Avoiding Theoretical Optimality to Efficiently and Privately Retrieve Security Updates

Justin Cappos
NYU Poly
jcappos@poly.edu

Abstract. This work demonstrates the feasibility of building a PIR system with performance similar to non-PIR systems in real situations. Prior Chor PIR systems have chosen block sizes that are theoretically optimized to minimize communication. This (ironically) reduces the throughput of the resulting system by roughly 50x. We constructed a Chor PIR system called upPIR that is efficient by choosing block sizes that are theoretically suboptimal (from a communications standpoint), but fast and efficient in practice. For example, an upPIR mirror running on a three-year-old desktop provides security updates from Ubuntu 10.04 (1.4 GB of data) fast enough to saturate a T3 link. Measurements run using mirrors distributed around the Internet demonstrate that a client can download software updates with upPIR about as quickly as with FTP.

1 Introduction

Each year, thousands of vulnerabilities in software are discovered and fixed. To fix a vulnerability, a computer will request and install a security update. However, the request to retrieve a security update is very much a public action. Most software updaters do not encrypt the request for a security update in any way and the request itself is often directed to an untrustworthy party like a mirror. For example, Cappos [1] set up an official mirror for popular Linux distributions using dubious credentials and rented hosting. The official mirrors received requests for security updates (and thus a notification that the requesting system is unpatched) from a large number of computers including banking, government, and military computers. Thus the act of fixing a security vulnerability ironically also notifies potential attackers that the client has a security vulnerability!

Fortunately, Private Information Retrieval (PIR) [2] addresses this issue. There are now myriad schemes proposing how clients can retrieve information from a database without disclosing which information is requested [2–5]. The academic literature has primarily optimized these systems by improving their theoretical properties [6–9], primarily to reduce communications overhead.

The biggest open problem related to PIR systems is how to make them practical. An academic panel titled “Achieving Practical Private Information Retrieval” lamented that the performance of existing PIR systems makes them unsuitable for practical use [10]. Recently, Sion suggested that many PIR techniques are so inefficient that it is faster to simply transmit all data stored on the server to the client [11]. More recently, Olumofin and Goldberg [12] have shown faster practicality results (especially with Chor PIR); however, these results are still much slower than non-PIR systems.

We demonstrate that it is possible to build a practical PIR system that provides performance similar to that of non-PIR production systems. Our system, upPIR uses the Chor multi-server PIR scheme [2], which uses XOR instructions that can be efficiently computed on modern hardware. By carefully choosing the block size to match the processor's cache size, upPIR's throughput is substantially faster than existing results. (This is opposed to prior work which has focused on reducing communication complexity.) upPIR allows clients to retain information-theoretic privacy while providing performance similar to popular HTTP and FTP servers.

2 Software Updaters

Software Updater Architecture. The architecture of software update systems (including upPIR) consists of three parties. The software vendor, such as Ubuntu or Microsoft, creates a set of updates and bundles them into a *release*. The vendor also creates some metadata that describes the release, called a *manifest*. The release is obtained and copied by a set of mirrors. For economic and configurability reasons, mirrors are an important and essential part of the software update landscape. Unfortunately, it is trivial for a malicious party to register as an official mirror and receive requests from clients, including requests for security updates [1].

Software Update Contents. The size and number of items stored by a mirror vary over software projects, ranging between .5GB and 4.3GB for recent versions of popular Linux distributions. The size of the security updates for a distribution is several orders of magnitude smaller than the full mirror data which contains normal updates. In this work we focus on distributing security updates and leave private distribution of complete software mirrors for future work. Further details about the suitability of PIR for software updates are provided in a tech report [13].

3 Threat Model

In our threat model, a client may contact many mirrors, including those that may be honest-but-curious. Our goal in this work is to prevent a mirror from knowing which software update is being retrieved by a vulnerable client. We assume that:

- The vendor is creating valid updates that the client wishes to retrieve.
- A non-malicious mirror may fail at any time.
- A malicious party may operate one or more mirrors. These mirrors may share or publicize any information they receive.
- An adversary may be able to observe all traffic sent over the network. This is consistent with a malicious access point or ISP.

4 Architectural Overview

Vendor. The vendor produces a set of updates that it wishes to package into a *release* and provide to clients. It also provides a list of mirrors to clients. The vendor generates a *manifest* that contains metadata about the updates provided in the release including secure hashes of the files. The release provided by a vendor conceptually breaks the updates into equally sized blocks. If this were not done, then performing an XOR of all updates together causes every XORed chunk of data to be the size of the largest update. This would effectively mask the size of the update being retrieved, but would be very inefficient if there is a wide distribution of update sizes. In our implementation, the vendor selects the block size when the manifest is created. (Section 5.3 discusses how to choose an efficient block size).

Mirror. An upPIR mirror obtains the files for the release from the vendor using `rsync` or another file transfer mechanism for distributing updates to mirrors. Following this, the mirror reads in all of the software updates in the release and stores them in one contiguous memory region. (The order of the software updates in memory is specified in the manifest file.) The mirror uses the manifest to validate each block. The mirror then notifies the vendor's server that it is ready to serve blocks to clients. The mirror provides the vendor with a public key to prevent a man-in-the-middle from viewing client requests. When a client sends a string of bits to the mirror, the mirror will XOR together all blocks with a 1 in their position of the client's request string. The mirror then sends the result back to the client (which is the size of one block). This response over an encrypted channel and is signed by the mirror's private key to provide non-repudiation. Note that the mirror can concurrently serve (non-private) FTP and HTTP requests.

Client. A client first contacts the vendor's server to obtain the latest manifest and mirror list. From the manifest, the client can determine which blocks of the release it needs to retrieve in order to receive its update. The client also has some value N that represents the number of mirrors that would have to collude to compromise the client's privacy. To retrieve a single block, the client generates $N - 1$ cryptographically suitable random strings. The client derives the Nth string by XORing the other $N - 1$ random strings together and flipping the bit of the desired update. Each string is sent to a different mirror over an encrypted channel (to prevent eavesdropping on the strings). Each mirror returns a block consisting of the specified blocks XORed together. The mirror signs the request and response in its reply to allow the client to demonstrate to a third party when a mirror is corrupt or malicious. If multiple blocks are desired, the procedure is repeated.

5 Evaluation

This section compares the impact of different block size choices. Our focus is on examining whether the theoretically optimal block size from a communications perspective [12] (where the block size is the square root of the database size) results in good throughput. We also examine the performance of upPIR on real data sets and in a realistic deployment using the block size we recommend. Since mirrors are often set up using outdated server hardware or in a VM on shared resources. The different machines used are as follows:

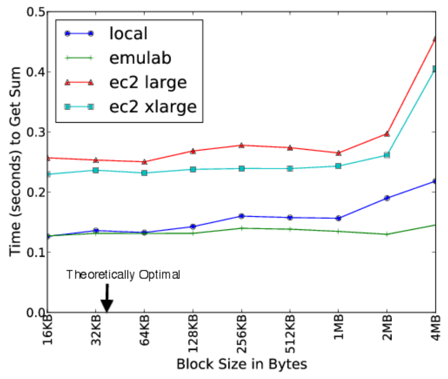

Fig. 1: Time to fetch one block against the Ubuntu release on multiple machines

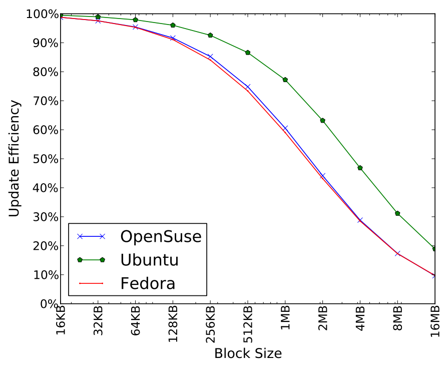

Fig. 2: Space efficiency of updates with various block sizes

- **ec2 large** and **ec2 xlarge** are Amazon EC2 instances [14].
- **emulab** is an three-year-old Emulab node with an Intel E5530 CPU with an 8MB L2 cache and 12GB of RAM on a virtual 100Mbps LAN [15, 16].
- **local** is an undergraduate student's three-year-old PC with an Intel E5506 CPU with a 4MB L2 cache and 6GB of RAM on a shared 100Mbps LAN.

5.1 Mirror XOR Microbenchmarks

Figure 1 demonstrates the time it takes to produce a block of data when a mirror serves the Ubuntu 10.04 data. We generated 10 random bit strings of the appropriate size and then measured the amount of time the mirror spent XORing the relevant update blocks together. Notice that the theoretically optimal block size from a communications standpoint [12] has essentially the same speed as block sizes up to 1MB. Both a 1MB and 2MB block size allow local and emulab to produce blocks quickly enough to saturate a T3 link. Once the block size increases to 2MB, the throughput no longer increases linearly with the block size due to data and code not fitting entirely in L2 cache. Producing a 1MB block in the same time as the theoretically optimal block size results in about a 50x increase in throughput.

The release size is an important factor in speed because larger releases contain more blocks. To explore the impact of the release size, we fixed the block size to be 1MB and then varied the release size on 'emulab' (not shown) The performance scales at the same rate as the release size until the release no longer fits in memory. Once the release exceeds the size of RAM, the performance drops by over an order of magnitude as disk latency comes into play (not shown). Our results show that upPIR scales linearly as the release size grows provided the data served fits within memory.

5.2 The Impact of Block Size on Efficiency

The previous discussion showed how quickly a mirror could produce XOR blocks. However, there is a difference between useful data and data. If a mirror can pro-

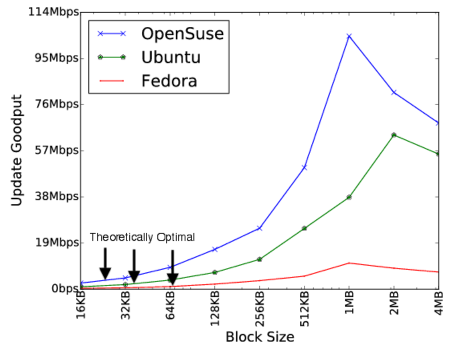

Fig. 3: Goodput for an average sized update in three releases.

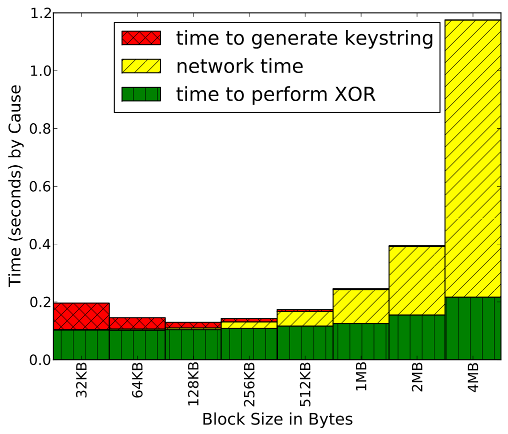

Fig. 4: Time to privately fetch one block.

duce a 1MB block in .1 second or a 2MB block in .15 second, from a throughput standpoint, the 2MB block size is superior. However, if the client wants a 1MB update but must retrieve 2MB of data to get it, then 1MB of the space is wasted. In essence, the data efficiency is the amount of retrieved data that is useful.

Figure 2 shows how changing the block size impacts efficiency for different data sets. The lines represent different update sizes (or data sets) and illustrate the performance difference when block size is varied. The values given were calculated by dividing the size of the release by the amount of data that a client would have to download to obtain every update in it one update at a time using our PIR scheme. This figure 2 demonstrates that the amount of useful data within a block decreases rapidly as the block size increases over 2MB. This is to be expected since larger blocks imply that there is more wasted space when retrieving an update. For example, between 70-85% of update data is unneeded when using 8MB blocks, but less than 5% is unneeded with 64KB blocks.

5.3 Choosing a Block Size to Optimize Goodput

One decision the vendor makes when creating the manifest for a release is to choose the block size. As we previously saw, this choice greatly impacts both the mirror XOR performance and the client's goodput. (Goodput is defined as the desired bytes per second, so ignores padding and packet headers.) In order to determine how to optimize the mirror's goodput, one can combine the mirror XOR time and the data efficiency to compute the goodput of the mirror.

Figure 3 shows how the goodput varies based on the throughput of the mirror and the space efficiency of the block size. This chart is generated by retrieving an average sized update from three distributions on the system 'local'. (Other systems show qualititatively similar results.) This graph shows that the goodput is optimal when block size is between 1MB to 2MB for each distribution. As a result, these block sizes seem to be the most efficient for this system. The theoretically optimal from a communications standpoint (the square root of the distribution size) has one to two orders of magnitude less throughput.

5.4 Controlled Macrobenchmarks

Figure 4 shows where time is spent retrieving a block from a mirror on emulab. First of all, the time to generate the cryptographically suitable random string is only paid on the client side of the connection. Similarly, the time that is spent XORing content is only performed by the mirror. The communication time is perceived by both systems. When the block size is small, the client's time to generate the string incurs a non-negligible cost. For larger block sizes, the network communication time is the dominant factor. For example, for a 4MB block size, the XOR takes about 200 ms, and the retrieval time is nearly 1 second. The theoretically optimal block size from a communications standpoint (about 38KB) has about 100 times slower throughput than a 2MB block.

5.5 Deployment

To understand the performance of upPIR in realistic environments, we deployed our software on machines around the world. We used our machine 'local' as an Ubuntu 10.04 vendor with a 2MB block size, three EC2 instances in either the US East, US West, or EU West availability zones as mirrors, and ran the client at the University of Washington. For the worldwide setting, we ran one mirror in each availability zone. We compared the time to download the 1.5MB `libc6-prof_2.12.1-0ubuntu6_i386.deb` package using upPIR, HTTP (apache), and FTP (vsftpd) on the same EC2 instances. For comparison's sake, we also downloaded the same file using FTP and HTTP from every available official Ubuntu mirror inside the United States.

Location	Protocol	Time
US-West (EC2)	HTTP	.64s
US-West (EC2)	FTP	1.5s
US-West (EC2)	upPIR	1.2s
US-East (EC2)	HTTP	1.5s
US-East (EC2)	FTP	2.1s
US-East (EC2)	upPIR	3.1s
EU-West (EC2)	HTTP	2.7s
EU-West (EC2)	FTP	4.5s
EU-West (EC2)	upPIR	4.1s
US Mirrors	HTTP	1.6s
US Mirrors	FTP	2.1s
Worldwide (EC2)	upPIR	3.5s

Table 1: Ubuntu update times.

Table 1 shows the result of distributing updates via upPIR and other mechanisms. The first thing to observe is that HTTP is slightly faster than FTP. We believe that this is because FTP uses more back and forth communication than HTTP (or upPIR) and therefore suffers the most from latency. HTTP is faster than upPIR, which is expected because the client is downloading 2MB of data from three mirrors instead of 1.5 MB from one mirror. Despite the additional information downloaded, upPIR's time is comparable to FTP on the same hardware. However, unlike HTTP and FTP, upPIR retrieves the update privately. Since our upPIR client downloads from three mirrors, even if two mirrors collude, they do not learn which update the upPIR client is retrieving.

6 Related Work

Impracticality results from researchers including Sion [11], Yoshida [17] and Sassaman [18] reveal inefficiencies in existing PIR schemes. Perhaps most interesting is Sion's argument that many types of computational PIR are presently impractical and, given hardware trends, unlikely to improve from a performance perspective [11]. He argues that it is faster to transfer the entire database than to perform PIR with a large class of proposed schemes.

Olumofin and Goldberg [12] recently provided performance results for a PIR system that does not use the primitives mentioned as impractical in Sion's prior work. Olumofin's resulting system is shown to be one to three orders of magnitude more efficient than transferring the entire database. They use the theoretically optimal block size in their analysis, so their reported performance results are significantly slower. For example, for a 2GB database, they produce a 46KB block in just over 1 second (roughly 370Kbps). upPIR produces a 1MB block in .2 seconds from a 2GB data store, showing throughput of roughly 40Mbps: two orders of magnitude higher throughput. We contacted the authors and discovered that their Chor implementation is similar to ours in performance when given the non-theoretically optimal block size.

Similarly, Melchor [19] provides a fast PIR implementation that uses lattices instead of the XOR-based primitives in our work. The authors mention they aimed to maximize throughput by choosing experimental data that fit exactly within cache (instead of using realistic data sets). As a result, they retrieved 3MB results from a 36MB database to achieve their 230Mbps speed number. On our system with comparable hardware ('local') [20], our implementation can produce results for a 36MB database at over 1Gbps. This demonstrates that careful block size choice results in far greater throughput improvements.

Another common way to try to speed up PIR is to use specialized hardware. Proposals have suggested leveraging GPUs [19], secure co-processors [21] or oblivious RAM [22]. These results show promise, but our work demonstrates that it is possible to achieve excellent performance simply with universally deployed hardware (commodity CPUs).

7 Conclusion

This work demonstrates that in PIR systems, the theoretically optimal block size (for minimizing communications cost) can be far less efficient than larger block sizes in practice. In fact, it is possible to construct a PIR system with performance similar to production non-PIR systems. We chose to motivate and test upPIR by privately distributing security updates on commodity hardware and show this has performance similar to FTP. Our source code is available at `https://uppir.poly.edu`.

References

1. Cappos, J., Samuel, J., Baker, S., Hartman, J.: A Look in the Mirror: Attacks on Package Managers. In: CCS 2008, New York, NY, USA, ACM (2008) 565–574
2. Benny Chor and Oded Goldreich and Eyal Kushilevitz and Madhu Sudan: Private Information Retrieval. Journal of the ACM **45** (1998) 965–982
3. Ding, X., Yang, Y., Deng, R., Wang, S.: A new hardware-assisted PIR with O(n) shuffle cost. International Journal of Information Security **9** (2010) 237–252 10.1007/s10207-010-0105-2.

4. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: STOC. (1997) 294–303
5. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: single database, computationally-private information retrieval. In: FOCS 1997. (oct 1997) 364 –373
6. Beimel, A., Ishai, Y., Kushilevitz, E., franois Raymond, J.: Breaking the O(n 1/(2k1)) Barrier for Information-Theoretic Private Information Retrieval. In: FOCS 2002. (2002) 261–270
7. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: EUROCRYPT'99, Berlin, Heidelberg, Springer-Verlag (1999) 402–414
8. Asonov, D., Freytag, J.C.: Almost Optimal Private Information Retrieval. In: PETS 2002, Springer (2002) 209–223
9. Ambainis, A.: Upper bound on communication complexity of private information retrieval. In: ICALP '97, London, UK, Springer-Verlag (1997) 401–407
10. : Achieving Practical Private Information Retrieval (Panel @ Securecomm 2006) `http://www.cs.sunysb.edu/~sion/research/PIR.Panel.Securecomm.2006/`.
11. Sion, R.: On the Computational Practicality of Private Information Retrieval. In: NDSS '07. (2007)
12. Olumofin, F.G., Goldberg, I.: Revisiting the computational practicality of private information retrieval. In: Financial Cryptography. (2011) 158–172
13. Cappos, J.: Avoiding Theoretical Optimality to Efficiently and Privately Retrieve Security Updates (full version). Technical Report TR–CSE–2013–01, Department of Computer Science and Engineering, NYU Poly (February 2013)
14. : AWS Instance Types `http://aws.amazon.com/ec2/#instance`.
15. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. In: Proc. 5th OSDI, Boston, MA (Dec 2002) 255–270
16. : Emulab d710 Node Type Information `https://www.emulab.net/shownodetype.php3?node_type=d710`.
17. Yoshida, R., Cui, Y., Shigetomi, R., Imai, H.: The practicality of the keyword search using PIR. In: ISITA 2008. (dec. 2008) 1 –6
18. Sassaman, L., Preneel, B., Esat-cosic, K.U.L.: The Byzantine Postman Problem: A Trivial Attack Against PIR-based Nym Servers. Technical report, ESAT-COSIC 2007-001 (2007)
19. Melchor, C., Crespin, B., Gaborit, P., Jolivet, V., Rousseau, P.: High-Speed Private Information Retrieval Computation on GPU. In: SECURWARE '08. (aug. 2008) 263 –272
20. : Compare of Intel E5506 to E5345 `http://ark.intel.com/Compare.aspx?ids=37096,28032`.
21. Khoshgozaran, A., Shirani-Mehr, H., Shahabi, C.: SPIRAL: A Scalable Private Information Retrieval Approach to Location Privacy. MDMW 2008
22. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: CCS '08, New York, NY, USA, ACM (2008) 139–148