Avoiding Theoretical Optimality to Efficiently and Privately Retrieve Security Updates

Justin Cappos

Department of Computer Science and Engineering

Technical Report
TR-CSE-2013-01
2/06/2013

Avoiding Theoretical Optimality to Efficiently and Privately Retrieve Security Updates

Justin Cappos

TR–CSE–2013–01 (Extended version of FC 2013 paper)
Computer Science and Engineering
NYU Poly
jcappos@poly.edu

Abstract. By requesting a security update, a client also notifies potential attackers that it is vulnerable to attack. Fortunately, this problem can be solved using Private Information Retrieval (PIR), a problem which has been widely studied by the security community. Unfortunately, due to performance reasons PIR solutions have been dismissed as impractical by academia and have not been adopted by industry.
This work demonstrate 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 a factor of roughly 50x. We constructed an efficient 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

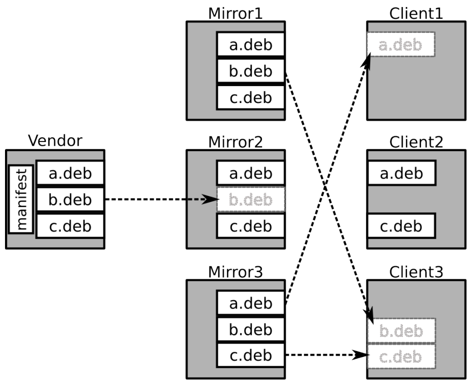

Fig. 1: Architecture of a typical software updater

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.

While security updates were chosen to motivate this work, PIR systems are applicable to a rich set of problems. This includes stock quotes [13], pharmaceutical databases [14], location tracking [15], census information [13], and email [16, 17]. A high-performance PIR system will have wide reaching privacy benefits across a large number of fields.

Distribution	Version	Size	Updates
OpenSUSE	11.4	556MB	679
Ubuntu	10.04	1.4GB	845
Fedora	14	4.3GB	6305

Fig. 2: Security update information for popular Linux distributions

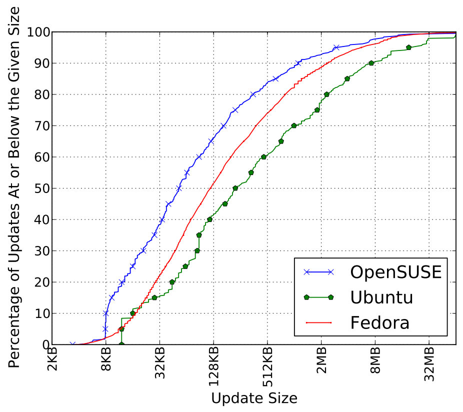

Fig. 3: Figure showing the number of updates at or below the given size (by project)

2 Software Updaters

2.1 Software Updater Architecture

The architecture of software update systems (including upPIR) is similar to what is shown in Figure 1. The software vendor, such as Ubuntu or Microsoft, creates a set of updates and bundles them into a *release*. In this example, the set of updates contains the packages `a.deb`, `b.deb`, and `c.deb`. 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].

2.2 Software Update Contents

The size and number of items stored by a mirror vary over software projects, as illustrated by Figure 2 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 the appendix.

3 Threat Model

Given information about the usage environments for software update systems, we can devise a realistic threat model. In particular, a software update system may contact many mirrors, including those that may be malicious. 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. Therefore the adversary may see all communications and decode any encrypted messages for their mirrors. Furthermore, 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.
- A malicious mirror may corrupt or modify content.

For the bulk of the paper, we focus on allowing rapid retrieval of updates given the first four constraints. We discuss an extension to handle mirrors that corrupt or maliciously modify content in Section 4.3.

4 Architectural Overview

The overall architecture of our system (upPIR) is divided into the same three components of traditional software updaters; a vendor, mirrors, and clients. These parties use Chor PIR to allow the client to privately retrieve updates from the mirror and vendor (the intuition for this is described in an appendix). We specifically highlight differences between traditional update systems and upPIR.

4.1 Vendor

The vendor produces a set of updates that it wishes to package into a *release* and provide to clients. The vendor generates a *manifest* that contains metadata about the updates provided in the release. 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 5.4 discusses how to choose an efficient block size).

The manifest contains the secure hashes of each of the blocks within the release. A client can use the manifest to determine which blocks to retrieve from the mirrors and to validate their correctness. The client can then privately retrieve those blocks and reassemble its update. To protect against timeliness attacks and similar threats, upPIR leverages best practices for software update security [18, 19].

The vendor's server is also responsible for providing the manifest and a current list of mirrors to interested clients. The vendor's server polls the mirrors for liveness and removes unresponsive mirrors. It also removes any mirror that has been demonstrated to be malicious by a client.

4.2 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. 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).

In order to prevent a man-in-the-middle from snooping on the client's request strings, communications between a client and mirror are encrypted. To bootstrap this process, the mirror provides the vendor with a public key when it registers to distribute updates. This is used both to establish a session key for the communication with the client and also to attest to the correctness of the block returned to the client. This signature is useful for non-repudiation because it allows the client to prove that the mirror is producing corrupt blocks.

Note that our mirror implementation may also simultaneously serve content to legacy clients. The mirror can look up the location of the update in the release by checking the manifest. The security update can then be returned in response to the client's HTTP or FTP request.

4.3 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 describes our evaluation of upPIR. In particular, we compare the impact of different block size choices focusing 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.

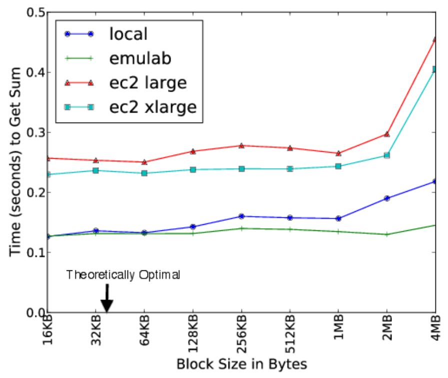

Fig. 4: Time to fetch one block against the Ubuntu release on multiple machines

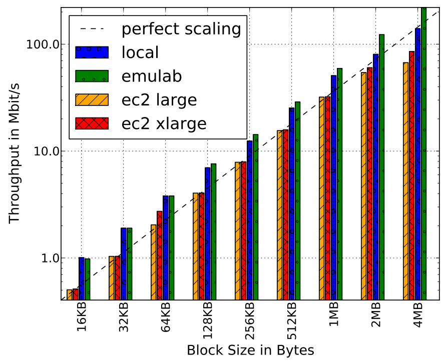

Fig. 5: Throughput in fetching one block against a 1GB release on multiple machines.

5.1 Implementation and Experimental Setup

The client and vendor code for upPIR are written in entirely in Python. The mirror code is a mix of Python and 64-bit C code. The XOR portion of the algorithm is in C for speed reasons, but the remaining code is all in Python for maintainability. According to `sloccount`, the XOR portion of upPIR has 297 lines of C code. The client, mirror, and vendor implementations combined are 499 lines of Python code.

We wanted to test upPIR with realistic hardware for software mirrors. In practice mirrors are often set up using outdated server hardware or in a VM on shared resources. As a result, we chose a range of servers that had these properties. The different machines used are as follows:

- **ec2 large** is a "large" 64 bit Amazon EC2 instance [20], with 7.5 GB of RAM and 2 ECUs. One ECU is roughly equivalent to a 1.0Ghz Intel Xeon circa 2007. Other specifications such as L2 cache size and network bandwidth are not disclosed by Amazon.
- **ec2 xlarge** is an "extra-large" 64 bit Amazon EC2 instance with 15GB of RAM and 4 ECUs.
- **emulab** is an approximately two-year-old Emulab node with an Intel E5530 CPU with an 8MB L2 cache and 12GB of RAM on a virtual 100Mbps LAN [21, 22].
- **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.

5.2 Mirror XOR Microbenchmarks

In this section we explore the performance with different block sizes and release sizes. One potential bottleneck in upPIR is the rate at which the mirror can XOR data. Recall that a mirror may have a release of 1GB or more and need to XOR sizeable blocks of this data together. We implemented the XOR loop of our mirror to XOR 64-bit chunks of data at a time. Our performance is bottlenecked mostly on the time it takes to read the chunks from memory.

Figure 4 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 size of the block has little impact until the block size exceeds 2MB. If the necessary code, the block that is being XORed, and the current result of XORing all fit in cache, the performance is similar.

Notice that the theoretically optimal block size from a communications standpoint [12] has essentially the same speed as larger block sizes. Producing a 1MB or larger block in the same time as the theoretically optimal block size results in about a 50x increase in throughput.

Figure 4 also indicates that the outdated-but-dedicated hardware performs slightly better than the shared EC2 instances. We believe this may be due to two factors. First, our EC2 instances may be co-located with other code which causes a higher degree of L2 cache misses. Second, while Amazon is vague about the exact specifications one can expect, the EC2 instances' expected processing power is lower than that of our dedicated machines.

Another way of visualizing the data in Figure 4 is to look at the resulting throughput, as is seen in Figure 5. This graph shows that as the block size increases, the throughput improves. However, once the block size increases to 2MB, the throughput no longer increases linearly with the block size. This is due to data and code not fitting entirely in L2 cache.

The block size is not the only thing that may vary depending on how upPIR is used. The size of the release also varies significantly between vendors. The release size is an important factor 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' as shown in Figure 6. 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.

5.3 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 produce 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.

One of the main factors in calculating the efficiency is the block size. Figure 7 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

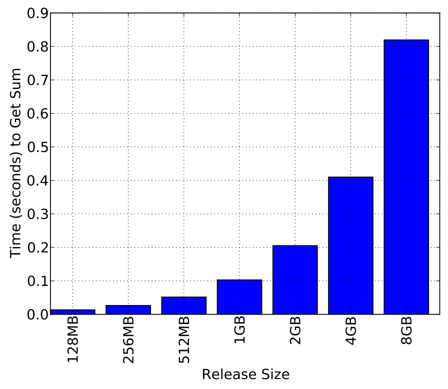

Fig. 6: Impact of the release size on performance

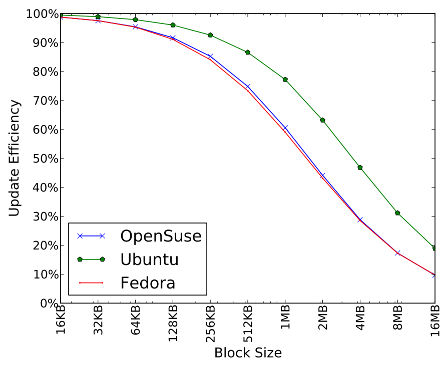

Fig. 7: Space efficiency of updates with various block sizes

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. Note that this graph assumes that all update data in a block other than the requested update is not of interest to the client. This is not always the case, as a client may request two updates that share a block. As such, our results represent the worst case efficiency for updates from these distributions.

Figure 7 demonstrates that the amount of useful data within a block decreases rapidly as the block size increases. 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. For a block size of 1MB, the amount of unneeded data is about 20-40%. There is a significant increase in the amount of unneeded update data for block sizes greater than 2MB.

5.4 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 8 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'. 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.

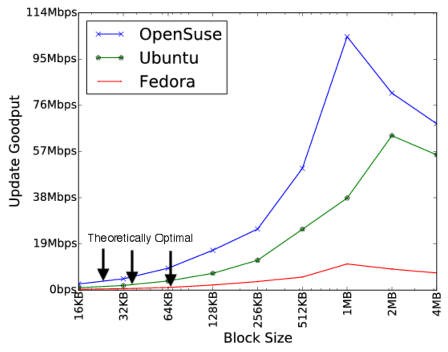

Fig. 8: Goodput for an average sized update in three releases.

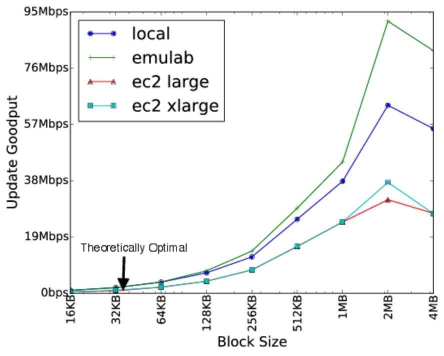

Fig. 9: Goodput for an average Ubuntu update.

In addition to the differences seen by releases, the characteristics of the machine also impacts the performance characteristics. Figure 9 shows how the goodput varies for Ubuntu 10.04 across platforms. Note that the EC2 instances have nearly identical performance for most block sizes. Since the average update size for Ubuntu is more than 1MB, it is unsurprising that the peak for each platform is at a 2MB block size. It is worth noting that the relative improvement between 1MB and 2MB is much lower for systems that have smaller L2 caches and thus process 2MB block sizes more slowly. Once again, the theoretically optimal block size is more than an order of magnitude slower than 2MB on all systems.

5.5 Controlled Macrobenchmarks

All of the benchmarks we have observed so far have focused purely on the mirror's XOR speed. However, other important factors influence the efficiency of retrieving an update, including the time the client takes to generate a suitable random bit string and the transmission time for the resulting block. We instantiated a client and an Ubuntu 10.04 mirror on Emulab machines and used them to evaluate upPIR's overall performance aspects. Since the client is only communicating with one mirror, this experiment does not capture the time the client needs to XOR the results together. However, further testing (not shown) has indicated this cost is insignificant. This test was performed with 100Mbps maximum virtual links between nodes with LAN latencies.

Figure 10 shows where time is spent retrieving a block from a mirror. 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

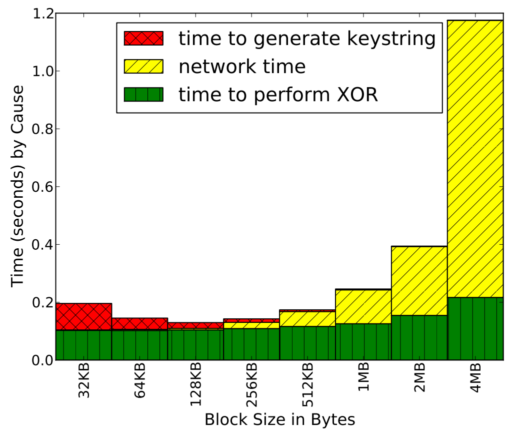

Fig. 10: Time to privately fetch one block.

Location	Protocol	Time (s)
US-West (EC2)	HTTP	0.64
US-West (EC2)	FTP	1.5
US-West (EC2)	upPIR	1.2
US-East (EC2)	HTTP	1.5
US-East (EC2)	FTP	2.1
US-East (EC2)	upPIR	3.1
EU-West (EC2)	HTTP	2.7
EU-West (EC2)	FTP	4.5
EU-West (EC2)	upPIR	4.1
Official US Mirrors	HTTP	1.6
Official US Mirrors	FTP	2.1
Worldwide (EC2)	upPIR	3.5

Fig. 11: The time taken when retrieving an Ubuntu security update by different mechanisms.

block size from a communications standpoint (about 38KB) has about 100 times slower throughput than a 2MB block.

A goal of mirror software is to serve a reasonable number of clients per second. Our mirror implementation is naïve with respect to parallelism and largely operates on clients sequentially. By parallelizing network I/O with XORing blocks, our mirror can handle between 5 and 8 requests per second. As a point of comparison, in April 2011 the 99.999th percentile request rate on our official Ubuntu mirror for versions 6.06 to 11.10 was 3 requests per second. We believe upPIR's throughput is more than sufficient for practical scenarios, especially given that a T3 link has a theoretical maximum throughput of about 3.6 average sized Ubuntu updates per second.

5.6 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 to show the expected performance for a client choosing mirrors at random. We compared the time to download the 1.5MB `libc6-prof_2.12.1-0ubuntu6_i386.deb` package using different mechanisms. We chose this package because it is popular and close to the average update size. We compared upPIR running in this configuration with hosting the files via 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. (We list the median time for the Ubuntu mirrors because a few slow nodes skew the average.)

Figure 11 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 we configured our upPIR client to download from three mirrors, even if two mirrors collude, they do not learn which update the upPIR client is retrieving.

6 Related Work

6.1 PIR History and Impracticality Results

In 1995, Chor et al. proposed PIR as a novel mechanism for allowing clients to retrieve information from a database without disclosing to the server what was being retrieved [2]. This generated significant academic interest, particularly on two perceived weaknesses of the basic scheme: its need for multiple non-communicating servers (referred to as the replication problem) and its communication complexity, which some termed its efficiency. In 1997, Kushilevitz and Ostrovsky solved both problems by moving from the information- theoretic model to a model that admits computationally bounded adversaries. This work proved that one could obtain in sublinear asymptotic complexity provable privacy using only a single server. This spurred exploration of their CPIR (Computationally Private Information Retrieval) problem [23–25, 5, 7]. Much of this research effort focused on reducing the communication complexity at the expense of computational cost.

However, PIR implementations were few and far in between and known to be difficult to apply to real problems [16]. In 2006, a panel entitled "Achieving Practical Private Information Retrieval" [10] lamented the impracticality of much of PIR research and discussed suggestions on how this might be improved. Different panelists threw out a wide variety of proposals (some of which we leverage in this work), including focusing on returning blocks instead of bits and using multiple instead of single servers. There were also extended discussions about specialized hardware and computing environments which have inspired other research.

Impracticality results from researchers including Sion [11], Yoshida [26] and Sassaman [27] 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.

6.2 PIR on Commodity Hardware

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). As was shown in Figure 6, 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 their Chor implementation is similar to ours in performance when choosing the non-theoretically optimal block size.

Similarly, Melchor [28] provides a fast PIR implementation that uses lattices instead of the XOR-based primitives in our work. Their implementation boasts speeds of 2Gbps on GPUs or 230 Mbps on CPUs similar to ours. However, as was pointed out by Olumofin et al [12], these results are more of a micro-benchmark as they do not account for many costs that would be incurred in actual system use. 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') [29], 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.

6.3 PIR on Specialized Hardware

Another common way to try to speed up PIR is to use specialized hardware. Proposals have suggested leveraging GPUs [28], secure co-processors [30] or oblivious RAM [31]. These results show promise, but our work demonstrates that it is possible to achieve excellent performance simply with universally deployed hardware (commodity CPUs).

6.4 Applying PIR

Previous attempts to use PIR on practical problems on commodity hardware have included stock quotes and census information [13], pharmaceutical databases [14], location tracking [15], and email [16, 17]. Unfortunately, many of these systems do not provide performance results [16, 14]. Yang et al [13] do provide performance results for their study of stock quotes and census information; however, it is hard to compare results which were run on systems from 2002. Notably, the database they compare with is 86kb, and so on their hardware, it fits easily within the L2 cache. Their results indicate that they can produce between 2 to 5 8kb blocks per second.

Ghinita et al [15] perform private queries specifically tailored to the problems encountered by location services. Their system does not process traditional PIR queries but instead provides answers to problems like nearest neighbor queries. When running on modern hardware, their system can answer queries from 128KB to 1.2MB of location data in 1 second or more per query. It is unclear whether the reduced rate in processing queries poses a barrier to adoption by location service providers.

Wang et al. [32] propose a concept called Bounding Box PIR that allows the client to specify more relaxed privacy constraints and a budget for server side computation. They evaluate their system using around 3MB of records from a voter database [33]. Their implementation returns records of about 100 bytes from this database in 400ms to 600ms (depending on the privacy and computation budget settings) on hardware similar to what we use. These results are many orders of magnitude slower than a conventional database would be on similar hardware.

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 work on upPIR does not represent the completion of a goal, but the beginning of an exploration into practically deployable PIR systems. We are currently working towards a production deployment of upPIR on software mirrors to see what issues arise in practice. We make our source code publicly available with a MIT License (`https://uppir.poly.edu`).

References

1. Cappos, J., Samuel, J., Baker, S., Hartman, J.: A Look in the Mirror: Attacks on Package Managers. In: Proc. 15th ACM Conference on Computer and Communications Security, 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: Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on. (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: In Proc. of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS). (2002) 261–270
7. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: Proceedings of the 17th international conference on Theory and application of cryptographic techniques. EUROCRYPT'99, Berlin, Heidelberg, Springer-Verlag (1999) 402–414
8. Asonov, D., Freytag, J.C.: Almost Optimal Private Information Retrieval. In: In 2nd Workshop on Privacy Enhancing Technologies (PET2002), Springer (2002) 209–223

9. Ambainis, A.: Upper bound on communication complexity of private information retrieval. In: Proceedings of the 24th International Colloquium on Automata, Languages and Programming. 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: In Proceedings of the Network and Distributed Systems Security Symposium (NDSS). (2007)
12. Olumofin, F.G., Goldberg, I.: Revisiting the computational practicality of private information retrieval. In: Financial Cryptography. (2011) 158–172
13. Yang, E., Xu, J., Bennett, K.: A fault-tolerant approach to secure information retrieval. In: Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on. (2002) 12 – 21
14. Asonov, D.: Private Information Retrieval - An Overview And Current Trends. In: GI Jahrestagung (2). (2001) 889–894
15. Ghinita, G., Kalnis, P., Khoshgozaran, A., Shahabi, C., Tan, K.L.: Private queries in location based services: anonymizers are not necessary. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. SIGMOD '08, New York, NY, USA, ACM (2008) 121–132
16. Sassaman, L., Cohen, B.: The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval. In: In Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2005). (2005) 1–9
17. Sassaman, L., Cohen, B., Mathewson, N.: The Pynchon Gate: A Private Information Retrieval-based Pseudonym Service (2011)
18. Bellissimo, A., Burgess, J., Fu, K.: Secure Software Updates: Disappointments and New Challenges. In: 1st USENIX Workshop on Hot Topics in Security, Vancouver, Canada (Jul 2006) 37–43
19. Samuel, J., Cappos, J.: Package Managers Still Vulnerable: How To Protect Your Systems. ;login: Magazine (Feb 2009)
20. : AWS Instance Types http://aws.amazon.com/ec2/#instance.
21. 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. of the Fifth Symposium on Operating Systems Design and Implementation, Boston, MA, USENIX Association (December 2002) 255–270
22. : Emulab d710 Node Type Information https://www.emulab.net/shownodetype.php3?node_type=d710.
23. Blundo, C., D'Arco, P., Santis, A.D.: A t-Private k-Database Private Information Retrieval Scheme (2002)
24. Narayanam, K.S.: A Novel Scheme for Single Database Symmetric Private Information Retrieval. In: Financial Cryptography. (2006)
25. Mishra, S.K., Sarkar, P.: Symmetrically private information retrieval. In: INDOCRYPT. (2000) 225–236
26. Yoshida, R., Cui, Y., Shigetomi, R., Imai, H.: The practicality of the keyword search using PIR. In: Information Theory and Its Applications, 2008. ISITA 2008. International Symposium on. (dec. 2008) 1 –6
27. 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)

28. Melchor, C., Crespin, B., Gaborit, P., Jolivet, V., Rousseau, P.: High-Speed Private Information Retrieval Computation on GPU. In: Emerging Security Information, Systems and Technologies, 2008. SECURWARE '08. Second International Conference on. (aug. 2008) 263 –272
29. : Compare of Intel E5506 to E5345 `http://ark.intel.com/Compare.aspx?ids=37096,28032`.
30. Khoshgozaran, A., Shirani-Mehr, H., Shahabi, C.: SPIRAL: A Scalable Private Information Retrieval Approach to Location Privacy. Mobile Data Management Workshops, 2008 Ninth International Conference on **0** (2008) 55–62
31. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: Proceedings of the 15th ACM conference on Computer and Communications Security. CCS '08, New York, NY, USA, ACM (2008) 139–148
32. Wang, S., Agrawal, D., El Abbadi, A.: Generalizing PIR for Practical Private Retrieval of Public Data. In: Proceedings of the 24th annual IFIP WG 11.3 working conference on Data and applications security and privacy. DBSec'10, Berlin, Heidelberg, Springer-Verlag (2010) 1–16
33. : Adult Data Set `http://rss.acs.unt.edu/Rdoc/library/arules/html/Adult.html`.
34. : Windows Update `http://windows.microsoft.com/en-US/windows/help/windows-update`.
35. Gkantsidis, C., Karagiannis, T., VojnoviC, M.: Planet scale software updates. In: SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications, New York, NY, USA, ACM (2006) 423–434
36. : Ubuntu Popularity Contest `http://popcon.ubuntu.com/`.
37. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In: Security and Privacy, 2008. SP 2008. IEEE Symposium on. (May 2008) 143 –157
38. Dingledine, R., Mathewson, N., Syverson, P.: Tor: the second-generation onion router. In: Proceedings of the 13th conference on USENIX Security Symposium - Volume 13. SSYM'04, Berkeley, CA, USA, USENIX Association (2004) 21–21

A Software Update PIR Suitability

Do all clients retrieve the same update? An important question is whether other factors indicate the exact update that is being retrieved. For the most part, Microsoft directly serves updates for their software, with each client downloading essentially the same update [34, 35]. This means that private retrieval of updates is almost useless for Windows machines. Linux systems are much more diverse. To understand this, we examined mirror requests from 120K unique IPs that contacted offical mirrors for CentOS, Debian, Fedora, OpenSuSE, and Ubuntu. Clients issued a total of 260K unique requests (omitting retries for the same package). We then counted the number of IP addresses that requested the same set of packages from our mirrors. The most popular set of packages only was requested 692 times which is less than .3% of the time! In contrast, 24K clients (about 20%) requested a completely unique set of packages. More than 50% of the clients downloaded a set of packages downloaded by 14 or fewer other IPs. Part of the apparent diversity here stems from the fact that some package managers distribute requests across multiple mirrors (as we describe later).

Does the size indicate the update? Most software updates have a unique size. We propose padding to mask the size of the updates that are being downloaded. Figure 3 demonstrates the distribution of update sizes within some popular versions of Linux. 95-99% of all updates would be included in the first 8 size denominations. This decreases the utility of the package size as an indicator for which update is being retrieved. The efficiency impact of using padding is explored in more detail in Section 5.4.

One additional influence on the size that complicates the above analysis is the fact that packages may depend on each other. Over 260K client requests, clients downloaded 2.2 packages on average. Part of the reason is that some package managers will request packages from different mirrors when performing a download. This helps to mitigate the mirrors' ability to use size as a differentiator.

Does the time indicate the update? To understand the correlation between the update requested and the time of the request, we collected data provided by the `popularity contest` package [36] on Ubuntu. This utility assembles installation and update information for all package types from Ubuntu users and reports aggregate information about the entire Ubuntu community. When looking at the two weeks of user-reported updates on Feb 3rd, 2011, there were about 35 million updates for over 35 thousand distinct packages. Over this time period, the most popular package update represented only 0.4% of the total updates performed. Also, the top 100 packages only comprised about 23% of the updates.

Is the IP address enough? An attacker still receives a client's IP and could try to attack all possible updates for that distribution that have a known vulnerability. However, most Linux distributions have hundreds of security updates available. An attacker who could previously pinpoint a single vulnerability and attack only when contacted by a vulnerable client must now attempt to blindly exploit clients. This greatly increases the probability of such an attacker tripping an IDS or otherwise being discovered.

Isn't the opportunity for attack short? A natural question is whether this threat has significant impact given that a client will download and apply a patch soon after issuing the request. Given the ability for an attacker to automatically generate exploits from a patch [37], this is a significant threat. More importantly, an attacker who controls a mirror can increase the attack window by accepting the TCP connection from the package manager but returning data at an arbitrarily slow rate. With a PIR system, the attacker must blindly attack vulnerabilities, greatly increasing the chance of discovery.

Does the combination of information reveal the update? We have shown that while the IP address, size, and time provide some information about what security update is retrieved, this information is rather limited. Even when combined it does not disclose the update in the majority of situations. However, despite the relatively short time period that many updates occur in, there is still a significant risk. In practice today, without PIR every client discloses all vulner-

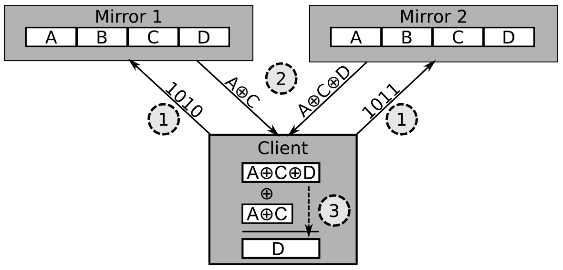

Fig. 12: Illustration of a PIR client privately requesting block D

abilities to untrusted parties. As such security updates seem to be an appropriate domain to apply PIR techniques.

Do existing techniques solve this problem? Existing software solves some of the problems. HTTPS uses encryption to protect communication between the client and mirror and can stop an eavesdropper from detecting which update is retrieved — a upPIR uses. However, HTTPS only encrypts the communication between the mirror and client, so the mirror learns exactly what security update the client is requesting.

One could also use a mixnet like Tor [38] to provide privacy when retrieving updates. Unfortunately, it is blocked by many governments and organizations around the world. This would prevent users in those areas from anonymously obtaining security updates. We believe that an upPIR mirror serving security updates is much less likely to be blocked.

B Private Information Retrieval Intuition

In this section, we describe the intuition behind the fundamental building block in our private information retrieval scheme. Our PIR scheme is the basic linear-summation scheme for multi-server PIR proposed by Chor et al. [2] and dismissed as theoretically sub-optimal later in the same paper. The intuition behind why this scheme privately retrieves information follows.

Suppose a client wants to privately retrieve an update from two mirrors so that neither mirror knows which update the client is retrieving (as is shown in Figure 12). Let's suppose that a vendor has produced a release containing 4 updates of the same size called A, B, C, and D. This release is currently being served by the two mirrors. The client will generate a random string of bits of length 4 (one bit for each update in the release) and send this to the first mirror. The mirror receives the random string and then generates a response consisting of the bitwise XOR of each update where there is a 1 value in that position. For example, the string 1010 would contain A XOR C. Then the client takes the random string it sent to the first mirror and flips the bit of the update it wants. Let's suppose the client wanted update D, if the first string was 1010, it would send 1011 to the second mirror. The second mirror sends back a response that contains all of the updates with a 1 XORed together (or A XOR C XOR D in our example). The client can then XOR the responses together to obtain the update D.

In our simple example, we addressed a client that retrieves an update from two mirrors. However, this only provides protection if those two mirrors do not collude. If a client wishes to protect against $N-1$ colluding mirrors, the client can generate $N-1$ random bit strings to send to the first $N-1$ mirrors. The Nth mirror should be sent the first $N-1$ strings XORed together with the bit for the desired update flipped. The client can XOR all N results together to obtain the result. If any $N-1$ of the N mirrors collude, they will not know what information the client is retrieving. Consequently, all N mirrors would need to collude in order to determine the requested update.

Each mirror receives a random string of bits (or a random string with an unknown bit flipped) and therefore gains no information about which update is being retrieved. Only by sharing the bit strings could the mirrors discover which update the client is retrieving. As long as the client contacts enough mirrors such that one of them does not share this information, no information is leaked regarding which security updates are being requested.

However, a man-in-the-middle placed near the client, such as their ISP or access point, could simply read all of the client's bit strings and easily decode which update is desired. To protect against an adversary who can observe all network traffic, the client can communicate with each mirror over an encrypted channel.