent ample time to construct and execute an attack. For instance, by examining the binary difference between a vulnerable version of the Microsoft Secure Socket Layer (SSL) library and a corresponding patch, Flake [5] constructed a program that reliably exploited this vulnerability within 10 hours. Marketplaces for buying and selling exploits already exist [12]. It is therefore imperative that software vendors disseminate patches with low end-to-end latency. Such a patch dissemination service must be resilient to *denial-of-service* (DoS) attacks and intrusions as hackers might target the service to increase their opportunity to exploit the vulnerabilities exposed by the patches.

This paper describes *FirePatch*, a scalable and secure overlay network for disseminating security patches. *FirePatch* employs the following three techniques:

1. A patch is disseminated in two phases. First, an encrypted version of the patch, which cannot be reverse engineered, is disseminated. Some time later, the decryption key is disseminated. As the key will typically be significantly smaller than the patch, it can be disseminated much faster to a large collection of machines.
2. In order to deal with DoS attacks against dissemination of patches, attempting to increase the time during which a vulnerability can be exploited, we have developed a distributed software mirroring service. While replication makes DoS attacks more difficult, it increases the likelihood that individual servers are compromised—a highly undesirable situation for a server that disseminates security patches to clients. Therefore, our service is also made tolerant of Byzantine failures.
3. For machines that are not on-line at the time that a patch is disseminated, we have developed a simple protocol for secure download and installation of patches, run each time a machine goes on-line. While this goes on, a packet filter prevents the machine from participating in other network communication.

The rest of this paper is organized as follows. In the next section we present related work. In Section 3 we outline the architecture of *FirePatch* and state our assumptions. Section 4 describes our two-phase dissemination protocol which we use in our dissemination overlay described in Section 5. *FirePatch* is evaluated in Section 6. Section 7 concludes.

2 Background and Related Work

A study done on several software vulnerabilities appearing in the last half of the 1990's [1] found that almost all intrusions can be attributed to vulnerabilities known by both the software vendor and by the general public and to which patches existed. The study found that vulnerable software remained unpatched for months or even years. The primary reason for such long patching cycles was, the authors claim, that the studied software was not enrolled with an automatic updating service. Instead, end-users were required to discover the existence of both vulnerabilities and patches on their own by browsing the vendors web-sites, visiting bulletin-boards, etc.

With approximately 300 million clients, Microsoft Windows Update is currently the world's largest software update service [6]. The service consists of a (presumably large) pool of servers that clients periodically pull for updates. Other commercial patch management products like ScriptLogic's Patch Authority Plus[2] and PatchLink Update[3] enable centralized management of patch deployment on the Windows platform. However, it is unclear how any of these systems protect themselves from intrusion and if they address the possibility that hackers reverse-engineer patches into exploits.

Open-source communities, like the Debian GNU/Linux Project[4], organize their software update services similarly to Windows Update as a pool of servers that clients periodically pull for updates. Clients can freely choose which server to pull. The servers are organized into a hierarchy with children periodically querying their parent for updates. As these communities rely on donated third party hosting capacity, an attacker can easily intrude into the server pool.

The ratio of how often a patch is released and how quickly it must be received by clients implies substantial overhead for pull-based retrieval mechanisms like those used in the above systems. Pushing is better suited for this type of messaging, but incurs overhead to maintain an up-to-date list of clients. Peer-to-peer content distribution systems, like SplitStream, Bullet, and Chainsaw [3, 10, 11] approach this by spreading both maintenance and forwarding load to all clients. Although the elimination of dissemination trees in Chainsaw makes it more robust to certain failures than SplitStream and Bullet, these systems do not tolerate Byzantine failures. SecureStream [7] provides Byzantine tolerant dissemination by layering a Chainsaw-style gossip mesh on top of our *Fireflies* membership protocol [8] similarly to *FirePatch*. However, SecureStream targets multimedia streaming, which allows for certain packet loss.

A promising approach to detecting vulnerabilities in existing software is to use machine clusters that emulates a large number of independent hosts in order to attract attacks. Such "honeyfarms" have been shown to be able to emulate the execution of real Internet hosts in an scalable manner [13] and can be used

[2] `http://www.scriptlogic.com/products/patchauthorityplus/`
[3] `http://www.patchlink.com/`
[4] `http://www.debian.org/`

to generate self-certifying alerts (SCAs) [4] automatically upon detection of intrusion.

3 Architecture and Assumptions

We distinguish three roles: *patchers*, *clients*, and *mirrors*. Patchers are typically software providers that issue patches. For simplicity, we will assume a single patcher in this paper, although any number of patchers is supported. Clients are machines that run software distributed by the patcher, mirrors are servers that store patches for clients to download, and notify clients when a new patch is available.

We assume that the patcher is correct and is trusted by all correct clients. In particular, using public key cryptography clients can ascertain the authenticity of patches. In our system, clients are passive participants, and in particular do not participate in the dissemination system. Thus we do not have to assume that clients are correct.

In order to increase the patcher's upload capacity and ability to fight attacks, we employ a distributed network of mirror servers. The more mirrors, the harder it is to mount a DoS attack against the network. However, the easier it is to compromise one or more mirrors. We allow a subset of mirrors to become compromised, but assume that individual compromises are independent of one another, and that the probability that a mirror is compromised is bounded by a certain P_{byz}. However, we do allow compromised mirrors to collude when mounting an attack.

The patcher publishes (and signs) the list of servers that it considers mirrors for its patches. This list contains a version number so the patcher can securely update this list when necessary.

We assume that all communication goes over the Internet, the shortcomings of which are well-known. In order to deal with spoofing attacks, all data from the patcher is cryptographically signed, and we assume that the cryptographic building blocks are correct and the private key is securely kept by the patcher.

4 Two-Phase Dissemination

We refer to the time from when a software vulnerability is first made public to when the number of exploitable systems shrinks to insignificance as the *window of vulnerability*, or WOV for short. We have devised a dissemination protocol that, when layered on top of a secure broadcast channel, makes the WOV independent of message size. The net result of such an invariant is that the WOV can be kept fixed and small despite the fact that voluminous data has to be transferred over the wire.

We disseminate patches (or any data) in two phases. In phase one, we distribute an encrypted patch, and in the second phase, we disseminate the small

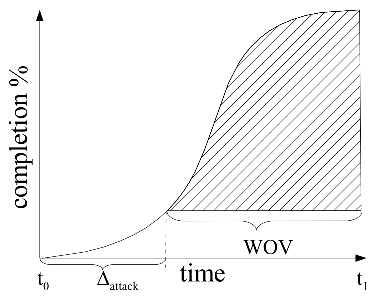

Fig. 1. Cleartext dissemination

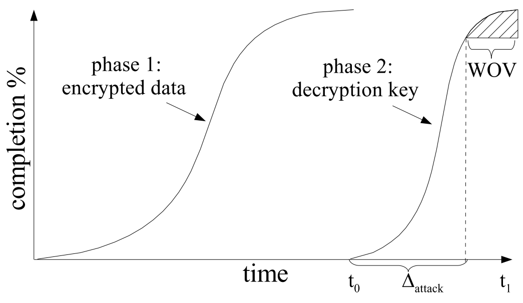

Fig. 2. Two-Phase dissemination

fixed size decryption key. More formally, our general applicable protocol is specified as follows. Let d be a message that a source s wants to disseminate to a set of clients. In the first phase, s generates a symmetrical encryption key K and a unique identifier UID, and broadcasts a ⟨ENVELOPE, UID, K(d)⟩ message, signed by s. Upon receipt and verification of the signature, a client stores this message locally. In the second phase, s broadcasts ⟨KEY, UID, K⟩ to all clients. Upon receipt, clients can decrypt the ENVELOPE message. The UID contains a version number so clients can distinguish newer from older versions of patches.

If t_0 is the time when the first client receives a patch p, if t_1 is the time when the last client receives p, and if Δ_{attack} is the time needed by an attacker to reverse engineer p into a workable exploit, then, as illustrated in Fig. 1, the WOV opens at time $t_0 + \Delta_{\text{attack}}$ and closes at time t_1. In traditional dissemination the size of the patch determines the length of the WOV. The advantage of the two-phase dissemination scheme is, as illustrated in Fig. 2, that the WOV only depends on phase two. That is, the dissemination of a small fixed size decryption key.

The time between the two phases is a policy decision. One extreme is to do the second phase immediately when the first phase completes. This would require a mechanism by which the patcher detects when all recipients have received the encrypted patch and are ready to install it. However, this is not a viable approach as disconnected clients can delay the completion arbitrarily. More alarmingly, malicious clients can prevent the second phase for happening by never acknowledging receipt. A better scheme is to start phase two some configured time after phase one is initiated. For instance, in the Windows Update system, a 24 hour time period between the phases would allow at least 80% of the clients to receive the encrypted patch [6].

5 Secure Dissemination Overlay

As mentioned before, *FirePatch* employs a network of mirrors to increase the patcher's upload capacity and to fight DoS attacks. The mirrors form a superpeer-like network structure [15] to which clients connect. Thus, the patcher

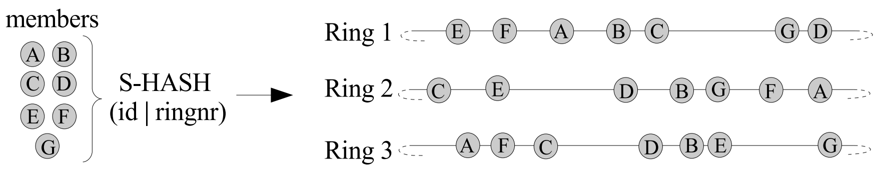

Fig. 3. *Fireflies* membership with three rings

does not broadcast patches and keys directly to the clients, but instead to the collection of mirrors. The mirrors forward this information to all clients that are currently connected to the Internet, and provides it on demand to clients that connect to the Internet at a later time. Each client connects to a minimum number of mirrors such that at least one mirror is correct with high probability.

5.1 Mirror Mesh

An attacker might be in control of one or more mirrors. Such Byzantine mirrors are not bound to any overlay protocol and might display arbitrary and malicious behavior. Although cryptographic signatures prevent Byzantine mirrors from modifying or inserting patches, they can still mount a DoS attack by neglecting to forward data. Our approach to fight such attacks is to ensure that the dissemination overlay contains sufficient link redundancy and link diversity such that, with high probability, there exists at least one path of only correct mirrors from the patcher to each correct mirror and to each correct client.

For this we build on *Fireflies* [8]—our intrusion-tolerant membership protocol that provides to each member a reasonably current view of all live members. *Fireflies* ensures, with high probability, that malicious members cannot keep crashed members in the view of live members, or live members out of these views. For this, members monitor one another and issue *accusations* (failure notices) whenever a member is suspected to have failed. If a member is falsely accused, it has the opportunity to issue an *rebuttal* before it is removed from the views of correct members.

Accusations and rebuttals are disseminated to all member using a *secure broadcast channel*, which is constructed by organizing the members in k circular address spaces, or rings. Each ring is a pseudo-random permutation of the membership list and is calculated deterministically from the secure hash of the members' identities in combination with a ring identifier. A ring defines successor and predecessor relationships between the members such that each member has k successors and k predecessors. As an example, consider the seven members A through G in Fig. 3 hashed into three rings. The successors of C are $\{G, E, D\}$, and its predecessors are $\{B, A, F\}$. Each member exchanges notes and accusations with its successor in each ring.

The number of rings, k, determines the probability that the resulting submesh of correct members is connected such that Byzantine members cannot successfully execute omission attacks. It turns out that k grows logarithmically with the number of members [8]. For instance, if one-third of the members are Byzantine in a network of 1000 members, then k should be at least 14. With $1,000,000$ members, k should be at least 19.

5.2 Data Dissemination

FirePatch reliably disseminates patches by an efficient flooding protocol on the neighbor mesh created by *Fireflies*, much like ChainSaw [11]. First, a patch is split into a set of fixed sized blocks that are individually signed by the patcher and disseminated through the mesh. A mirror m_1, upon receiving block b, notifies all of its neighbors by sending them a ⟨BLOCK-NOTIFY, block-id⟩ message, where block-id is the signature of the block. Upon receiving this notification, m_2 can request this block by issuing a ⟨BLOCK-REQUEST, block-id⟩ message to m_1. m_1 then responds with a ⟨BLOCK, block⟩ message containing the requested block. Upon receiving the block, m_2 verifies the signature and stores the block locally. m_2 then notifies all its neighbors, except m_1 that it has received the block.

To enable clients to reassemble the patch from the blocks, the patcher disseminates a signed ⟨PATCH, UID, block-id list⟩ message, where UID is the unique patch identifier. Upon receiving such a message for the first time, a mirror forwards it immediately to all its neighbors except the neighbor from which the message was received. Finally, after some time, the patcher reveals the content of the patch by disseminating a signed ⟨KEY, UID, key⟩ message. These messages are disseminated similarly to the BLOCK-NOTIFY and PATCH messages. Figure 4 summarizes the *FirePatch* dissemination protocol.

To run this protocol, each mirror maintains a TCP connection to each of its neighbors. Mirrors strive to keep all connections busy downloading missing blocks while trying to minimize the number of redundant blocks that they both send and receive. For this we use two techniques. The first technique is to randomize the order in which BLOCK-NOTIFICATION messages are sent. This helps disperse the block randomly upstream from the patcher such that mirrors are able to request different blocks from different neighbors. This is particularly important during the initial phase of the dissemination. The second technique is to schedule block requests randomly such that a request for the same block is not made to more than one neighbor unless some timeout has expired and the other connections are not busy.

5.3 Disconnected Nodes

A problem with the approach so far is that not all clients may be up and connected to the Internet at the time that the patch is being disseminated. When at some later time such a client connects to the Internet, it is vulnerable as hackers have now had ample time to create an exploit and may be lurking on such clients. We thus need a protocol for connecting clients to get the patches it is missing without being compromised.

```
on receive ⟨BLOCK, block⟩ from m:
  blockid = block.signature
  if  blockid in missingBlocks:
    blockStore.add(blockid, block)
    missingBlocks.remove(blockid)
    for patch in patches:
      if patch.completed(): decrypt_and_install(patch)
    for n in neighbors:
      if n != m: send ⟨BLOCK-NOTIFY, blockid⟩ to n
  schedule_next_request(m)

on receive ⟨BLOCK-NOTIFY, blockid⟩ from m:
  if not blockid in blockStore: availableBlocks[m].add(blockid)

on receive ⟨BLOCK-REQUEST, blockid⟩ from m:
  if blockid in blockStore:
      send ⟨BLOCK, blockStore[blockid]⟩ to m

on receive ⟨PATCH, uid, blockList⟩ from m:
   if not uid in patches:
     patches.add(uid, blockList)
     for blockid in blockList: missingBlocks.add(blockid)
     for n in neighbors:
       if n != m: send ⟨PATCH, uid, blockList⟩ to n

on receive ⟨KEY, uid, key⟩ from m:
   patches[uid].setKey(key)
   if patches[uid].completed(): decrypt_and_install(patches[uid])
   for n in neighbors:
       if n != m: send ⟨KEY, uid, key⟩ to n

proc schedule_next_request(m)
  queue = randomize( missingBlocks ∩ availableBlocks[m])
  next_request = queue[0]
  for blockid in queue:
    if blockid not requested: next_request = blockid; break
  send ⟨BLOCK-REQUEST, next_request⟩ to m
```

Fig. 4. Pseudo-code for the *FirePatch* dissemination protocol

Our approach is as follows. When running, clients store the list of all mirrors (disseminated by the patcher just like patches and keys) on disk. When a client connects, a local firewall is initially configured to block all network traffic except certain message formats to and from the mirrors selected at random from the stored list. A client connects to a minimum number of mirrors in order to make it likely that at least one of the mirrors is correct . If all clients connect to all mirrors an unreasonable load might ensue on the mirrors.

First, the client sends a $\langle$RECOVER, $v\rangle$ message to each selected mirror, where v is the version of the latest installed patch at the client. Each mirror responds with notifications of the missing patches as in the protocol described above for connected clients, and the client proceeds to download the necessary patches and keys while all other messages are ignored and dropped. When completed, the client reconfigures its firewall to allow arbitrary communication.

6 Evaluation

Our prototype implementation[5] is written in Python and has been evaluated on a local cluster consisting of 36 3.2 GHz Intel Prescott 64 machines with 2 GB of RAM. The machines were connected by a 1 Gbit Ethernet network. We ran 10 mirrors on each machine for a total 360 mirrors. In addition, one dedicated machine was used to run a mirror that acted as the patcher. To limit the effect of network congestion, the outbound bandwidth of each agent was, using a hierarchical token bucket, limited to a rate of 500 kB/s with a max burst size of 1 MB. In addition, each agent divided its total bandwidth equally amongst all its active neighbors. In all experiments we used $k = 9$ rings, resulting in each mirror having 18 neighbors. Hence, bandwidth between two mirrors was approximately 28 kB/s.

In our first experiment we measured the effect of the block size on the end-to-end latency. Our experiment consisted of injecting 2 MB patches with block sizes varying between 4 kB and 2 MB. We used a 240 second delay between consecutive patches to prevent interference. A 20 B decryption key was released after a fixed delay of 180 seconds after each patch. To achieve acceptable 95% confidence intervals, we repeated each experiment 20 times.

Figure 5 shows the resulting average total dissemination time[6]. As can be seen from the figure, the block size has a significant impact on the end-to-end latency. As expected, the messaging overhead increases with the number of blocks. Also, as the block size increases, the efficiency of our randomized block selection algorithm decreases, producing more duplicate messages and hence a longer dissemination time. We observe that in our set-up the optimal block size is between 16 kB and 64 kB.

Next we tested *FirePatch*'s resilience to attacks from an increasing fraction of Byzantine mirrors in both phase-one and in phase-two of our dissemination protocol. We fixed the block size at 32 kB and repeated the previous experiment with the fractions of Byzantine mirrors varying between 0% and 20%. Each Byzantine mirror was configured to execute omission attacks by notifying block arrivals but not responding to block requests from neighbors. Byzantine mirrors were chosen randomly from the list of all mirrors. In all our experiments Byzantine mirrors were not able to prevent correct mirrors from completing either phase-one or phase-two.

[5] The source code is available on `http://sourceforge.net/projects/fireflies`.

[6] The measured 95% confidence intervals were small and are left out for clarity.

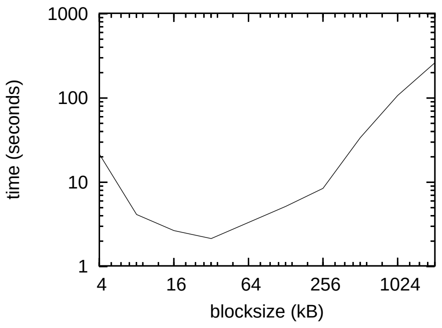

Fig. 5. Effect of the block size on dissemination

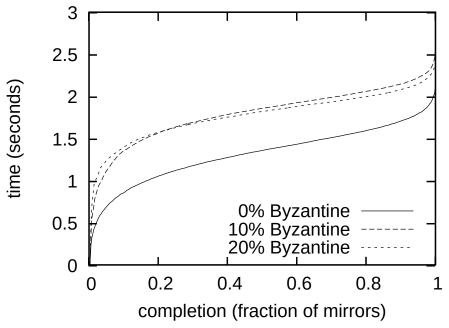

Fig. 6. Time to complete phase-one

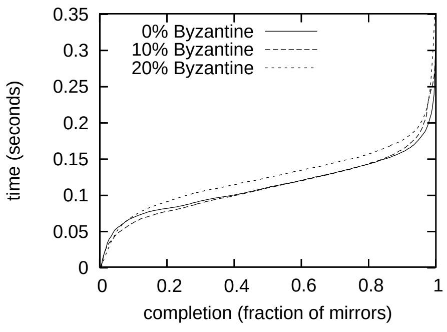

Fig. 7. Time to complete phase-two

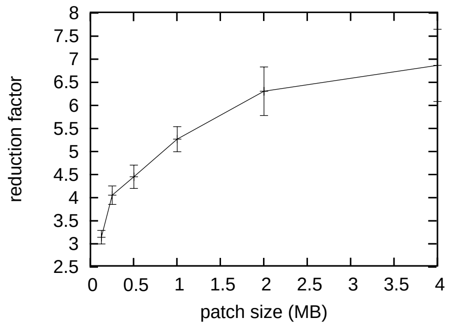

Fig. 8. WOV reduction due to two-phase dissemination

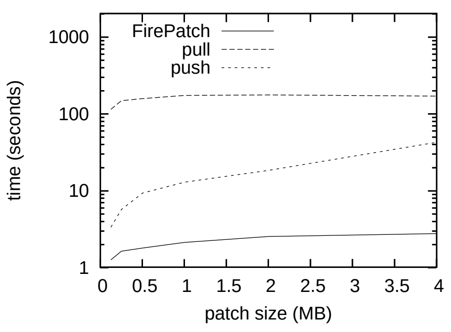

Fig. 9. Comparing naïve pull and push

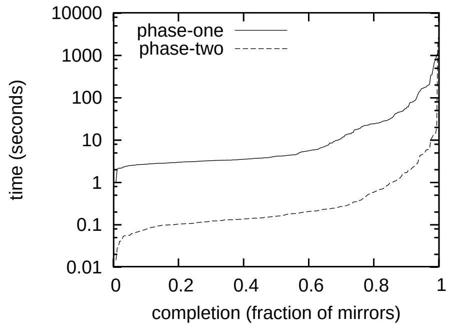

Fig. 10. Dissemination on PlanetLab

Figure 6 shows the resulting average time for an increasing fraction of the mirrors to complete phase-one of our protocol. As expected, the graph displays a clear gossip-like behavior by starting slow, speeding up, then ending slow. When under attack by 20% of the mirrors, we observed a delay of less than 1 second compared to when all mirrors were correct. This indicates that *FirePatch* is highly resilient to omission attacks. Note that for larger systems we expect the dissemination time to grow logarithmically in the number of mirrors because the diameter of the *Fireflies* mesh grows logarithmically.

Figure 7 shows a similar graph of the completion of phase-two. As expected, the dissemination of the smaller decryption key in phase-two is significantly faster than for the larger sized patch in phase-one. Also, omission attacks had little impact. Figure 8 shows the reduction of the WOV size due to our two-phase dissemination protocol when the patch size varies between 128 kB and 4 MB.

Next we compare our phase-one dissemination protocol with naïve push and pull mechanisms. For the push mechanism we modified our code such that mirrors transmitted the blocks instead of block notifications. To implement a pull mechanism we modified our block request scheduler such that it would not make more than one request for a block unless a static timeout of 20 seconds had expired. The performance of pull, push, and *FirePatch* dissemination for varying patch sizes is shown in Fig. 9.

To test *FirePatch* in a more realistic environment, we deployed our code on PlanetLab[7]. We set the fraction of Byzantine mirrors to 20% and removed the bandwidth limitation. Figure 10 shows the result of one experiment that we ran on the 30th of October 2006 where a 2 MB patch and a 20 B key were disseminated in a mesh of 279 mirrors. In this particular setup 80% of the mirrors had completed phase-one within 24 seconds and phase-two within 0.58 seconds. However, we also observed that a few mirrors used a significantly longer time. It turned out that these mirrors had become unresponsive due to heavy CPU and network load from other projects. This was particularly noticeable during phase-two where all but two mirrors received the key within 19 seconds. The last two mirrors became unresponsive between the phases but reintegrated themselves into the mesh and completed phase-two one hour later. Because each client connects to multiple mirrors, such outages will not prevent clients from receiving updates.

7 Conclusion

We have investigated a secure approach to distribute software security updates in a partially connected Internet environment, combining encryption, replication, and sandboxing upon reconnection of disconnected computers. Our findings are intuitive, but are highly effective.

We have demonstrated that an intrusion-tolerant overlay substrate can be used to scale the system without adding trusted mirrors. Notice that our two-phase dissemination protocol has wide and general applicability. We conjecture that the protocol can be incorporated easily into existing large-scale software patching schemes. It also enables secure peer-to-peer distribution of virus definition files.

[7] `http://www.planet-lab.org/`

References

1. William A. Arbaugh, William L. Fithen, and John McHugh. Windows of Vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59, 2000.
2. Hilary K. Browne, William A. Arbaugh, John McHugh, and William L. Fithen. A trend analysis of exploitations. In *Proc. of the 2001 IEEE Symp. on Security and Privacy*, pages 214–229, 2001.
3. Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 298–313, 2003.
4. Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, pages 133–147, 2005.
5. Halvar Flake. Structural comparison of executable objects. In *Proc. of the 2004 Conf. on Detection of Intrusions and Malware & Vulnerability Assessment*, Lecture Notes in Informatics, pages 161–173, 2004.
6. Christos Gkantsidis, Thomas Karagiannis, Pablo Rodriguez, and Milan Vojnović. Planet scale software updates. *ACM SIGCOMM Computer Communication Review*, 36(4):423–434, 2006.
7. Maya Haridasan and Robbert van Renesse. Defense against intrusion in a live streaming multicast system. In *Proc. of the 6th IEEE Int. Conf. on Peer-to-Peer Computing*, pages 185–192, 2006.
8. Håvard Johansen, André Allavena, and Robbert van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proc. of the 1th ACM Eurosys*, pages 3–13, 2006.
9. Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, pages 91–104, 2005.
10. Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 282–297, 2003.
11. Vinay S. Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proc. of the 4th Int. Workshop on Peer-to-Peer Systems*, volume 3640 of *Lecture Notes in Computer Science*, pages 127–140, 2005.
12. Brad Stone. A lively market, legal and not, for software bugs. *The New York Times*, online, January 30 2007. `http://www.nytimes.com/2007/01/30/technology/30bugs.html`.
13. Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, pages 148–162, 2005.
14. Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of the 2004 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 193–204, 2004.
15. Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *Proc. of the 19th IEEE Int. Conf. on Data Engineering*, pages 49–60, 2003.