The Dissertation Committee for Sebastian Gomez Angel
certifies that this is the approved version of the following dissertation:

Unobservable communication over untrusted infrastructure

Committee:

Emmett Witchel, Supervisor

Simon Peter

Michael Walfish

Brent Waters

Nickolai Zeldovich

Unobservable communication over untrusted infrastructure

by

Sebastian Gomez Angel

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2018

Acknowledgments

I am thankful to have had the opportunity to work with my adviser Michael Walfish. Mike's relentless dedication and his attention to detail has improved the content and presentation of this dissertation and of all of my other work. While I will not deny that writing camera-ready versions of papers (that had already been accepted for publication) nearly from scratch seemed extreme at first, I have come to appreciate his philosophy of taking all feedback to heart and prioritizing the needs of readers. I also appreciate the freedom that Mike has given me to pursue any project or idea that I find interesting. Mike's willingness to put my needs and interests ahead of his own is truly inspiring and something that I hope to replicate.

I am also thankful to Lorenzo Alvisi and Emmett Witchel for serving as my official supervisors at various times once Mike departed from UT, and for offering me advice and encouragement during my job search. My committee members Mike, Emmett, Simon Peter, Brent Waters, and Nickolai Zeldovich gave me great feedback on my proposal, defense, and several drafts of this dissertation. Brent's comments during my thesis proposal inspired the content of Chapter 4.6.

I am indebted to Srinath Setty, with whom I co-authored the work presented in this dissertation, and who has served as a fantastic mentor. Without his support and his 24/7 availability, this dissertation would not exist. My other co-authors, Hao Chen and Kim Laine, made critical contributions to the design and implementation

of SealPIR (Chapter 5) and PBCs (Chapter 6), and I am grateful for their help. I thank David Lazar, Jing Leng, Ioanna Tzialla, and Minjie Wang for helping me improve the content and presentation of Chapter 4.6.

Josh Leners came up with the name Pung (which is the ROT13 encoding of the word "chat"), vetted all of my ideas with a healthy amount of skepticism, and taught me LaTeX and gnuplot hacks that were incredibly useful when writing this dissertation. Trinabh Gupta introduced me to PIR, which ended up being a key component of this dissertation. Riad Wahby taught me how to abuse machines for fun and profit, and was always an endless source of hilarious ideas. Riad also taught me the value of meticulously brewing every cup of coffee; I have attempted to write this dissertation with that same devotion.

I thank Joe Bonneau, Jinyang Li, Dennis Shasha, and Lakshmi Subramanian for making me feel at home, and for giving me crucial help and advice during the job search process. To all of my friends at UT and NYU, Talal Ahmad, Varun Chandrasekaran, Natacha Crooks, Chien-chin Huang, Tyler Hunt, Shiva Iyer, Manos Kapritsos, Youngjin Kwon, Michael Lee, Shuai Mu, Chunzhi Su, Yan Shvartzshnaider, Cheng Tan, Zhaoguo Wang, Ed Wong, and Lingfan Yu, thank you for always being there for me, and for listening to my terrible jokes without too much judgement. Finally, I thank my family for their unconditional support.

Sebastian Gomez Angel

The University of Texas at Austin
August 2018

Unobservable communication over untrusted infrastructure

Publication No. _____

Sebastian Gomez Angel, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Emmett Witchel

In the past decade there has been a significant increase in the collection of personal information and communication metadata (with whom users communicate, when, how often) by governments, Internet providers, companies, and universities. While there are many ongoing efforts to secure users' communications, namely end-to-end encryption messaging apps and email services, safeguarding metadata remains elusive. This dissertation discusses the design, implementation, and evaluation of a system called Pung that makes progress on this front. Pung lets users exchange messages over the Internet without revealing any information in the process. Perhaps surprisingly, Pung achieves this strong privacy property even when all providers (ISPs, com-

panies, servers, etc.) are arbitrarily malicious.

As part of realizing Pung, this dissertation introduces two orthogonal but complementary techniques: *SealPIR* and *probabilistic batch codes* (PBCs). SealPIR is a new private information retrieval (PIR) library that reduces the communication costs of the most computationally efficient PIR protocol by over two orders of magnitude. SealPIR can also be used in other contexts to instantiate private services (for example, private variants of media streaming services). PBCs are a new data encoding that amortizes the computational costs associated with PIR, and are significantly more network-efficient than prior encodings. Thanks to these two techniques, our small deployment of Pung can scale out to support hundreds of thousands of users.

Contents

Chapter 1

Introduction

Can two or more users communicate over a public network like the Internet without anyone else learning of the existence of this communication? This is the central question studied in this dissertation. This question is decades old [61], but has received renewed interest due to a proliferation of controversial mass surveillance practices [32, 54, 95, 117, 118] that defy existing privacy laws and long-held beliefs [79, 195, 214, 215, 222], and the monetization of users' private information [33, 150, 161, 185]. Many privacy-conscious companies have responded to a weaker formulation of this question (and the aforementioned privacy threats) by deploying email and chat services that safeguard users' communications with *end-to-end encryption* [1–4, 116]. While end-to-end encryption hides the content of the messages exchanged—which is a notable achievement—it does not hide the messages' existence nor any of their associated metadata (identity of participants, number of messages, time and duration of communication, etc.).

Leaking communication metadata is troubling because it can be as sensitive and revealing as the actual content of the messages [200]. For instance, former NSA's General Counsel Stewart Baker states that "metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content" [196]. The former director of the NSA and the CIA Michael Hayden not only agrees with Baker, but further admits that "We kill people based on metadata" [72]. Several academic works study the amount and the type of information

that can be learned from analyzing metadata [158, 162, 187, 198], and their results echo Baker's statement.

Fortunately, the threat of metadata leakage has received considerable attention from academics and practitioners; there is a vast literature focused on preventing such disclosures [23, 43, 47, 60–62, 68, 74, 76–78, 86, 96, 139, 145–149, 164, 165, 170, 183, 192, 193, 199, 201, 216, 225]. While these works make great strides toward providing strong privacy guarantees (that is, hiding the data and metadata associated with users' communications), most require trusting one or more entities in the public network (for example, proxy servers, Internet service providers, large coalitions of users) to achieve their goals. In many contexts, trusting that certain entities will perform their job and will follow a prescribed protocol is a sensible and justified assumption. For example, it is reasonable to trust a service provider when its interests align with those of the service's users, or when an adversary does not have the resources to compel the provider to violate its duties. However, online communication is a particularly problematic setting. There is enough precedent to believe that any trustworthy provider, good intentions notwithstanding, can be subverted through technical, financial, or political means [28, 73, 150, 161, 197]. This stems from the strength of the adversary, which typically consists of nation states and well-connected organizations.

To withstand these very strong adversaries, there are several proposals based on Chaum's dining cryptographers (DC) networks [62] that hide metadata without requiring trusted intermediaries [77, 113, 121, 218]. Although these works inspire the content of this dissertation in many ways, we significantly diverge from them to avoid the prohibitive costs that are inherent in their architecture. In particular, these protocols are peer-to-peer and require messages to be communicated to all users. This has two negative consequences. First, all participants must know each other (by "know" we mean having everyone else's public keys, or sharing pairwise secrets). This introduces challenges in dynamic settings such as online communication, especially when dealing with high membership churn. Second, and more importantly, these protocols are very expensive: the network costs are quadratic in the number of users in the system. This is best exemplified by Dissent [77], which despite significantly ad-

vancing the state of DC network systems, supports only dozens of concurrent users.

Resolving this tension between trust and performance is the overarching theme of this dissertation. As part of the contributions of this work, we demonstrate that private communication can be achieved with reasonable performance, even in the presence of strong adversaries. To substantiate this position, we present Pung, a system that provably hides all metadata associated with users' conversations—even against adversaries that control all the communication infrastructure (ISPs, cloud providers, etc.) and arbitrary coalitions of users. An experimental evaluation of Pung confirms that Pung can support hundreds of thousands of users sending multiple messages per minute, which is $10^4\times$ more users than prior systems that withstand a similar adversary. When this comparison is extended to similar systems under a weaker threat model (for instance, Vuvuzela [216], Stadium [213]), Pung's performance is promising, but admittedly falls short of serving as a viable replacement: Pung handles 10–100× fewer users, and clients incur significantly higher network costs (§8.4).

To build Pung, this dissertation addresses two main challenges. The first is architectural: devising a way for users to send and receive messages without a trusted intermediary. The resulting proposal consists of combining untrusted servers and powerful cryptography through a synthesis of new and known ideas (§4). The second and more salient aspect of this work is reducing the costs of the underlying cryptographic machinery. Our contributions apply in contexts beyond Pung, and include algorithms to extend the interface of the cryptographic machinery (§4.3), to reduce its communication costs (§5), and to amortize its computational expense (§6). We discuss these in detail below.

1.1 High-level architecture and challenges

Pung is architected as an untrusted key-value store that exposes private deposit and retrieval procedures to users. Users can communicate with each other by depositing and explicitly retrieving messages via Pung. This model of communication is different from existing chat applications (such as WhatsApp) where the server sends a push notification to the recipient; instead, Pung more closely resembles email where

clients explicitly fetch their messages using POP3 or IMAP. A key distinction between the two, besides privacy, is that Pung operates in synchronous rounds to avoid leaking timing information, whereas email communication is asynchronous.

Pung's deposit procedure keeps the destination of a message private by exploiting the ability of communicating users to secretly agree on a shared *label* (or "key" in the key-value store context) under which to store a message (§4.2). Pung's retrieval procedure hides which message a user accesses by relying on a powerful cryptographic primitive: private information retrieval (PIR) [70]. PIR allows a client to fetch an element (such as a message in Pung) from a server without revealing to the server which element was fetched. While PIR is powerful, it is expensive and it is hard to integrate into a larger system as we outline below.

With regards to expense, PIR forces the server to operate over all the stored elements in order to answer a single client request [70]. After all, if the server could omit even one element when answering a request, then it would learn that the omitted element is of no interest to the client—violating the desired privacy guarantee. To put PIR's expense in context, recent work [49] uses reducibility to PIR as a litmus test on whether certain problems admit concretely practical solutions. This is akin to the use of NP-completeness to separate hard problems from easy ones: if a problem implies PIR, its solutions likely have poor concrete efficiency.

With regards to integration, PIR has a narrow interface. It requires the client to know the index of the desired element in the data structure that the server uses to store all of its elements. Not only does this introduce communication overhead since the server is forced to share this information with all clients, but in Pung, this data structure changes continuously (§4.3).

Despite the negative outlook on performance and the less than ideal interface of PIR, this dissertation is an exercise in optimism: to build Pung we do not need PIR to scale to arbitrarily large databases—supporting 7 billion entries is likely enough to provide worldwide communication. With this frame in mind, this dissertation makes several contributions to extend the interface of PIR, and to reduce the computational and network costs of PIR in practice. While the proposed techniques ultimately fall short of the intended 7 billion target, they represent a significant step forward.

1.2 Contributions

This dissertation describes the architecture of the first metadata-private messaging system that support hundreds of thousands of users and withstands an adversary that controls all communication infrastructure. To realize this architecture, our work weaves various existing and new cryptographic building blocks with careful system design. We also provide a formal proof that Pung's end-to-end design meets all of our privacy guarantees under standard assumptions. Incidentally, as part of writing the proof we uncovered a new attack that affects all existing messaging systems that hide metadata; we also describe how to mitigate this attack.

Besides building Pung, this dissertation introduces several technical contributions that are general and can be used in other contexts. In particular, this work discusses three extensions to the computational variant of PIR (CPIR) [143], which guarantees privacy under cryptographic assumptions (the other variant of PIR requires multiple non-colluding servers which conflicts with Pung's goals). These extensions are orthogonal but compose with each other, and alleviate PIR's drawbacks.

The first PIR extension is SealPIR, a new CPIR library that builds on top of the most computationally-efficient CPIR protocol, XPIR [20], and introduces a new query compression technique that reduces network costs (§5). Specifically, a query in XPIR (and its base protocol [207]), consists of d vectors of $\sqrt[d]{n}$ ciphertexts, where n is the number of elements in the server's database, and d is a small positive integer (the size of the response increases exponentially with d, so d is usually less than 4). SealPIR takes a different approach. Instead of creating a query vector, SealPIR has the client send a single ciphertext containing an encoding of the index of the desired element. The server then executes a new *oblivious expansion procedure* that extracts the corresponding n-ciphertext vector from the single ciphertext, without leaking any information about the client's index, and without increasing the size of the response (§5.6). The server then executes the rest of the XPIR protocol on the extracted vector.

In terms of concrete savings over XPIR, SealPIR results in queries that are 274× smaller and are 16.4× less expensive for the client to construct. However, SealPIR

introduces a 6% CPU overhead to the server (over XPIR) to obliviously expand queries. This constitutes an excellent trade-off since answering a PIR query is an embarrassingly parallel task, and one can regain the lost throughput by employing additional servers. Furthermore, reducing communication overhead makes PIR usable in settings where clients have limited bandwidth, such as mobile devices or wired connections with data limits [15].

The second extension is a technique to amortize the server's CPU cost when processing multiple PIR queries from the same client (multi-query PIR). This scenario applies when clients in Pung engage in group communication, or when clients exchange many messages in one round of communication (recall that unlike email, Pung is a synchronous communication system). The proposed technique is a relaxation of *batch codes* [127], which is a data encoding that was originally intended for this purpose. In practice, most batch code constructions target a different domain—providing load balancing and availability guarantees to distributed storage [177, 189] and network switches [221]; using these constructions to amortize the processing of a batch of PIR queries is not worthwhile since they introduce onerous network costs while yielding only modest CPU speedups (§6.1).

Our data encoding, called a *probabilistic batch code (PBC)*, addresses this issue at the expense of introducing a small probability of failure (§6). In the context of multi-query PIR, a failure simply means that a client can only get some (and not all) of her queries answered in a single interaction. While the implications of a failure depend on the application, we argue that this is not really an issue in Pung (§6.6.2). Moreover, the failure probability of our constructions is low—about one in a trillion multi-queries would be affected.

The key idea behind our PBC construction is a simple new technique called *reverse hashing* (§6.3). This technique flips the way that hashing (for instance, multi-choice [166], Cuckoo [174]) is typically used to build hash tables or to achieve load balancing in distributed systems: instead of executing the hashing algorithm during data placement and replicating queries during data retrieval, reverse hashing replicates data during placement and uses hashing during retrieval. Like batch codes, PBCs can be used to amortize the server's CPU costs when processing a batch of

PIR queries. Unlike batch codes, PBCs introduce orders of magnitude less network overhead (§8.3).

The third and last extension broadens PIR's interface in order to integrate SealPIR and PBCs with the rest of Pung. This is achieved through the introduction of an *oblivious search* technique that adapts prior work [69] to allow clients to retrieve messages without having to know the corresponding index in the server's data structure. More importantly, the proposed oblivious search technique works even when the server's data structure is encoded with a batch code or a PBC (§4.4.3), which ties together all of the techniques proposed in this dissertation.

1.3 Limitations

While Pung introduces a new point in the design space of private communication systems by leveraging powerful cryptography in lieu of trusting part of the network infrastructure, its costs remain high. Furthermore, like all past private communication systems, Pung does not hide the fact that users are part of the system (it only hides if and with whom they are communicating), nor does it provide location privacy. Pung also does not prevent analog attacks (for example, when a malicious WiFi router detects users' keystrokes by tracking the position of fingers through WiFi signals [24]), and requires clients' devices to constantly interact with the system to avoid leaking timing information. Finally, Pung does not provide liveness guarantees (censorship resistance). This is fundamental, since under our threat model, an ISP could simply refuse to route network packets.

1.4 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 gives a brief overview of other works in this general space, and how Pung compares to them. Chapter 3 discusses Pung's goals, assumptions, and concrete guarantees, and explicitly outlines the capabilities of the adversary. Chapter 4 provides the high level architecture of Pung, discusses how users send and retrieve messages, and how they bootstrap their

communication. This chapter also discusses how clients can use PIR to retrieve messages even when the server's data structure is encoded with a batch code.

Chapter 5 introduces SealPIR, a new PIR library used in Pung, but that can be used in other contexts and applications as well. Chapter 6 discusses PBCs, which are a relaxation of batch codes [127], and how they can be used in PIR to amortize the computational costs of the server. Chapter 7 discusses our prototype implementation of SealPIR, PBCs, and Pung, and Chapter 8 contains the corresponding experimental evaluation on a variety of deployment scenarios. Finally, Chapter 9 summarizes this work, outlines the remaining challenges, and discusses avenues for future work.

Chapter 2

Related work

This chapter discusses systems related to Pung, with a focus on the major architectural differences; for a detailed discussion of many of these works, we recommend the survey of Danezis, Diaz, and Syverson [83]. These systems, Pung included, are inspired by Chaum [61], who proposed the first communication system that hides both data and metadata. We group these works into five high-level categories: mix networks, peer-to-peer routing, onion routing, dining cryptographer networks, and private mailbox systems.

2.1 Mix networks

The most common architecture for private communication systems [43, 44, 60, 61, 86, 107, 120, 132, 145, 146, 148, 149] are networks of servers called *mixes* that route messages on behalf of users. Mixes batch requests from many users, shuffle them, add fake requests to serve as noise, and remove outer layers of encryption from messages. The result is that if a mix performs its duty, an adversary who observes network packets cannot correlate input messages (sent by clients) with output messages. By combining many of these mixes, and assuming that at least one is correct, clients can send messages anonymously. Furthermore, since the operations that mixes perform are relatively lightweight, these systems enjoy higher throughput than many works in the literature—including Pung.

One challenge with mix networks is that malicious mixes can replay, duplicate, and drop messages, violating the anonymity guarantees [151, 157, 172, 180, 181, 191, 203, 224]. To deter malicious mixes, systems like Riffle [146] and Atom [145] use *verifiable shuffle* protocols that force mixes to prove the correctness of their actions. Loopix [183], on the other hand, introduces message *loops* where particular messages loop around (potentially going back to the sender), allowing clients and mixes to detect if messages are being handled improperly. Finally, systems like Aqua [149] and Herd [148] target deployment scenarios where critical mixes are assumed to operate correctly.

Besides its architecture (as we discuss later), Pung differs from mix networks in its threat model, use cases, and guarantees. Specifically, Pung targets an ecosystem in which all servers are malicious (rather than some fraction of them), and supports private messaging (for example, email, chat) that hides metadata from anyone besides the sender and the recipient. Crucially, Pung does not provide anonymity: the recipient of a message in Pung knows the identity of the sender. As a result, Pung is not useful for anonymous content publishing (such as allowing a whistleblower to anonymously send documents to a news organization), which is one of the primary use cases of mix networks.

While it might be possible to redesign Pung into a mix network and relax its threat model to achieve anonymity, it might not be worthwhile. First, some of the existing systems are likely better alternatives. Second, Kesdogan et al. [131] show that mix networks are fundamentally susceptible to certain traffic analysis attacks, and Das et al. [89] show that strong anonymity cannot be achieved with low bandwidth and low latency. As a result, we believe that Pung strikes a good balance between supporting a meaningful application (private messaging), while providing provable guarantees under a realistic threat model.

2.2 Peer-to-peer routing

Peer-to-peer routing systems [35, 71, 85, 102, 104, 170, 192, 201], notably Crowds [192], offer an alternative to mix networks. In these systems, clients send a message to one

or more of their peers, who then randomly decide whether to forward the message to other peer nodes or to the final destination. While a benefit of these systems is their ease of deployment, they incur high network costs and require a threshold of peers to be correct in order to guarantee anonymity. For example, Salsa [170] requires that fewer than 20% of all nodes be malicious, whereas Blindspot [104] and `Drac` [85] suggest peering only with contacts from existing social networks, which leaks information about users' relationships and results in smaller anonymity sets. Furthermore, peer-to-peer routing systems are susceptible to network adversaries [99, 163, 206] and Sybil attacks [98].

Similar to mix networks, peer-to-peer routing systems differ from Pung in threat model (partial compromise versus full compromise), guarantees (anonymity versus metadata privacy), and use cases (Web browsing versus messaging).

2.3 Onion routing

Works based on onion routing [96, 164, 165, 210], especially Tor [96], are readily adopted due to their relative low latency and support for anonymous Web browsing. They have an architecture superficially similar to that of mix networks, but there are key differences. First, nodes are *routers* instead of mixes, meaning that they do not batch or shuffle requests from many users, which keeps latency low. Second, unlike mix networks where all clients use the same set of nodes, clients in onion routing select 3 or 4 nodes from a large list of geo-distributed volunteer nodes, and route messages through them. One drawback of onion routing systems is their inability to resist routing attacks [209] and traffic analysis attacks [125, 168, 191], including those performed by local adversaries [56, 144, 176, 219]. Furthermore, many volunteer nodes can be malicious, making it difficult for clients to select safe routes. While there is hope that future Internet architectures may address many of these shortcomings [63, 64], Pung is designed to work in today's ecosystem.

2.4 Dining cryptographers networks

Another line of work initiated by Chaum [62] is the Dining Cryptographers (DC) network [62, 77, 113, 121]. Unlike the prior categories of systems, DC networks target the same threat model as Pung, while supporting both metadata-private messaging and anonymous publishing. In this regard, DC network systems are more flexible than Pung. Their drawback, and what motivates Pung's techniques, is that they are peer-to-peer (requiring all users to know each other) and are based on all-to-all broadcast of messages, which results in prohibitive network communication. Despite their appealing guarantees, DC network systems typically accommodate only hundreds of users.

Verdict [78] and Dissent in numbers [225] make great strides to reduce the costs of DC networks. These systems support thousands of users, but in the process they relax DC networks' threat model and introduce servers of which at least one must be trusted. While Pung cannot support anonymous publishing, it can support hundreds of thousands of users while retaining a desirable threat model.

2.5 Private mailboxes

Finally, there are a number of systems [23, 47, 74, 76, 139, 146, 199, 216] that employ an architecture and techniques similar to Pung's (clients privately retrieve messages from mailboxes kept at third-party servers). The key differences between these works and Pung is their reliance on at least one correct server, and the mechanisms that follow from that assumption. We elaborate on the most related ones below.

P^3 [139], like Pung, employs a key-value store from which users can privately pull messages. While P^3's focus is a retrieval mechanism that supports general queries when fetching a message (for instance, prefix search), Pung's primary goal is to drive down the cost of retrieval by introducing new protocols (§5) and batching optimizations (§6).

Riposte's [76] mechanisms are the opposite of Pung's: while Riposte hides which messages a user deposits into the servers (in order to provide sender anonymity),

Pung hides which messages a user retrieves. The Pynchon Gate [199] provides anonymity by composing a mix network with a private information retrieval (PIR) protocol, which is a primitive that Pung also uses (§4.3.1) and improves (§5). However, the Pynchon Gate's guarantees hold only for passive adversaries that do not compromise mixes; under our threat model, several attacks exist [172, 180, 181, 224].

Vuvuzela [216] and Stadium [213] have a mix network architecture and provide privacy through request shuffling and the careful addition of cover traffic. These systems have lower costs and achieve better performance than Pung (§8.4.1, §8.4.3), but Pung has some benefits. In Vuvuzela and Stadium, messages are ephemeral and can be accessed only during one round; Pung supports long-lived messages that can be retrieved any time prior to garbage collection (§7). Unlike these works, Pung supports group communications in addition to point-to-point exchanges. Finally, the guarantees of a Vuvuzela and Stadium deployment are based on differential privacy and are valid only for a certain number of rounds (based on a privacy budget). Pung's guarantees hold for any number of rounds.

Chapter 3

Goals, assumptions, and threat model

This chapter makes explicit the goals and assumptions of this work, and the general ecosystem in which we intend for Pung to be deployed.

3.1 Target ecosystem and guarantees

We strive for a messaging system that allows two or more users to communicate over the Internet (or any other public network) while hiding the content of all messages exchanged in addition to the metadata associated with the exchange. The metadata that we wish to hide from anyone—except from the users directly involved in the conversation—includes the start and end time of a conversation, the frequency of messages exchanged, the size of messages, and the identity of participants. Some of this information is difficult to keep private since existing services rely on it for their proper functioning. For instance, ISPs need to know the destination of a message in order to route packets. Consequently, Pung must be compatible with existing services and infrastructure while meeting the following security goals:

Message integrity and privacy. The content of a message must be intelligible only to its intended recipient. Furthermore, no one should be able to tamper with a message while it is in transit without the recipient being able to detect alterations. Specifically, we target two cryptographic properties that capture these goals, namely *indis-*

tinguishability under adaptive chosen ciphertext attacks (IND-CCA2) [171, 186], and *integrity of ciphertexts under chosen plaintext attacks* (INT-CTXT) [39, 130].

Metadata privacy. An adversary must not be able to determine if (or when) a user sent or received a message. Furthermore, an adversary must not be able to link a message exchange with the users who participated in that exchange. Specifically, we target the privacy notion of *relationship unobservability* as defined by Pfitzmann and Hansen [179]. Informally, relationship unobservability states that an adversary does not learn useful information from observing (or actively interfering with) all network traffic, provided that the sender and the recipient are not compromised. In the case where the adversary compromises the sender or the recipient, relationship unobservability offers little value: the sender could trivially disclose that it is sending a message and to whom, and the recipient could similarly leak the sender's identity.

Relationship unobservability is sufficient for our setting of two-way communication. However, this property does not protect the identity of a whistleblower who wishes to remain anonymous from everyone, including the recipient; that would require *sender anonymity* [179]. For settings where anonymity is desired, a related class of systems—including the Tor anonymity network [96]—is more appropriate; we discuss these systems in Chapter 2. We give a formal definition of metadata privacy and prove that Pung meets this definition in Appendix C.

Non-guarantees. Pung does not attempt to hide the fact that a user is part of the system, the geographic location of the user, or the maximum number of concurrent conversations that a user can have. Indeed, these limitations are shared by all other systems related to Pung. Hiding that a user is part of Pung relates to the goal of many steganographic systems (see for instance the work of Weinberg et al. [223]), which remains elusive in practice [126]. Pung provides no guarantees in the presence of targeted attacks that compromise a user's device (for example, malware, malicious firmware) or that determine the user's actions through analog channels (such as video cameras that observe users' screens, or routers that detect users' keystrokes through WiFi signals [24]). Finally, Pung does not prevent censorship: a government

or network operator could stop users from accessing Pung.

3.2 Assumptions

Pung provides the above guarantees and can be deployed in our target ecosystem only if the following assumptions hold true.

Cryptographic assumptions. Pung assumes the existence of an *authenticated encryption* (AE) scheme [39], a public key cryptosystem with *key privacy* [38] and *weak robustness* [17], a *pseudorandom function* (PRF) [111], and a *computational private information retrieval* (CPIR) scheme [143]. AE schemes such as AES-GCM [159] hide the content and preserve the integrity of a message, and are part of TLS 1.2 [93, §6.2.3.3] and end-to-end encrypted chat and email services (for example, WhatsApp, Signal). Public key cryptosystems that provide key privacy (that is, a ciphertext does not leak which public key was used to encrypt a message) and weak robustness (meaning that a ciphertext is only valid under a single encryption key) include Cramer-Shoup [80], Kurosawa-Desmedt [142], and DHIES [18, 40]. PRFs, for example HMAC-SHA256, are functions whose output on a given input is indistinguishable to an adversary from the output of a truly random function. We review CPIR schemes in Section 4.3.1, and give one construction in Chapter 5.

Pung's protocol relies on an AE scheme for message integrity and privacy, the robust key-private cryptosystem to bootstrap the communication, and the CPIR and PRF schemes for metadata privacy. In Section 7 we discuss the instantiations that we use for each of these primitives, but Pung's design is independent of these choices.

Trust assumptions (threat model). Pung assumes that pairs (or groups) of users who wish to communicate know their peers' public keys (or can exchange a secret through an out-of-band channel). Pung provides privacy guarantees only to pairs (or groups) of users who communicate through Pung while following the prescribed protocol. However, these guarantees are not predicated on the behavior of any other user in the system, or the communication channel between users. In other words,

Pung's guarantees for a correct user u hold even if there is compromise or arbitrary behavior of all of the infrastructure that Pung uses (servers, ISPs, DNS, etc.) and all users not directly conversing with u. We therefore assume that the adversary is malicious and may control all infrastructure and any subset of users.

Liveness assumption. Pung assumes that its own servers (which can act arbitrarily) as well as online services typically used to communicate over the Internet (such as DNS) do not deny service. That is, we expect ISPs to carry traffic, DNS to provide name resolution, and servers to process requests. While this assumption is not needed for Pung to meet its security and privacy guarantees (§3.1), it is essential for Pung to be usable in practice.

3.3 Alternate trust models

Pung adopts a threat model where the adversary controls all of the existing infrastructure. A benefit of this model is ease of deployment: any provider can deploy Pung, and it is compatible with standard monetization strategies such as having the operator charge customers for using the system. As we discuss in Chapter 2, weaker threat models such as threshold assumptions (for example, that a fraction of servers or services are not compromised) can be used to design alternatives to Pung that achieve better performance, but these alternatives have a harder deployment path. Specifically, it is difficult to find an uncorruptible consortium of servers to run the system—especially given the likely adversary of a metadata-private communication system (such as nation states or well-connected organizations).

Chapter 4

Design and architecture of Pung

This chapter presents the general architecture of the Pung metadata-private messaging system. Pung is a client-server system in which third-party servers mediate the exchange of messages between users. Figure 4.1 depicts this architecture. From the perspective of end users, Pung's servers act as a storage service. This parallels services like Gmail or Outlook that store messages on behalf of users.

Users exchange messages with each other via a Pung client application that deposits client messages into *mailboxes* located at Pung's servers. These mailboxes are addressed by a *label* that is known to both the sender and the recipient (but nobody else). Recipients can access a message sent to them by retrieving the contents of a mailbox from Pung's servers using an appropriate label. Pung's mailbox architecture borrows heavily from prior systems [47, 74, 139, 146, 199, 216]. The key differences are which entities run the storage nodes, the kinds of processing that these nodes do, and the mechanisms for storing and retrieving messages. We discuss each of these components in the following sections, but we first highlight how this architecture fits within our target ecosystem.

Pung's mailbox architecture forces all messages sent and retrieved to go through

This chapter contains material from two previously published works: "Unobservable communication over fully untrusted infrastructure" (OSDI '16) by Sebastian Angel and Srinath Setty [27], and "What's a little leakage between friends?"(WPES '18) by Sebastian Angel, David Lazar, and Ioanna Tzialla [26]. Sebastian contributed to all aspects of the design, implementation, and experimental evaluation of the system and attacks described in this chapter.

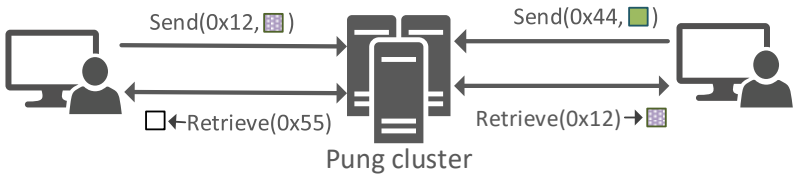


Figure 4.1: Pung client applications issue send and retrieve requests to the Pung cluster at a given rate, introducing dummy requests whenever the user is idle (or when the user issues fewer requests than the established rate).

entities like ISPs and Pung's servers. These services rely on (or can easily infer) the types of metadata that we wish to hide, since they process all network traffic. Consequently, protecting metadata without harming the functioning of these services requires that the act and the rate of sending and receiving network packets be disentangled from the act and the rate of sending and retrieving messages in Pung. In other words, just because a network packet is sent by a client does not mean that the network packet contains an actual message. This requirement is key to preventing many types of traffic analysis attacks [82, 134, 191] in which an adversary can observe when messages are sent or received, and can, over a long period of time, establish which users are communicating. Unfortunately, this disentanglement results in an unavoidable inefficiency: clients must send and receive network packets at a rate that is independent of users' actual communication (such as maintaining a constant rate), even when a user is idle. This forces clients to queue requests that are in excess of this rate and add cover traffic or *chaff* [194] (dummy requests that are indistinguishable from real ones) when the user sends few messages or is idle.

4.1 Mailbox labels and discrete rounds

Pung operates in discrete rounds or time epochs. Round duration is configurable and depends on the use case. Pung's servers act as a point of synchronization for clients and dictate when a new round starts. While this allows Pung's servers to force clients out of sync, doing so results in a denial of service but does not violate our goals of privacy or integrity (§3.1). In particular, clients keep track of the current

round and increase it monotonically. During each round, client applications send exactly one message and retrieve exactly one message from Pung's servers. This ensures that clients issue requests at a constant rate that is independent of the users' intent. In Chapter 6 we relax this model and let clients issue multiple send and retrieve requests per round, enabling several applications, and achieving lower (amortized) costs (§4.4). Finally, Section 4.5 discusses how clients can manage existing connections (for example, add friends), and how they can agree on a round on which to start a new conversation.

Deriving mailbox labels. Pung's servers effectively act as a key-value store service that treats mailbox labels as keys and (encrypted) messages as values. This means that users' communication depends on their ability to agree on a label under which to store and retrieve messages. This label should be unique (to avoid multiple pairs of users overwriting each other's messages), and it must also be independent of the users communicating, as otherwise an adversary could link a label to a conversation. Pung achieves both of these properties through a combination of shared secrets and a pseudorandom function (PRF).

Recall from Chapter 3.2 that we assume that users who wish to communicate have access to each other's public key, or have exchanged a secret through an out-of-band channel. In Section 4.5.2 we discuss how a user can add some other user (identified by a public key) as a friend, and derive a shared secret. Consequently, the next few sections assume that pairs of users who want to communicate share a secret that acts as a master key. This master key is used to derive two additional keys, k_L and k_E, with a *key derivation function* [141] that ensures that the resulting keys are uniformly distributed and indistinguishable from truly random keys. The derived keys are used for mailbox label generation and message encryption, respectively. We also assume that users have a unique identifier, *uid*, within each pair of communicating users. For example, if Alice and Bob wish to communicate with each other, the one with the lowest name in lexicographic order (Alice) could be "0", and the other (Bob) could be "1". This information need not be private (nor does it need to be an integer), so users could choose any identification scheme including using their

names or public keys as *uid*s.

Each user can derive the corresponding labels for the current round r, $label_S(r)$ and $label_R(r)$, by invoking the pseudorandom function (PRF) keyed with k_L:

$$label_S(r) = \mathrm{PRF}_{k_L}(r \,||\, uid_{peer})$$

$$label_R(r) = \mathrm{PRF}_{k_L}(r \,||\, uid_{own})$$

where the round number r is a fixed-width integer and $||$ is the concatenation operator when r and *uid* are treated as binary strings. Note that labels need not be symmetric: a user can send a message to Alice and retrieve one from Bob in the same round. In such cases, the labels would be generated using different keys and *uid*s. If a user is idle and has nothing to send or retrieve, it generates random mailbox labels of the appropriate width.

4.2 Sending messages in Pung

Sending a message in Pung consists of deriving the recipient's mailbox label ($label_S$, which stands for "label to use for sending"), and encrypting the message m with an authenticated encryption scheme (§3.2) using key k_E, and the round r as a nonce. The client then sends the resulting ciphertext, $c = AE_{k_E}(m, r)$, along with $label_S$ to one of Pung's servers as a $(label_S, c)$-tuple. Idle users send a tuple that consists of a random label and an encryption of a random message instead. We ensure that all messages are the same size by: (1) padding small messages, and (2) breaking up large messages into (potentially padded) chunks and prepending additional information (to the first chunk which is then encrypted) to allow the recipient to piece together the large message once it has received all the chunks after several rounds.

For security, it does not matter to which particular Pung server a user sends their tuple since all servers belong to the same untrusted provider. For performance, an operator might wish to redirect users to particular servers to balance the load or to provide better service (for instance, if a server is geographically closer to the user). In the rest of this chapter we will simply assume that clients send and retrieve

messages from "Pung's server", which is just a highly-replicated logical entity.

4.3 Retrieving messages from Pung's server

Observe that if Pung's server were to broadcast to all users the $(label, c)$-tuples received during a round, users could iterate through the list locally and find the tuple with the label that is of interest to them (or determine that it is not present). Intuitively, this operation would not leak any information about which label (if any) was of interest to a retriever, and would not allow the adversary to determine with whom a user is communicating (or if the user is idle). Of course, broadcasting all tuples would incur prohibitive network costs. Fortunately, retrieving an item from an untrusted server without revealing *which* item was retrieved is the problem addressed by a powerful cryptographic primitive: private information retrieval (PIR) [70]. PIR protocols trade off computation at the server to achieve lower network costs than the above broadcast scheme. We summarize PIR next since it is the basis of message retrieval in Pung, and we discuss it in detail in Chapter 5 where we propose an extension to an existing construction.

4.3.1 Background: Private information retrieval (PIR)

Chor et al. [70] introduce private information retrieval (PIR) to answer the following questions: can a client retrieve an element from a database managed by an untrusted server (or set of servers) without the server learning *which* element was retrieved by the client? And can this be done more efficiently than simply having the client download the entire database? Chor et al.'s affirmative response inspired two lines of work: *information theoretic* PIR (IT-PIR) and *computational* PIR (CPIR).[1]

In IT-PIR schemes [36, 70, 90, 92, 110] the database is replicated across several non-colluding servers. The client issues a carefully constructed query to each server (that reveals no information as long as the servers do not collude) and combines the responses from all of the servers locally. IT-PIR schemes have two bene-

[1] Also known as multi-database PIR (IT-PIR) and single-database PIR (CPIR).

fits. First, the servers' computation is relatively inexpensive: an XOR for each entry in the database. Second, the privacy guarantees are information-theoretic, meaning that they hold even against computationally-unbounded adversaries (so there is no need for cryptographic hardness assumptions). However, basing systems on IT-PIR poses a significant deployment challenge since it can be difficult to enforce the non-collusion assumption in practice (see the discussion in Section 3.3).

On the other hand, CPIR protocols [20, 51, 55, 59, 97, 109, 137, 143, 152, 153, 227] can be used with a database controlled by a single operator (which is the setting that Pung targets), under cryptographic hardness assumptions. The drawback is that they are more expensive than IT-PIR protocols as they require the database operator to perform costly cryptographic operations on each database element. Fortunately, there is a long line of work that focuses on improving the resource overheads of CPIR schemes (see [20, 137] for the state-of-the-art); recent work [20] proposes a construction that achieves, for the first time, plausible (although still high) computational costs.

CPIR protocol. We will give a concrete CPIR protocol in Chapter 5. For now, we discuss only its interface. Our CPIR protocol operates over a collection DB of n items held by a server, and consists of three procedures: Query, Answer, Extract. The $\textsc{Query}(pk, idx, n)$ procedure is run by the client; it outputs a query q that encodes the index, idx, in DB of the desired element with the client's encryption key pk. The $\textsc{Answer}(q, DB)$ procedure is run by the server; it returns an encrypted response a that contains the element in DB at the index encoded in q. Producing the response a requires the server to perform cryptographic operations over *every* element in DB. That is, the running time of the computation that the server performs is linear in n. The intuition for this is simple: if the server were to omit even a single element while still being able to correctly answer the query, then the server would learn that the client did not request the omitted element. The $\textsc{Extract}(sk, a)$ procedure is run by the client; it decrypts a with the client's decryption key sk to recover the desired element in DB.

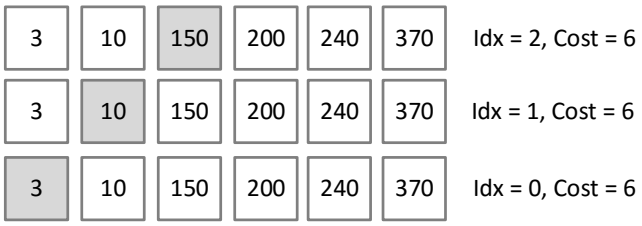


Figure 4.2: A client wishing to retrieve an item with label "3" from a server holding a sorted list of 6 items would need to perform three rounds of probing. During each probe, the client guesses an index, uses PIR to retrieve the $(label, c)$-tuple at that index, and refines the guess accordingly. "Cost" indicates the number of items processed by the server in each probe.

4.3.2 Retrieving messages

Since PIR allows clients to privately retrieve an item from a server at some index, one possibility is to use labels as indices: clients can retrieve a message from $label_R(r)$ with $q = \textsc{Query}(pk, label_R(r), n)$. However, the size of the collection (n) would need to match the range of the labels (§4.1), which is 256 bits in our implementation (§7). This would require Pung's server to input a collection of 2^{256} elements to Answer, consisting of a combination of real tuples (sent by users) and dummy tuples (needed to satisfy the structure and size expected by PIR).

Instead, we can arrange for Pung's server to insert all tuples sent by clients in some search data structure (such as a sorted list or a search tree) and present them as a collection *DB* of size n (where n is the total number of nodes in the search data structure). This enables clients to perform PIR directly on the search data structure, but there is a problem: clients know from which mailbox they wish to retrieve ($label_R(r)$), but they do not know the mapping between labels and the index of the desired tuple in the data structure representing *DB*, or if the tuple even exists. This can easily be addressed by having clients obtain this label-to-index mapping explicitly from Pung's server. However, when the collection is large (for example, n>10K), downloading this mapping consumes significant network resources; we show how clients can use a search scheme to reduce network costs below.

The key idea is that clients can find their desired element in *DB* via an "oblivi-

ous" search procedure. This procedure consists of a client guessing an index, issuing a PIR query to Pung's server requesting the element at that index, and then gradually refining the index based on the retrieved element and the known structure of *DB*. Figure 4.2 depicts an example of this search when *DB* is stored as a sorted list. In this case, the client guesses an index and performs $\log(n)$ probes to locate its desired element (or determine that it is not present), refining the index after each probe. Even if the client gets lucky and finds its element early, it must continue until the end to preserve privacy; the remaining probes can just use any index. Since each probe is a PIR query to all of *DB*, the server must process n elements each time (recall that ANSWER is linear in n); the computational complexity of this search is therefore $\Theta(n\log(n))$. However, this scheme has a lot of redundancy since the server processes each item $\log(n)$ times. Chor et al. [69] show that one can eliminate this "double counting" overhead by using data structures that can be (logically) split into independent chunks while retaining their search capability. We elaborate on this idea below in the context of the specific construction that Pung uses.

4.3.3 Retrieving messages from large collections using a BST

We choose to use a complete[2] binary search tree (BST) as our underlying data structure for several reasons. First, a complete BST is balanced, enabling search in $\mathcal{O}(\log(n))$ probes. Second, for any collection there is a unique complete BST, so the server needs not communicate the structure to clients (aside from n). Last, since every level of a complete BST is full (except for possibly the last) and every node contains an actual data item, there is no need for padding or auxiliary elements that aid the traversal; it can be represented as a contiguous array without overhead. This last point is crucial. While Chor et al. [69] also propose other search data structures that result in fewer probes—and therefore lower network costs—using a BST adds no computational overhead to the PIR protocol over the baseline where the client knows the index a priori. Given that computation is the current bottleneck in CPIR protocols and that Chapter 5 introduces techniques to reduce network costs, using a BST appears the

[2] A tree is complete if all levels (except the last) are full, and the last level is filled from left to right.

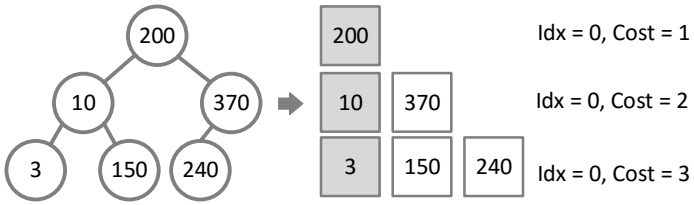


Figure 4.3: The Pung cluster can store $(label, c)$-tuples in a complete BST, allowing clients to treat each level as an independent collection. Clients can issue a PIR query for the top level, and can recursively derive the index of lower levels using BST semantics. This figure depicts the search for label "3".

best choice for our use case (at least asymptotically).

We therefore set up Pung's server to store the collection of $(label, c)$-tuples in a complete BST, and have clients treat all the nodes at the same depth in the tree (that is, on the same level) as a (logically) separate collection. As depicted in Figure 4.3, clients can then query each of the $\log(n)$ collections sequentially from top to bottom, deriving the index of the next level from the semantics of the BST. The pseudocode for this procedure is listed in Figure 4.4. Since each collection (and therefore each element) is accessed exactly once, there is no overhead due to double counting; as expected, the computational complexity of this BST-based PIR retrieval scheme is $\Theta(n)$, which is the same as if the clients had known the index in the first place. Compared to performing PIR over a known index, clients do incur $\log(n)\times$ higher network costs due to retrieving a tuple at every level. As an optimization, clients could fetch (non-privately) all of the tuples of the first few levels, saving both bandwidth and CPU. This is because CPIR queries and answers are typically much larger than the elements in the collection (tuples in Pung); when the collection is small, it is more efficient to download all elements (naive PIR) than to use a CPIR scheme.

The next section proposes an alternative to using a BST that results in lower communication costs in practice when the collection is small.

```
1:  function BST-Retrieval(pk, sk, L*, n)
2:      h ← ⌊log2(n)⌋                                   // last level of the BST
3:      c* ← ⊥                                  // target ciphertext (⊥ means not yet found)
4:      idx ← 0                                          // index of the current level
5:      len ← 1                                          // length of the current level
6:
7:      for i = 0 to h do
8:          // use PIR to get element at position idx from collection at level i of the BST
9:          q ← Query(pk, idx, len)                                  // see Figure 5.1, Line 4
10:         a ← send i and q to server and get answer
11:         (L, c) ← Extract(sk, a)                                  // see Figure 5.1, Line 14
12:
13:         if c* == ⊥ then
14:             if L* < L then                                       // access left child next
15:                 idx ← 2 · idx
16:             else if L* > L then                                  // access right child next
17:                 idx ← 2 · idx + 1
18:             else                                          // L* == L, found target ciphertext
19:                 c* ← c
20:
21:         // length of the next level of the BST (last level, h, might not be full)
22:         if i < h − 1 then
23:             len ← 2^(i+1)
24:         else
25:             len ← n − (2^h − 1)
26:
27:         if idx ≥ len or c* ≠ ⊥ then
28:             idx ← random index between 0 and len − 1
29:     return c*
```

Figure 4.4: Client procedure for retrieving an encrypted message $c^\star$ from a mailbox with label $L^\star$. The keys *pk* and *sk* are an encryption/decryption key pair as we explain in Chapter 5. The server holds a collection of n $(label, c)$-tuples in a complete binary search tree. The client issues a PIR query at every level of the tree (initially getting the root). As the client reads the label of the retrieved tuples, comparing this label with $L^\star$ and the semantics of a BST inform the choice of the index at the lower level of the tree. The client probes all levels regardless of whether the tuple associated with $L^\star$ is found at an earlier level.

4.3.4 Retrieving messages from small collections

In the previous section we show that when the size of the collection (n) is large, BST retrieval incurs lower network overhead (logarithmic in n) than explicitly downloading the label-to-index mappings (linear in n) and performing PIR with a known index. We now describe how to delay the *breakeven point* (meaning the value of n at which BST retrieval is better than explicitly downloading labels) by using a *Bloom filter* [45]. A Bloom filter is a probabilistic data structure that encodes a compressed representation of a set, and is widely used to reduce network costs in many settings, including private communication [147, 184] (although our use case is different). It exposes a *check* procedure that allows anyone to check whether an element is in the set; false positives are possible and occur with small probability.

In our protocol, after Pung's server has received all messages from users for a given round, the server creates a Bloom filter as follows. For each $(label, c)$-tuple in DB, the server adds to the Bloom filter the bit string: $idx \,||\, label$; idx is the index of the tuple in DB, $label$ is the tuple's label, and $||$ is the concatenation operator. Pung's server then sends the resulting Bloom filter to clients, who can then find the index of their desired label $L^\star$ by testing for set membership locally. In particular, clients use the check procedure of the Bloom filter while varying the index until they find a match: $check(0 \,||\, L^\star), \ldots, check(n-1 \,||\, L^\star)$. While standard Bloom filters require computing a large number of hash functions for each add and check operation, there exist constructions that require only two [138]. Thus, with little computation, clients can locally derive their desired index while saving network resources. For very large collections, retrieval via BST (Fig. 4.4) is still the most efficient option, since the size of a Bloom filter is linear in n (asymptotically the same as explicitly sending the label-to-index mapping, but with lower constants).

4.4 Cheaper group communication with batch codes

One compelling application of Pung is private group communication. This includes not only group chat, but also discussion boards and collaboration tools (such as

Slack). One way for Pung to support group communication is for each member in the group to have a key that is shared with every other member. Clients can use these keys to derive the label of every member in the group, and retrieve a message from all labels (we describe the specific protocol in Section 4.4.4). One issue with this proposal is that, as we discuss in Section 4.1, clients must send and retrieve messages at a rate that is independent of their communication pattern. Consequently, clients who wish to participate in a group with up to k people must configure their retrieval rate to be at least k (that is, clients retrieve k messages every round, even if they are not communicating with anyone). This increases the network communication and the computational load on Pung's server linearly with k, which is undesirable given PIR's high costs.

The good news is that while reducing the computational resources needed for the server to answer a single PIR query appears hard, there are several proposals to amortize the costs of processing multiple PIR queries. Below we survey a few of these ideas, and we then give background on the particular technique that Pung uses to reduce the computational resources of the server when processing a batch of queries. We improve on this technique in Chapter 6 (we describe it in a different chapter for modularity, since the technique is general and can be used in other applications).

4.4.1 Existing PIR amortization approaches

Beimel et al. [37] describe two query amortization techniques. The first is based on the observation that queries in many PIR schemes consist of a vector of entries, and answering these queries is equivalent to computing a matrix-vector product (where the product could be over ciphertexts instead of plaintexts, or it could be an XOR operation). By aggregating multiple queries—even from *different* users—the server's work can be expressed as a product of two matrices. As a result, subcubic matrix multiplication algorithms (such as Coppersmith-Winograd [75] or Strassen's algorithm [208]) provide amortization over multiple matrix-vector multiplication instances. This approach is further studied by Lueks and Goldberg [155] in the context of Goldberg's IT-PIR scheme [110].

The second proposal described by Beimel et al. [37] is to preprocess the database in certain IT-PIR schemes to reduce the cost of future queries. Since this works well, recent proposals [48, 58] employ an analogous approach in CPIR schemes. However, making the preprocessed database accessible by more than one client under these schemes requires cryptographic primitives that are currently too inefficient to be implemented (*virtual black-box obfuscation* [34] heuristically instantiated from *indistinguishability obfuscation* [105]).

Several works [90, 119, 123, 124] extend specific PIR schemes to achieve CPU, disk IO, or network amortization. For example, Popcorn [122] pipelines the processing of queries in IT-PIR to amortize disk I/O, which is a bottleneck for databases with very large files such as movies. Related to CPIR, Groth et al. [119] extend the scheme of Gentry and Ramzan [109] to retrieve k elements at lower amortized network cost by having the client compute k discrete logarithms (with tractable but expensive parameters) on the server's answer. This results in low network costs, but Gentry and Ramzan's scheme is computationally expensive (tens of minutes to process one PIR query, based on our estimates); the extension of Groth et al. compounds this issue.

Finally, the most general approach to achieve the type of amortization that we seek is a *batch code* [127]. A batch code is a type of encoding that can be applied to the PIR server's database (which is made up of n elements) to obtain many small databases. The property that batch codes provide is that a client can get any k elements from the original n-element database by querying all of the small databases at most once. Since batch codes guarantee that the sum of the number of entries across all the small databases is less than kn, this can be used to amortize computational costs. Furthermore, batch codes treat PIR as a black box, so they work with *any* PIR protocol (information-theoretic or computational), and compose with other optimizations (for example, Beimel et al.'s [37] matrix multiplication proposal).

In Pung, we use batch codes to reduce the computational cost of group communication. The next sections give detailed background on batch codes, and introduce a new technique that allows a client to continue to use BST retrieval (§4.3.3) even when the server's collection has been encoded with a batch code.

4.4.2 Background: Batch codes

A (n, m, k, b)-batch code $\mathcal{B}$ takes as input a collection DB of n elements, and produces a set of m codewords, C, distributed among b buckets with three key properties: *locality*, *availability* to sets (or multisets) of k elements in DB, and the number of codewords (m) grows sublinearly with k. Formally,

$$\mathcal{B} : DB \rightarrow (C_0, \ldots, C_{b-1})$$

where $|C_i|$ is the number of codewords in bucket i, and the sum of codewords across all buckets is $m = \sum_{i=0}^{b-1} |C_i| \geq n$.

Locality [114] means that one can recover an element in the input collection (DB) by accessing only a small number of codewords (in C). Availability [188] to sets (or multisets) of k elements means that any set (or multiset) of k elements from DB can be recovered from k disjoint sets of codewords (in C). In other words, any k elements from DB can be retrieved from the b buckets by fetching at most one codeword from each bucket.

The last property of batch codes is that $m < k \cdot n$, which is precisely what ensures computational amortization in PIR. Recall from Section 4.3.1 that PIR's costs are linear in the size of the collection, which is m after encoding; the amortized per-request cost is $\mathcal{O}(m/k)$. In contrast, running k instances of PIR directly on DB results in computational costs linear in $k \cdot n$, which corresponds to a per-request cost of $\mathcal{O}(n)$.

Example. We describe a $(4, 6, 2, 3)$-batch code, more specifically the *subcube batch code* [127]. Let $DB = \{x_1, x_2, x_3, x_4\}$. For the encoding, DB is split in half to produce 2 buckets, and a third bucket is produced by XORing the entries in the first two buckets: $\mathcal{B}(DB) = (\{x_1, x_2\}, \{x_3, x_4\}, \{x_1 \oplus x_3, x_2 \oplus x_4\})$. Observe that one can obtain any 2 elements in DB by querying each bucket at most once. For example, to obtain x_1 and x_2, one can get x_1 from the first bucket, x_4 from the second bucket, and $x_2 \oplus x_4$ from the third bucket. One can then locally recover x_2 by computing $x_2 = x_4 \oplus (x_2 \oplus x_4)$.

This encoding is helpful for PIR because a client wishing to retrieve 2 elements from DB can, instead of querying DB twice, issue one query to each bucket.

The server is in effect computing over 3 databases with 2 elements each, which results in 25% fewer operations than computing twice over one database of 4 elements. This benefit, however, comes with two drawbacks. First, using a batch code increases network costs superlinearly with k: by increasing the number of databases from 1 to 3 in the above example, the client is forced to generate 3 queries and receive 3 answers (versus 2 queries and 2 answers if a batch code had not been used). Chapter 6 discusses this drawback at length, and introduces a relaxation of batch codes that is significantly more efficient.

A second drawback of batch codes is that they are not compatible with existing keyword or label-based retrieval schemes (§4.3.3). We discuss this below.

4.4.3 Retrieving messages from encoded collections

Recall from Section 4.3.3 that when the collection is large, it is beneficial for Pung to use the BST retrieval algorithm of Figure 4.4. However, this algorithm no longer works when the collection is encoded with a batch code since some of the entries may be XORs of other entries, and it is unclear where in the BST to place these encoded entries. Furthermore, clients cannot directly compare their target label $L^\star$ to an encoded label, complicating tree traversal.

We now show how to adapt BST-Retrieval (Figure 4.4) to work on collections that have been encoded with a subcube batch code. We focus on a $(n, \frac{3}{2}n, 2, 3)$-subcube batch code (this was our earlier example with $n = 4$), but our approach generalizes.

Server setup. The server starts with a collection of n tuples, which it sorts based on labels. Analogous to the batch code scheme described earlier, the server splits the collection into two halves, and stores each of them as a complete BST. Call the resulting BSTs b_1 and b_2. Finally, the server creates a third binary tree, b_3, from b_1 and b_2 by computing element-wise XORs as follows: for every level i and index j, $b_3(i,j) = b_1(i,j) \oplus b_2(i,j)$. Note that unlike b_1 and b_2, b_3 is not a BST since its nodes are the XOR of the nodes in the other trees, and they are not guaranteed to follow

BST semantics (that the left child has a lower value and the right child has a higher value than its parent).

The server then indicates to clients the collection size (n) and the lowest label in b_2, L_{mid}; tuples with labels lower than L_{mid}, if they exist, would be found in b_1.

Client lookup. A client wishing to retrieve two elements labeled L_1 and L_2 does so as follows. Assume without loss of generality that $L_1 < L_2$. There are two cases:

- If $L_1 < L_{mid}$ and $L_2 \geq L_{mid}$: the client calls BST-Retrieval($L^\star$, $\frac{n}{2}$) on each of the three trees, passing the label L_1 as $L^\star$ for b_1, the label L_2 as $L^\star$ for b_2, and a random label as $L^\star$ for b_3.
- If $L_1 < L_{mid}$ and $L_2 < L_{mid}$, the client calls BST-Retrieval(L_1, $\frac{n}{2}$) on tree b_1, and performs a *joint tree traversal* on b_2 and b_3 to retrieve L_2 (the case where both $L_1 \geq L_{mid}$ and $L_2 \geq L_{mid}$ is symmetric and simply requires exchanging the role of b_1 and b_2). We describe joint tree traversal next.

Joint tree traversal. Since b_3 is not a BST (specifically, the order of its elements does not respect BST semantics), it cannot be used directly for search. However, it can be jointly traversed with the help of one of the other trees. We describe this for the case where $L_1 < L_{mid}$ and $L_2 < L_{mid}$. A client starts by retrieving the tuples at level 0 and index 0 for both b_2 and b_3 in parallel. This is equivalent to lines 9–11 in Figure 4.4 (during the first iteration of the loop when $i = 0$). The result of these two separate calls (one for each tree) to the Extract procedure in line 11 of Figure 4.4 is the pair of $(label, c)$-tuples t_2 and t_3. While the label of t_3 is unintelligible (since it is encoded) and the label of t_2 is irrelevant to the client's search (since the client is not interested in an element in b_2), they can be combined to compute $(L, c) = t_1 = t_2 \oplus t_3$, which is the corresponding tuple in b_1.

This yields a way to jointly traverse the trees: the client can compare L_2 to L and choose whether to go left or right on both b_2 and b_3 for the next level. If $L_2 = L$, the client can save c (as this is the desired ciphertext), and continue with random

indices for the remaining levels. The above steps are analogous to lines 13–28 in Figure 4.4 when one replaces $L^\star$ with L_2.

4.4.4 Group communication

With the use of batch codes and the multi-query BST-retrieval algorithm described in the previous section, Pung allows a client to specify multiple labels (we discuss the details of this interface in Section 6.6) and get back multiple messages from Pung's server at an amortized computational cost. This can be used as a building block for group communication. In particular, suppose that a group of users G has privately derived a group shared key k_L. For now, assume this key derivation is done securely through an out-of-band channel; we discuss a more pragmatic alternative in the next section. A user $i \in G$ can use the group shared key to send its message to G under label $\text{PRF}_{k_L}(r \,||\, uid_i)$ during round r. Here, uid_i is the unique id of user $i \in G$ (§4.1). Furthermore, any user in G can simultaneously retrieve all messages sent during round r with the multi-query BST-retrieval algorithm. Specifically, the user passes in labels $\text{PRF}_{k_L}(r \,||\, uid_j)$ for all $j \in G$, which results in the user getting all the messages that were sent by group members during round r. This scheme provides metadata-privacy for group communication provided that all users in the group are honest, follow the protocol, and the group shared key is kept secret.

4.5 Managing contacts and starting conversations

This section addresses two important questions that we have avoided thus far: how exactly do users agree on a particular round to start communicating and how do pairs of users (or groups) derive a shared key? Once these parameters are established, clients exchange messages by deriving the appropriate labels (§4.1). In Pung, the answer depends on the type of pre-existing relationship that users have: *symmetric*, where users already know each other and have already derived a shared key, and *asymmetric*, where one user wishes to add a new contact. We describe both cases.

4.5.1 Managing symmetric connections with a control plane

Client applications of users who already know each other exchange *control messages* through Pung. Control messages have a special structure that client applications recognize and automatically act upon, so they are transparent to actual users (that is, the users never see these messages). Clients send control messages over Pung like any other message—so they too are private—and include statements like "`END`" to indicate that a conversation is over, or "`START [round]`" to indicate the round when a conversation should start. Clients send these messages periodically (for example, every 20 rounds), but can also send them during an active communication in response to events (for instance, `END` is sent when the application is placed in the background or when the user stops typing for a few minutes).

The frequency of control messages is configured the first time that two users communicate with each other, but it can be adjusted dynamically with the "`FREQ [rounds]`" control statement. Using a higher frequency leads to smoother operation (for example, client applications can agree on a round to start a conversation faster), but like any other message, they count toward the send and retrieve rate limit chosen by the user (§4.1).

Clients also use control messages to ensure message delivery by implementing a transport layer on top of Pung. In particular, when a client sends a message via Pung, the recipient's application sends an "`ACK`" to acknowledge that the message has been received and that the next message in the sender's send queue can be sent. In the absence of an acknowledgment, the sender continues to resend the same message until it reaches a predetermined maximum attempt number. At this point, the Pung client application displays a notification to the user stating that the message could not be delivered.

4.5.2 Managing symmetric connections with a dialing protocol

An alternative to periodically sending control messages is for clients to use a *dialing* protocol as proposed by Alpenhorn [147]. In Alpenhorn's dialing protocol, clients first derive *round keys* from the shared key and the current round, and use those

round keys to generate *dialing tokens*. The round key K_r is obtained by applying a PRF to the string "1" using the previous round key (or the shared key initially): for round r the round key is $K_r = PRF_{K_{r-1}}(\text{“1”})$. A dialing token for round r is then generated by applying a PRF to the string "2" using K_r: for round r the dialing token is $token = PRF_{K_r}(\text{“2”})$. (There is nothing special about "1" or "2", they just need to be different strings and they need to be well-known to all participants). Since both clients derive the same round key, they both generate the same dialing token. However, an adversary who does not know the round key cannot distinguish *token* from a random bitstring. Consequently, by looking at *token*, the adversary cannot determine if it is a real or a dummy dialing token, or the identity of the intended recipient of that token.

One way to think about dialing tokens is as Boolean *indicator labels*. That is, a label whose presence indicates interest in starting a conversation. Indeed, the way Alpenhorn generates dialing tokens is similar to the way Pung derives labels in Section 4.1. Pung can use Alpenhorn's dialing scheme as follows: Pung servers receive dialing tokens from clients and add them to a Bloom filter. Clients then download this Bloom filter and locally check, for each of their peers, if a corresponding dialing token is present. If so, the receiving client can send a "`START [round]`" control message to the peer, retransmitting if necessary until it receives an "`ACK`" or the target round is past. The submission of dialing tokens and the download of the Bloom filter happens in parallel with the rest of Pung (one can think of clients as interacting with a different system). It can happen, for example, every 10 minutes. Note that if clients are idle, they generate a random dialing token and send that instead.

4.5.3 Initiating asymmetric connections

The exchange of control messages and dialing, described above, presupposes an established relationship between clients. But how does Pung bootstrap this interaction in the first place? One option is for clients to use control messages to introduce their peers to others. This would allow clients to bootstrap connections, provided users had an honest shared contact. A more realistic alternative is for clients to use an *add-*

friend protocol, as proposed by Alpenhorn [147]. Alpenhorn's scheme uses a variant of *identity-based encryption* (IBE) [46] to allow clients to encrypt special control messages using easy-to-remember strings like email addresses. Unfortunately, it appears hard (and might not even be possible) to construct an IBE scheme under our threat model (where the adversary controls all servers). Consequently, Pung uses a simplified scheme that assumes clients know their peers' public keys rather than (the more user-friendly) email addresses (§3.2).

In Pung, each client has two well-known public keys: a *public verification key* that allows anyone to verify digital signatures generated by the client's secret signing key, and a *public encryption key* that allows anyone to encrypt a message that can only be decrypted by the client's secret decryption key. While in some cases both of these keys could be the same, this could lead to issues. One reason is that cryptosystems for digital signatures and public key encryption are designed and analyzed independently so reusing keys could open the door for attacks. Another reason is that one might want to change encryption keys often, but keep a long-term verification key (this long-term key can then be used to authenticate new public encryption keys). Indeed, we do this in Pung.

Specifically, a client in Pung only needs to get its peer's public verification key through an out-of-band channel; Pung has a key server that clients use to periodically deposit their most recently signed public encryption keys (which contain an explicit expiration date) under their name, email address, or any other string. Clients can then periodically (for instance, once a day) obtain these public encryption keys from Pung's servers using an instance of PIR with BST retrieval as we describe in Section 4.3.3 (where the associated label is a collision-resistant hash of the corresponding client's public verification key), adding chaff if a client has nothing to retrieve. Clients can check these keys' authenticity using the long-term well-known verification keys that they obtained out of band, preventing an adversary from forging and distributing fake public encryption keys.

Following Alpenhorn's protocol, a sender wishing to contact a recipient creates a *friend request*. Friend requests contain the sender's name or email address, a round number on which to start communicating, cryptographic material to de-

rive a shared key, and a signature of this information that can be verified with the sender's long-term verification key. This information is then encrypted using the recipient's public encryption key. Since in Chapter 3.2 we assume that the public key encryption scheme is key-private (which means that it is hard for an adversary to determine which public key was used to generate a particular ciphertext), the sender simply sends this encrypted friend request to a Pung server. If a sender does not wish to add a friend, the sender issues a fake request by encrypting random information with a dummy public key.

Just like in the dialing protocol above, recipients periodically check for friend requests by downloading *all* of the requests sent by all clients. One difference is that we cannot use a Bloom filter since the recipient does not know what he or she is looking for (by contrast, the recipient could derive the exact dialing token). Recipients instead attempt to decrypt each of the friend requests. Since in Section 3.2 we assume the public key encryption scheme is weakly robust[3] (that is, no honestly generated ciphertext decrypts to a valid message under two different keys), clients can determine exactly which requests are meant for them.

4.6 Leakage in the presence of compromised friends

Recall that Pung provides no privacy guarantees to any communication between a client and a friend who has been compromised by the adversary (§3). In this section we ask whether an adversary—by leveraging a compromised friend—can learn anything about a client's *other ongoing communications*. At first glance the answer appears to be no (assuming that the client does not voluntarily disclose the existence of any other communication to the compromised friend). After all, Pung's guarantee of relationship unobservability should prevent the adversary from learning about the existence of conversations between honest clients. This is indeed true, as we show in Appendix C, but only if Pung does not allow clients to start new conversations by us-

[3] We do not require *strong robustness* (meaning that a maliciously generated ciphertext does not decrypt to valid messages under two different keys) since clients are downloading and trying to decrypt *all* messages, and the adversary could instead simply send two different ciphertexts [140].

ing a dialing protocol or by agreeing securely out-of-band. Without this crippling assumption, we show that there are circumstances under which an adversary can learn information about a client's other ongoing communications by leveraging the client's compromised friends. Interestingly, we show that this leakage is fundamental and applies to *all* metadata-private messaging (MPM) systems that support dialing (or any mechanism that allows clients to start new conversations) [23, 145, 147, 213, 216]. In Section 4.6.5 we show how Pung can prevent this leakage at high cost.

4.6.1 The exclusive call center problem

We start by introducing an abstract problem that we call the *exclusive call center problem*. It consists of a call center that has k operators capable of receiving calls. The call center promises exclusivity to a single organization. This might be desirable to ensure high quality of service, for legal reasons, or to prevent the accidental leak of trade or business secrets to callers of a different organization. When a caller issues a call, an automatic *answering machine* M routes the call to an available operator. If M receives more calls than there are available operators, then M routes as many calls as it can, and notifies the remaining callers that all operators are busy.

While the above seems like a reasonable design, the call center in question is greedy and wishes to oversubscribe its resources by contracting with a second organization—thereby violating its exclusivity agreement. This poses two problems for the lying call center. First, M cannot determine to which organization a call belongs; only an operator is in a position of making that distinction. This places limits on how clever M can be. Second, with the current decision logic of M (route to available operators, notify remaining callers that operators are busy), either of the two organizations can easily determine that they are not being given exclusive access to the call center (for example, by placing k calls and noticing that not all are picked up). Given these issues and the limit of k operators (which is publicly known), can the call center do anything to maintain the illusion of exclusivity?

The first observation that the call center's CEO makes is that while there are k operators, there is no guarantee that all of them are available at any given point

in time. After all, operators are human and have the right to take breaks. This, the CEO believes, opens the door for some level of plausible deniability. In particular, if M gives a caller from organization O_1 a busy signal it could mean:

1. All k operators are busy handling calls of other callers from O_1.
2. Some operators are busy handling callers from O_1 and the remaining operators are on a break.
3. Some operators are busy handling callers from O_1, some are busy handling callers from O_2, and some are on a break.

Possibility 1 is the expected scenario of a high-efficiency trustworthy call center. Possibility 2 is an unwanted outcome since it is inefficient, but it does not violate the contractual agreement. Possibility 3, however, violates the promise of exclusivity. The goal of the lying call center is to design M in such a way that it is hard for either of the two organizations and their callers (that is, assume no collusion across organizations) to infer that possibility 3 is the one taking place. As we alluded to earlier, the key challenge is that M cannot distinguish between callers (and importantly cannot determine to which organization they belong), and therefore cannot selectively lie to keep a consistent set of responses. We thus ask whether there exists any M that can leverage the proposed ambiguity to fool the organizations into thinking they are exclusive. In other words, does there exist a *private answering machine M*?

We think of M as acting in rounds, where in each round, M receives a set of callers C and a number of operators k. We seek two properties from M.

- **Liveness**: eventually one of the callers in C gets to talk to an operator.
- **Privacy**: it is computationally hard for any colluding subset of callers $S \subseteq C$ (some of whom may get to speak to operators) to distinguish between a scenario where $S = C$ and a scenario where $S \subset C$ (in other words, it is difficult for the colluding subset of callers to determine whether they are the only callers or not).

The liveness guarantee is needed for M to be useful, but also to rule out a trivial solution: if M never puts anyone through to an operator, then the probability that any colluding set of callers S can distinguish between $S = C$ and $S \subset C$ is the same as randomly flipping a coin (assuming both scenarios are equally likely).

Security game. To define privacy and liveness more formally, we use a security game played between an adversary $\mathcal{A}$ and a challenger parameterized by a polynomial time answering machine M and a security parameter λ. M takes as input a subset of callers C from the set of all possible callers $\mathbb{C}$, a number of operators k, and a random string r, where $k = poly(\lambda), |\mathbb{C}| = poly(\lambda), |r| = poly(\lambda)$. M outputs a set of callers $U \subseteq C$, such that $|U| \leq k$.

1. $\mathcal{A}$ is given oracle access to M, and can issue a $poly(\lambda)$ number of queries to M with arbitrary inputs C, k, r. For each query, $\mathcal{A}$ can observe the corresponding result $U \leftarrow M(C, k, r)$.

2. Challenger samples a random bit b uniformly in $\{0, 1\}$, and a random string r uniformly in $\{0, 1\}^{\lambda}$.

3. $\mathcal{A}$ picks a set of callers S (where $S \subset \mathbb{C}$) and positive integer k, and sends them to the challenger.

4. Challenger sets $C = S$ if $b = 0$, and $C = S \cup \{e\}$ if $b = 1$ (where e is a uniform random element from the set $\mathbb{C} - S$).

5. Challenger calls $M(C, k, r)$ to obtain $U \subseteq C$ where $|U| \leq k$.

6. Finally, the challenger removes e from U (if it is present) and returns the result $(U - \{e\})$ to $\mathcal{A}$.

7. $\mathcal{A}$ outputs its guess b', and wins the security game if $b = b'$.

In summary, the adversary's goal in the above security game is to determine if the challenger is communicating with the uncompromised caller e after compromising all of the other callers (represented by S).

Definition 4.6.1 (Private answering machine)**.** An answering machine M is privacy-preserving if in the above security game with parameter λ, for all probabilistic poly-

nomial time algorithms $\mathcal{A}$, there exists a *negligible function*[4] negl such that:

$$|\Pr[b = b'] - 1/2| \leq \text{negl}(\lambda)$$

the probability is over the random coins of M and the challenger.

Definition 4.6.2 (Non-trivial answering machine)**.** An answering machine M is non-trivial if given security parameter λ, for any set of callers C, number of operators k (where $k > 0$), and random string r, the probability that $M(C, k, r)$ outputs a non-empty set is non-negligible in λ. Here $|C| = poly(\lambda), k = poly(\lambda), |r| = poly(\lambda)$, and the probability is over the random coins of M.

4.6.2 Challenge with building private answering machines

We now give two straw man proposals to highlight why constructing a private answering machine that meets Definitions 4.6.1 and 4.6.2 is challenging.

Straw man M_1:

- **Input**: C, k, r
- $\pi \leftarrow$ uniform pseudorandom permutation of C according to r
- **Output**: the first $\min(k, |C|)$ elements from π

This is not secure. Let X be the random variable describing the cardinality of the set returned to $\mathcal{A}$, namely $|U - \{e\}|$. Assuming that $k \leq |C|$, $\Pr[X < k \,|\, b = 0] = 0$ and $\Pr[X < k \,|\, b = 1] = k/|C|$. As a result, $\mathcal{A}$ can, by simply counting the elements in $U - \{e\}$, distinguish between $b = 0$ and $b = 1$ with non-negligible advantage.

Straw man M_2:

- **Input**: C, k, r
- $\pi \leftarrow$ uniform pseudorandom permutation of C according to r

[4] A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if there exists an integer c such that for all positive polynomials $poly$ and all x greater than c, $|f(x)| < 1/poly(x)$.

- Sample $m \in [0, \min(k, |C|)]$ uniformly at random
- **Output**: the first m elements from π

This is also not secure. For simplicity let $k = 1$ and $|S| = 1$. We have that the probability that the challenger returns to the adversary an empty set is much higher when $b = 1$ than it is when $b = 0$. This is due to Line 6 in the security game and the way we construct M_2. Again, let X be the random variable describing the cardinality of the set returned to $\mathcal{A}$. In particular, $\Pr[X < 1 \mid b = 0] = 1/2$, whereas $\Pr[X < 1 \mid b = 1] = 3/4$. As a result, $\mathcal{A}$ can distinguish between $b = 0$ and $b = 1$ with non-negligible advantage. More generally, since X is drawn from a uniform distribution when $b = 0$, the probability mass function (pmf) for X (assuming $k \leq |C|$) is:

$$f(x) = \begin{cases} \frac{1}{k+1} & \text{for } 0 \leq x \leq k \\ 0 & \text{otherwise} \end{cases}$$

On the other hand, if $b = 1$, the pmf for X is:

$$f(x) = \begin{cases} \frac{1}{k+1} + \frac{1}{2(k+1)} & \text{for } x = 0 \\ \frac{1}{k+1} & \text{for } 1 \leq x \leq k-1 \\ \frac{1}{2(k+1)} & \text{for } x = k \\ 0 & \text{otherwise} \end{cases}$$

An adversary $\mathcal{A}$ can leverage the difference in these pmfs to distinguish between $b = 0$ and $b = 1$ with non-negligible advantage.

We also consider sampling m and permuting C non-uniformly, but the effect of Line 6 (in the security game) is large enough for $\mathcal{A}$'s advantage to remain non-negligible (recall that M must output a non-empty set with non-negligible probability to satisfy non-triviality). As a result, building an answering machine that meets both the privacy and liveness guarantees does not seem possible. However, below we give the construction of a private answering machine under a relaxed setting.

4.6.3 Answering machines with a known set of callers

We now discuss the construction of an answering machine that provides privacy and liveness under the assumption that the set of all possible callers ($\mathbb{C}$) is fixed and known in advance to M (the machine still does not know which callers belong to a particular organization). As a result, we assume that each element e in $\mathbb{C}$ can be uniquely mapped to an integer with the function $id(e)$, and that this mapping is known to M.

Private answering machine M_3:

- **Input**: C, k, r
- $U \leftarrow \varnothing$
- $\forall_{e \in C}$, $\forall_{0 \leq i < k}$, if $id(e) = (r + i) \mod |\mathbb{C}|$, add e to U.
- **Output**: U

In other words, M_3 precomputes a schedule mapping callers to rounds: in each round a set of k callers will be serviced (the input r is the current round). If a caller happens to call during a round that has been allocated for it, it will be added to the set U (its call will be handled). Otherwise, the call will not be answered.

Machine M_3 guarantees liveness because for every caller e, every k out of $|\mathbb{C}|$ rounds are assigned to e; since $|\mathbb{C}| = poly(\lambda)$, this occurs with non-negligible probability. Machine M_3 guarantees privacy because the response given to $\mathcal{A}$ at the end of the game (Step 6 in the security game) depends only on r and not on b. As a result this response is exactly the same when $b = 0$ and $b = 1$; observing this response gives no advantage to $\mathcal{A}$.

4.6.4 The compromised friend attack

The exclusive call center problem is the scenario encountered by users in MPM systems who communicate with compromised friends. Clients in these systems can only handle a fixed number of concurrent conversations in one round (this maps to the k operators in the call center problem), which opens the door to an attack that we call

the *compromised friend attack* (CFA). An adversary—via compromised friends—can dial (or start a conversation through any other means supported by the MPM system) a client and observe whether the client responds or not. If the client does not have a private answering machine, then the adversary will be able to distinguish between a scenario where the client is also talking to some honest client (meaning that the adversary's subset of callers is not the full set, $S \subset C$), and a scenario where the client is not talking to some honest client ($S = C$). This leaks one bit of information that opens the door to existing attacks.

Intersection, disclosure, and hitting set attacks. There is a large literature of traffic analysis attacks [19, 82, 84, 87, 88, 133, 134, 156, 178, 178, 190, 212] that uncover patterns of communication by observing when users send and receive messages. For example, intersection attacks [190] can be used to narrow down the possible recipients of a message when users communicate with a single friend, while disclosure [19] and hitting set [134] attacks can handle the case where users communicate with multiple friends. There are also statistical variants of these attacks [82].

MPM systems typically avoid these attacks by requiring users to always be online, continuously sending and retrieving messages at a particular rate; when users are not communicating, they send and retrieve dummy messages instead. Unfortunately, the CFA allows an adversary to guess whether a user is sending dummy messages or not with non-negligible advantage. An adversary can therefore target a set of potential senders and potential recipients with a CFA, making these systems vulnerable to disclosure attacks.

How effective is the CFA in practice. Of course, there are some challenges in composing CFA with disclosure or statistical disclosure attacks. First, if a user is communicating with f friends in a given round, the CFA leaks that the user is sending at least one real message, but, depending on the particular design of the MPM system, it might be hard to pinpoint the exact value of f. Second, the CFA requires actively targeting users on a given round, which may limit the number of observations that are available to an adversary. Furthermore, this attack requires compromising users'

actual friends or it requires the use of phishing attacks to fool users into befriending malicious users. This is because the only message that a user can send to a stranger in MPM systems is a friend request, and this operation is handled differently than all other messages (typically through a different service, as we discuss in Section 4.5.3).

Once an attacker has compromised a user's friends, a CFA is actually very effective on existing MPM systems. This is because existing systems use a deterministic answering machine M that automatically accepts calls until a client's communication rate (k) is reached, at which point M ignores further requests. Moreover, since M does not pre-empt existing conversations, an adversary who calls a user with k compromised friends learns with probability 1 whether the user is engaged in a real conversation or not. The adversary could even learn the exact number of people with whom a user communicates by performing a binary search over multiple rounds (specifically, calling the user with $k/2$ compromised friends, observing the result, and adjusting the number in future rounds).

Variants of the compromised friend attack. Note that this same attack can be achieved in other ways. For example, if a pair of users are already communicating with each other and they wish to increase the number of messages they exchange per round, this is the moral equivalent of dialing. In particular, if clients can sustain k concurrent conversations, one can think of each of these k "channels" as logical clients with communication rate of 1. Increasing the number of channels between two clients is tantamount to a dialing interaction between the users' logical clients (the mechanism to achieve this type of dialing would be the exchange of control messages as we describe in Section 4.5.1).

Another instance of this attack occurs when the round duration is relatively short. Suppose that a client is engaged in a conversation with several other users (some of whom are compromised). If the human user is not quick enough to process and respond to all incoming messages during a given communication round, some of the user's friends will not receive a response. In other words, one can think of the human user's processing capacity as the k operators. If the client does not have a private answering machine that decides which of all incoming messages the client's

screen displays to the human user in a given round, an adversary could use the presence or absence of a response as a weak signal that the user is busy communicating with others.

4.6.5 Defending against compromised friends

We now discuss a couple of ways in which existing MPM systems could prevent compromised friend attacks. Assuming a secure answering machine M (for instance, M_3 in Section 4.6.3), each client could use M to determine which of the new (or existing) conversations to accept (or continue), and this decision would leak no information. As is the case in existing MPM systems, the client would continue to send and retrieve k messages every round, but only a subset of these messages (based on the output of M) correspond to actual conversations; the rest act as cover traffic. Note that if one uses M_3, a compromised friend can learn how many other friends a user has, or at least an upper bound on it (which corresponds to $|\mathbb{C}|$ in the abstract scenario). Furthermore, while M_3 gives a sliding window of communication to allow two clients to exchange messages for k consecutive rounds, clients must wait, possibly for a while, to be able to communicate again. Finally, it is possible that two clients who are using M_3 have no overlapping rounds and therefore cannot communicate with each other (in Pung this is not an issue because clients can retrieve messages sent to them in previous rounds). Additional mechanisms would therefore be needed to avoid this case for other MPM systems.

In the absence of a private answering machine M, clients could set k to a value larger than their maximum number of friends (assuming that each pair of friends exchanges at most one message per round). This is actually the solution that we use in Pung. In our evaluation (§8.4.3) we experiment with clients setting k up to 256. Furthermore, the frequency of rounds could be reduced to lower the number of samples that a compromised friend collects, and to mitigate the case where a user is too slow to respond to all messages. The drawback of these defenses is that making rounds less frequent increases message latency, and the communication and computational costs of MPM messaging systems increase linearly with k (computational costs in

Pung increase only sublinearly due to the use of batch codes).

4.7 Summary

The previous sections detailed Pung's architecture (§4), how clients send (§4.2) and privately retrieve messages (§4.3), even with groups (§4.4). They also discuss how clients can bootstrap their communication with mechanisms to add friends (§4.5.3), synchronize their rounds (§4.5.1), and dial them (§4.5.2), while preventing compromised friends from learning about other conversations (§4.6). These procedures are sufficient to build a version of Pung that meets our security goals (§3.1): it enables users to communicate with each other privately, preserving the integrity of messages, and hiding the content and metadata. Furthermore, none of the security guarantees depend on the correctness of Pung's servers (or other users). For instance, if Pung's server modifies the ciphertext associated with any tuple, clients can detect this due to the integrity guarantees of the authenticated encryption scheme. If the server drops tuples or stores them in a data structure that is not a complete BST, clients will be unable to find the tuple of interest to them (a denial of service), but the integrity of the content and the privacy of the communication is preserved.

The drawback with Pung's design as described so far is its costs: the server has to process over the entire collection to answer each PIR query, and each query is large (tens of MBs). Additionally, for applications where clients wish to retrieve more than one message in the same round (for example, email, group communication, file-sharing), using batch codes reduces computational costs but also leads to prohibitive network overhead. We address these issues in the coming chapters. Chapter 5 introduces a technique to reduce the network cost of PIR by orders of magnitude, while Chapter 6 relaxes batch codes and proposes a scheme to simultaneously achieve better computational amortization and orders of magnitude lower network costs.

Chapter 5

Reducing network communication with SealPIR

Besides being useful in Pung, PIR is a crucial building block in many other applications, including anonymous communication [146, 164], and privacy-preserving variants of media streaming [20, 122], ad delivery [115], friend discovery [47], and subscriptions [68]. Unfortunately, the costs of CPIR constructions [20, 51, 55, 97, 137, 143, 153, 207] are so significant that existing CPIR-backed systems must settle with supporting small databases with fewer than 100,000 entries [20, 115, 122]. If one were to use any of these constructions in Pung, it would be prohibitive to support more than a few tens of thousands of users.

In this chapter we discuss SealPIR, a new CPIR library that extends the most computationally-efficient CPIR protocol, XPIR [20], with a new query compression technique that reduces network costs. The main takeaway is that SealPIR introduces small computational costs (6%) to the server over XPIR, but simultaneously reduces computational costs for the client (by up to 17×) and the size of PIR queries (by up to 274×). This trade-off is an excellent fit for Pung, since clients could often be weak devices with limited bandwidth.

This chapter contains material from a previously published work: "PIR with compressed queries and amortized query processing" (S&P '18) by Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty [25]. Sebastian contributed to the design, implementation, and experimental evaluation of the techniques described in this chapter.

5.1 Background: Stern's PIR protocol

Our work extends Stern's PIR protocol [207] and incorporates the optimizations proposed in XPIR [20]. At a high level, a query in Stern's PIR protocol consists of a vector of n ciphertexts, where n is the number of elements in the server's database. All of these ciphertexts encrypt the integer 0 (the cryptosystem is randomized so all of the ciphertexts are different), except for the ciphertext at position *idx* (the index of the element desired by the client) which encrypts the integer 1. Clients generate these ciphertexts using an *additively homomorphic cryptosystem* (a common choice is Paillier [175]). In this type of cryptosystem, it is possible for anyone—even someone without access to the client's secret decryption key (for example, the PIR server)—to perform an operation directly on ciphertexts that results in a new ciphertext encrypting the sum of the corresponding plaintexts. For example, given two ciphertexts[1] encrypting a and b, $c_1 = Enc(a)$ and $c_2 = Enc(b)$, anyone can compute $c_3 = c_1 \odot c_2 = Enc(a + b)$ by performing $\odot$ (in Paillier, $\odot$ is a modular multiplication). Similarly, anyone can compute the multiplication of a ciphertext by a plaintext. For example, given c_1 and the integer 3 (acting as a plaintext), one can compute $c_4 = 3 \cdot c_1 = c_1 \odot c_1 \odot c_1 = Enc(3a)$.

Figure 5.1 gives a sketch of Stern's protocol. The server receives the query vector from the client and executes Answer by computing a dot product between its database (which can be thought of as a vector of n plaintexts) and the query (vector of n ciphertexts). This is a two step process. First, the server computes entry-wise plaintext-ciphertext multiplications (like c_4 in the example above, where the database element, which is some arbitrary binary data, is treated as the corresponding integer) to produce n output ciphertexts. Notice that all but one ciphertext encrypt 0, and multiplying any element (that is, a plaintext) by an encryption of 0 yields another encryption of 0. For example, if $c_5 = Enc(0)$, then $123 \cdot c_5 = Enc(123 \cdot 0)$. Multiplying an element by an encryption of 1 yields an encryption of the element itself. For

[1] For simplicity, we denote a ciphertext of x as $c = Enc(x)$. In reality, Enc is a randomized algorithm that depends on a cryptographic key pk and random coins R (which we omit), so $c = Enc_{pk}(x, R)$. By $c = c'$ we mean that c and c' encrypt the same value (for example x) under the same key, but the ciphertexts might be different due to different random coins.

```
1:  function Setup(DB)
2:      Represent DB in an amenable format (see [20, §3.2])
3:
4:  function Query(pk, idx, n)
5:      for i = 0 to n − 1 do
6:          c_i ← Enc(pk, i == idx? 1 : 0)
7:      return q ← {c_0, ..., c_{n−1}}
8:
9:  function Answer({c_0, ..., c_{n−1}} = q, DB)
10:     for i = 0 to n − 1 do
11:         a_i ← DB_i · c_i                    // plaintext-ciphertext multiplication
12:     return a ← a_0 ⊙ ... ⊙ a_{n−1}          // homomorphic addition
13:
14: function Extract(sk, a)
15:     return Dec(sk, a)
```

Figure 5.1: CPIR protocol from Stern [207] and XPIR [20] on a database DB of n elements. This protocol requires an *additively homomorphic* cryptosystem with algorithms (KeyGen, Enc, Dec), where (pk, sk) is the encryption and decryption key pair generated using KeyGen. We omit the details of all optimizations. The client runs the Query and Extract procedures, and the server runs the Setup and Answer procedures. Each element in DB is assumed to fit inside a single ciphertext. Otherwise, each element can be split into ℓ smaller chunks, and Lines 11 and 12 can be performed on each chunk individually; in this case Answer would return ℓ ciphertexts instead of one.

example, if $c_6 = Enc(1)$, then $123 \cdot c_6 = Enc(123 \cdot 1)$.

The second step is for the server to combine the resulting n ciphertexts from the prior step using the cryptosystem's additive homomorphic operator (like c_3 in the example above). Since all but one ciphertext encrypt 0, and the remaining ciphertext encrypts the chosen element, the final ciphertext encrypting their sum is just another encryption of the chosen element. The server then sends the resulting ciphertext as the PIR response to the client who uses its private decryption key to obtain the plaintext of the requested element.

One obvious issue with the above scheme is that the query is as large as the database (both have n entries). As a result, the communication costs of this scheme are too high (in fact, it would not even constitute a PIR scheme since PIR requires sublinear communication [70]).

Fortunately, Stern [207] shows that it is possible to reduce the number of ciphertexts in the query to $d\sqrt[d]{n}$ for any positive integer d, thereby making network costs sublinear in n. The downside of Stern's approach is that the size of the response—which depends on the size of the elements rather than the number of elements (n)—increases exponentially with d. We explain this below.

5.1.1 Achieving sublinear communication costs

Stern [207], based on the technique of Kushilevitz and Ostrovsky [143], proposes a modification to the protocol in Figure 5.1. Instead of structuring the database DB as an n-entry vector (where each entry is an element), the server structures the database as a $\sqrt{n} \times \sqrt{n}$ matrix M: each cell in M is a different element in DB. The client then sends 2 query vectors, v_{row} and v_{col}, each of size $\sqrt{n}$. The vector v_{row} has the encryption of 1 at position r, while v_{col} has the encryption of 1 at position c (where $M[r, c]$ is the client's desired element). The server, upon receiving v_{row} and v_{col}, computes the following matrix-vector product: $A_c = M \cdot v_{col}$, where each multiplication is between a plaintext and a ciphertext, and additions are on ciphertexts. Observe that A_c is a vector containing the encryptions of the entries in column c of M.

The server then performs a similar step using A_c and v_{row}. There is, however,

one technical challenge: the underlying cryptosystem used by Stern's protocol is randomized to guarantee semantic security [112]. As a result, every plaintext maps to many possible ciphertexts, which requires the size of a ciphertext to be larger than the size of the plaintext that it encrypts. This ciphertext-to-plaintext size ratio is called the cryptosystem's *expansion factor*, or F. Since each entry in A_c is a ciphertext, it is too big to fit inside another ciphertext (the largest plaintext that can fit in a ciphertext has size $|ciphertext|/F$).

To address the above expansion issue, the server splits elements in A_c into F chunks, where the first chunk contains the first $|ciphertext|/F$ bits of the element, the second chunk contains the next $|ciphertext|/F$ bits, etc. A_c can be therefore be thought of as a matrix of $\sqrt{n}$ rows and F columns. In this matrix, row i and column j contains $Enc(M[i,c])_j$, which is the j^{th} chunk of $Enc(M[i,c])$. The server can now repeat the process as before on the transpose of this matrix: it computes $A_c^T \cdot v_{row}$, to yield a vector of F ciphertexts (where the j^{th} ciphertext is $Enc(Enc(M[r,c])_j)$), which it sends to the client. The client then decrypts all F ciphertexts and combines them to obtain $Enc(M[r,c])$. Finally, the client decrypts $Enc(M[r,c])$ to obtain $M[r,c]$—the desired element in DB.

This scheme generalizes by structuring the database as a d-dimensional hypercube[2] and having the client send d query vectors of size $\sqrt[d]{n}$. The server then returns F^{d-1} ciphertexts as the response.

5.2 Background: XPIR

A major issue with Stern's protocol [207] is that the homomorphic operations that the server must perform (plaintext-ciphertext multiplication and ciphertext addition) are computationally expensive. XPIR [20] is a recent construction that reduces the computational costs of Stern's scheme. The key idea in XPIR is to perform the encryption and homomorphic operations using a lattice-based cryptosystem (the authors use BV [52]), and preprocess the database in a way that further reduces the cost of the operations in Lines 11 and 12 in Figure 5.1. Using a lattice-based cryptosys-

[2]The server could instead use a hyperrectangle; the client would send d vectors of different sizes.

tem helps because ciphertext addition requires only modular addition of vectors of small integers (60 bits) which can be done efficiently in existing CPUs. In contrast, a scheme like Paillier [175] requires modular multiplications over very large integers, which is computationally expensive.

Multiplying a ciphertext by a plaintext is expensive in both schemes. In Paillier, this requires a modular exponentiation over very large integers. In the lattice-based cryptosystem used by XPIR, this requires polynomial multiplications. However, XPIR introduces preprocessing techniques that significantly reduce the cost of this operation [20, §3]. To our knowledge, this makes XPIR the only CPIR scheme that is usable in practice.

A major drawback of XPIR, which it inherits from Stern's protocol but which it also exacerbates, is high network costs. This comes from two sources. First, ciphertexts in lattice-based cryptosystems are very large, around 32 KB each (we explain this in Section 5.4). Second, recall that Stern describes a way to represent the query using $d\sqrt[d]{n}$ ciphertexts (instead of n) for any positive integer d (§5.1.1). This increases the response size exponentially from 1 to F^{d-1} ciphertexts. For Paillier, the expansion factor is $F = 2$, whereas for the cryptosystem that XPIR uses, $F \geq 6.4$ for recommended security parameters [22, 65]. As a result, even with Stern's technique, XPIR results in either the query vector or the response containing hundreds or thousands of very large ciphertexts for the values of n that we evaluate (§8.2.1).

5.3 SealPIR's objective

At a high level, our goal is to realize the following picture: the client sends one ciphertext containing an encryption of its desired index i to the server, and the server inexpensively evaluates a function called Expand that outputs n ciphertexts containing an encryption of 0 or 1 (where the i^{th} ciphertext encrypts 1 and others encrypt 0). The server then uses these n ciphertexts as a query and execute Stern's protocol as before (Figure 5.1, Line 9).

A straw man approach to construct Expand is to create a circuit that computes the following function: "if the index encrypted by the client is i return 1, else

return 0". The server then evaluates this circuit on the client's ciphertext using a *fully homomorphic encryption* (FHE) scheme (for example, BV [52], BGV [50], FV [100]) passing in values of $i \in [0, n-1]$ to obtain the n ciphertexts. An FHE scheme supports both addition and multiplication of ciphertexts, which is sufficient to evaluate arbitrary circuits, like the one proposed by our straw man, directly on ciphertexts.

However, evaluating the above circuit using FHE to compute Expand is impractical. First, the client must send $\log(n)$ ciphertexts as the query (one for each bit of its index). Second, the circuit is concretely large (thousands of gates) and expensive to evaluate. Third, the security parameters for the FHE scheme would have to be large, so Expand and the rest of the PIR protocol would be very costly. Finally, the server must evaluate this circuit for each of the n possible indices.

Instead, we propose a new algorithm to implement Expand. It relies on the types of cryptosystems that typically support FHE, but perhaps surprisingly, it does not require encrypting each bit of the index individually, or performing any homomorphic multiplications. This last point is critical for performance, since homomorphic multiplications are expensive and require using larger security parameters. We note that the cryptosystem used by XPIR (BV [52]) supports FHE, so we could implement Expand using that. We choose instead to implement all of SealPIR using the SEAL homomorphic library [11], which is based on the Fan-Vercauteren (FV) [100] cryptosystem. We make this choice for pragmatic reasons: Expand requires the implementation of a new homomorphic operation, and SEAL already implements the necessary building blocks. Below we give some background on FV.

5.4 Background: Fan-Vercauteren FHE cryptosystem (FV)

In FV, plaintexts are polynomials of degree at most N with integer coefficients modulo t. Specifically, the polynomials are from the quotient ring:

$$R_t = \mathbb{Z}_t[x]/(x^N + 1)$$

operation	CPU cost (ms)	noise growth
addition	0.002	additive
plaintext multiplication	0.141	multiplicative*
multiplication	1.514	multiplicative
substitution	0.279	additive

Figure 5.2: Cost of operations in SEAL [11]. The parameters used are given in Section 8.1. Every operation increases the *noise* in a ciphertext (see text for details). Once the noise passes a threshold, the ciphertext cannot be decrypted. For a given computation, parameters must be chosen to accommodate the expected noise.
*While plaintext multiplication yields a multiplicative increase in the noise, the factor is always 1 (that is, no noise growth) in EXPAND because it is based on the number of non-zero coefficients in the plaintext (see the SEAL manual for details [65, §6.2]).

where N is a power of 2, and t is the *plaintext modulus* that determines how much data can be packed into a single FV plaintext. In our implementation, an FV plaintext is represented as an array of N 64-bit integers, where each integer is mod t (and $t < 2^{64}$). Each element in the array represents a coefficient of the corresponding polynomial. We encode an element e (for example a message in Pung) into an FV plaintexts $p(x)$ by storing $\log(t)$ bits of e into each coefficient of $p(x)$. If elements are small, we store many elements into a single FV plaintext (for example, the first element is stored in the first 20 coefficients, etc.). If elements are too big, we store the element across multiple FV plaintexts.

Ciphertexts in FV consist of two polynomials, each of which is in the ring:

$$R_q = \mathbb{Z}_q[x]/(x^N + 1)$$

where q is the *coefficient modulus* that affects how much *noise* a ciphertext can contain and the security of the cryptosystem. When a plaintext is encrypted, the corresponding ciphertext contains noise. As operations such as addition or multiplication are performed, the noise of the output ciphertext grows based on the noise of the operands and the operation being performed (Figure 5.2 gives the noise growth of several operations). Once the noise passes a threshold, the ciphertext cannot be

decrypted anymore.

Parameters. FV has 3 tunable parameters: N, t, and q. Since they present tradeoffs, finding the best combination depends on the use case. A larger polynomial degree N increases security but also increases the computational expense of homomorphic operations and the size of the ciphertext. A larger plaintext modulus t allows more data to be packed into a single FV plaintext (so one needs fewer FV plaintexts to represent, say, a movie), but it also increases the noise growth. A larger coefficient modulus q leads to a higher noise threshold (so one can perform more homomorphic operations), but it increases the size of a ciphertext and results in lower security [65]. The expansion factor of the FV cryptosystem is $F = 2\log(q)/\log(t)$. We discuss concrete parameters in Section 8.1.

Supported operations. In addition to the standard operations of a cryptosystem (key generation, encryption, decryption), FV also supports homomorphic addition, multiplication, and relinearization (which is performed after multiplications to keep the number of polynomials that make up a ciphertext at two); we are interested in the following operations.

- **Addition**: Given ciphertexts c_1 and c_2, which encrypt FV plaintexts $p_1(x), p_2(x)$, the operation $c_1 + c_2$ results in a ciphertext that encrypts their sum, $p_1(x) + p_2(x)$.
- **Plaintext multiplication**: Given a ciphertext c that encrypts $p_1(x)$, and given a plaintext $p_2(x)$, the operation $p_2(x) \cdot c$ results in a ciphertext that encrypts their product, $p_1(x) \cdot p_2(x)$.
- **Substitution**: Given a ciphertext c that encrypts plaintext $p(x)$ and an odd integer k, the operation $\mathsf{Sub}(c, k)$ returns an encryption of $p(x^k)$. For instance given plaintext $p(x) = 7 + x^2 + 2x^3 \pmod{x^N + 1}$, if c is an encryption of $p(x)$, then $\mathsf{Sub}(c, 3)$ returns an encryption of $p(x^3) = 7 + (x^3)^2 + 2(x^3)^3 = 7 + x^6 + 2x^9 \pmod{x^N + 1}$.

Our implementation of the substitution operator is based on the plaintext slot permutation technique discussed by Gentry et al. [108, §4.2]. However, substi-

tution is less general than Gentry et al.'s technique which can arbitrarily permute the coefficients of a plaintext polynomial, but at great expense. By giving up generality, we can implement substitution very efficiently, as shown in the last row of Figure 5.2. We give a detailed description of substitution in Appendix A.1.

5.5 Encoding the index

A client who wishes to retrieve the i^{th} element from the server's database using SealPIR generates an FV plaintext that encodes this index. The client does so by representing $i \in [0, n-1]$ as the monomial $x^i \in R_t$ (that is, the coefficient associated with x^i is 1 and all others are 0). In other words, instead of the client creating the vector of 0s and a 1 and placing each entry in a different ciphertext as in XPIR, the client in SealPIR represents this vector *in the coefficients of the plaintext polynomial*. The client then encrypts this plaintext to obtain $query = Enc(x^i)$, which is the query that the client sends to the server. In Section 5.8 we discuss how to query databases that are larger than the polynomial's degree (N); in these cases the index cannot be represented by a single FV plaintext.

5.6 Expanding queries obliviously

The server receives from the client $query = Enc(x^i)$, and uses a function called Expand to obtain a vector of n ciphertexts where the i^{th} ciphertext is $Enc(1)$ and all other are $Enc(0)$. To get a sense for how Expand works, we first give a description for the case where the database has only two elements ($n = 2$).

The server receives $query = Enc(x^i)$, with $i \in \{0, 1\}$ in this case (since $n = 2$) as the client's desired index. The server first expands *query* into two ciphertexts $c_0 =$

query and $c_1 = query \cdot x^{-1}$:

$$c_0 = \begin{cases} Enc(1) & \text{if } i = 0 \\ Enc(x) & \text{if } i = 1 \end{cases}$$

$$c_1 = \begin{cases} Enc(x^i \cdot x^{-1}) = Enc(x^{-1}) & \text{if } i = 0 \\ Enc(x^i \cdot x^{-1}) = Enc(1) & \text{if } i = 1 \end{cases}$$

The server then computes $c'_j = c_j + \mathsf{Sub}(c_j, N+1)$ for $j \in \{0, 1\}$. Since operations in R_t are defined modulo $x^N + 1$, a substitution with $N + 1$ transforms the plaintext encrypted by c_0 and c_1 from $p(x)$ to $p(-x)$.[3] Specifically, we have:

$$c'_0 = \begin{cases} Enc(1) + Enc(1) = Enc(2) & \text{if } i = 0 \\ Enc(x) + Enc(-x) = Enc(0) & \text{if } i = 1 \end{cases}$$

$$c'_1 = \begin{cases} Enc(x^{-1}) + Enc(-x^{-1}) = Enc(0) & \text{if } i = 0 \\ Enc(1) + Enc(1) = Enc(2) & \text{if } i = 1 \end{cases}$$

Finally, assuming t is odd, we can compute the multiplicative inverse of 2 in $\mathbb{Z}_t$, say α, encode it as the monomial $\alpha \in R_t$ (that is, the coefficient of the constant term is α and all other coefficients are 0), and compute $o_j = \alpha \cdot c'_j$. It is the case that o_0 and o_1 contain the desired output of Expand: o_i encrypts 1, and o_{1-i} encrypts 0.

We can generalize this approach for any database size (n) that is a power of 2 as long as $n \leq N$. In cases where n is not a power of 2, we can run the algorithm for the next power of 2, and take the first n output ciphertexts as the client's query. Section 5.8 discusses how to handle databases that are larger than N. Figure 5.3 gives the generalized algorithm, and Figure 5.4 depicts an example for a database of 4 elements. We prove the correctness of Expand in Appendix A.2 and bound its noise growth in Appendix A.3.

[3] Observe that $x^N + 1 \equiv 0 \pmod{x^N + 1}$, $x^N \equiv -1 \pmod{x^N + 1}$, and $x^{N+1} \equiv -x \pmod{x^N + 1}$.

1: **function** EXPAND($query = Enc(x^i)$)
2: find smallest $m = 2^\ell$ such that $m \geq n$
3: $ciphertexts \leftarrow [\, query \,]$
4:
5: // each outer loop iteration doubles the number of ciphertexts
6: // and only one ciphertext ever encrypts a non-zero polynomial
7: **for** $j = 0$ to $\ell - 1$ **do**
8: **for** $k = 0$ to $2^j - 1$ **do**
9: $c_0 \leftarrow ciphertexts[k]$
10: $c_1 \leftarrow c_0 \cdot x^{-2^j}$
11: $c'_k \leftarrow c_0 + \mathsf{Sub}(c_0, N/2^j + 1)$
12: $c'_{k+2^j} \leftarrow c_1 + \mathsf{Sub}(c_1, N/2^j + 1)$
13: $ciphertexts \leftarrow [c'_0, \ldots, c'_{2^{j+1}-1}]$
14:
15: // ciphertext at position i encrypts the monomial m and all other encrypt 0
16: $inverse \leftarrow m^{-1} \pmod{t}$
17: **for** $j = 0$ to $n - 1$ **do**
18: $o_j \leftarrow ciphertexts[j] \cdot inverse$ // ciphertext at position i now encrypts 1
19: **return** $output \leftarrow [o_0, \ldots, o_{n-1}]$

Figure 5.3: Procedure that expands a single ciphertext *query* that encodes an index i into a vector of n ciphertexts, where the i^{th} entry is an encryption of 1, and all other entries are encryptions of 0. We use the substitution group operator Sub (see text for details). Plaintexts are in the polynomial quotient ring $\mathbb{Z}_t[x]/(X^N + 1)$. N is a power of 2, n is the number of elements in the server's database, and $N \geq m \geq n$.

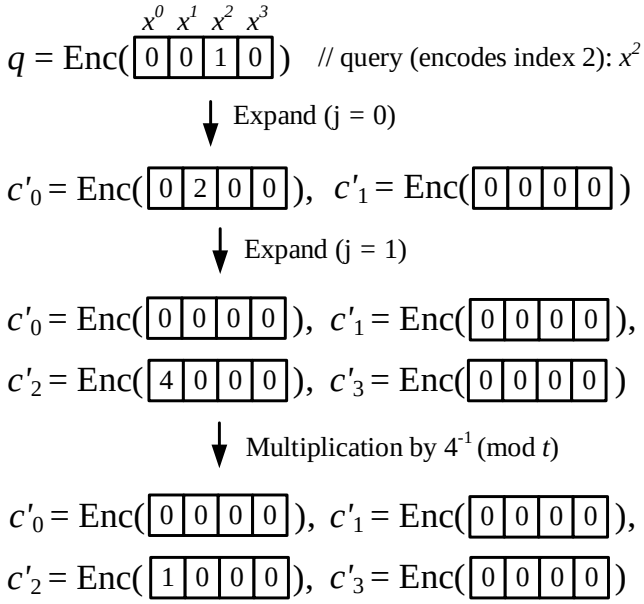


Figure 5.4: Example of EXPAND's effect on each iteration of the outer loop. Each array represents the coefficients of the corresponding polynomial. This example assumes a database with 4 elements ($n = 4$), polynomial degree $N = 4$, and a query retrieving the third item.

5.7 Reducing the cost of expansion

One issue with Expand is that despite each operation being inexpensive (Figure 5.2), $\mathcal{O}(n)$ operations are needed to extract the n-entry query vector. This is undesirable, since Expand could end up being almost as expensive to the server as computing the answer to a query (see Figure 5.1, Line 9). We show how to reduce this cost by having the client send multiple ciphertexts.

In Section 5.1.1 we describe how Stern's protocol [207] achieves communication costs that are sublinear in the size of the database. The key idea is to structure the database as a d-dimensional hypercube, and ask the client to send d query vectors instead of one. We show how to use this same technique to make the computational costs of Expand sublinear in the size of the database.

Instead of encoding one index, the client in SealPIR encodes d indices (on different ciphertexts), one for each dimension of the database. The server then calls Expand on each of the d ciphertexts and extracts a $\sqrt[d]{n}$-entry vector from each. The server then uses the modified PIR protocol described in Section 5.1.1 with the extracted d vectors. Observe that this reduces the CPU costs of Expand from $\mathcal{O}(n)$ to $\mathcal{O}(d\sqrt[d]{n})$. Of course, this approach has the downside that the PIR response gets larger because of the cryptosystem's expansion factor (F). Specifically, the network cost is d ciphertexts to encode the indices, and F^{d-1} ciphertexts to encode the response. The good news is that for small values of d (2 or 3), this results in major computational savings while still reducing network costs by orders of magnitude over XPIR (§8.2.1).

5.7.1 Optimizing Expand further

In FV, an encryption of $2^\ell \pmod{2^y}$, for $y \geq \ell$, is equivalent to an encryption of $1 \pmod{2^{y-\ell}}$. Observe that in Lines 16–18 of Figure 5.3, Expand multiplies the n ciphertexts by the inverse of m where $m = 2^\ell$ (the goal of this multiplication is to ensure that all ciphertexts encrypt either 0 or 1). Instead, we change the plaintext modulus of the n ciphertexts from $t = 2^y$ to $t' = 2^{y-\ell}$, which allows us to avoid the plaintext multiplications and the inversion, and reduces the noise growth of Expand. The result is $n - 1$ ciphertexts encrypting 0, and one ciphertext encrypting 1, as we

expect. This optimization requires t to be divisible by m rather than being an odd integer. One drawback is that the server must represent the database using FV plaintexts defined with the plaintext modulus t' (rather than t). As a result, we can pack fewer database elements into a single FV plaintext.

Note that the change of modulus is done by the client; the server simply encodes database elements into FV plaintexts with coefficients defined modulo t'. Specifically, the client generates the query as we discuss in Section 5.5 by generating an FV plaintext with coefficients modulo t. The server uses Expand without the multiplication by an inverse (Lines 16–18) to turn the client's query into a vector of ciphertexts where the corresponding plaintexts are defined with coefficients modulo t. In particular, the element at position i encrypts $m \in R_t$ and all other elements encrypt 0. The server then follows the PIR protocol (Figure 5.1) as before, which produces the PIR answer $Enc(m \cdot p(x)_i)$. Here, $p(x)_i \in R_{t'}$ is the FV plaintext requested by the client, and $m \cdot p(x)_i \in R_t$ is the polynomial that is actually encrypted by the PIR answer (since we did not perform the multiplication by the inverse of m, all coefficients in $p(x)_i$ are multiplied by a factor of m). When the client receives the answer from the server, it decrypts it into an FV plaintext with coefficients modulo t'. Observe that $m \pmod{t} \equiv 1 \pmod{t'}$, and therefore the PIR answer decrypts to $p(x)_i \in R_{t'}$, which is the desired outcome.

To select a value for t', we want the largest integer value of $\log(t')$ for which the following inequality holds:

$$\log(t') + \lceil \log(\lceil \sqrt[d]{n_{fv}} \rceil) \rceil \leq \log(t) \tag{5.1}$$

$$n_{fv} = \lceil n/\alpha \rceil$$

$$\alpha = \lfloor N \log(t')/\beta \rfloor$$

Here α is the number of elements of size β bits that can be packed into a single FV plaintext, and n_{fv} is the number of FV plaintexts that are needed to represent n elements of size β (in Pung's implementation, β = 2,304 bits).

The rationale behind Equation 5.1 is as follows. First, by definition $\log(t') \leq$

$\log(t)$, which places a bound on the value of t'. Second, t' should be as large as possible, since t' determines how much data the server can pack into a single FV plaintext. Finally, the second operand of Equation 5.1 corresponds to ℓ (and recall that $m = 2^\ell$). In other words, the second operand constrains how big of a database a single PIR query can index (since EXPAND outputs at most m ciphertexts). This operand is computed based on the number of FV plaintexts that make up the database, and the dimension of the database (§5.1.1). We discuss the empirical effects of this optimization (and the value of t') in our experimental evaluation (Sections 8.2.2 and 8.3).

5.8 Handling larger databases

As we discuss in Section 5.6, the size of the query vector that EXPAND can generate is bounded by N. Based on recommended security parameters [22, 65], N is typically 2048 or 4096 (larger N improves security but reduces performance). So how can one index into databases with more than N elements?

We propose two solutions. First, the client sends multiple ciphertexts and the server expands them and concatenates the results. For instance, if N is 2048, the database has 4000 elements, and the client wishes to get the element at index 2050, the client sends 2 ciphertexts: the first encrypts 0 and the second encrypts x^2. The server expands both ciphertexts into 2048-entry vectors and concatenates the results to get a 4096-entry vector where the entry at index 2050 encrypts 1, and all others encrypt 0. The server then uses the first 4000 entries as the query vector.

Another solution is to change the dimension of the d-dimensional hypercube that represents the database (§5.7). A dimension d allows the client to send d ciphertexts to index a database of size N^d. For $d = 3$ and $N = 2048$, three ciphertexts are sufficient to index 8.5 billion entries. One can also use a combination of these solutions. For example, given a database with 2^{24} entries, SealPIR uses $d = 2$ (so the database is a $2^{12} \times 2^{12}$ matrix), and represents the index for each dimension using $2^{12}/2048 = 2$ ciphertexts. The server expands these 2 ciphertexts and concatenates them to obtain a vector of 2^{12} entries. In total, this approach requires the client to send 4 ciphertexts as the query (2 per dimension), and receive $\lceil F \rceil = 7$ ciphertexts as

the response ($d = 3$ would lead to 1 ciphertext as the query, but $\lceil F^2 \rceil = 41$ ciphertexts as the response).

In conclusion, SealPIR's query compression and Expand procedures reduce the query communication complexity from $\mathcal{O}(Nd\sqrt[d]{n})$ in XPIR to $\mathcal{O}(Nd\lceil \sqrt[d]{n}/N \rceil)$. To see why this is the case, observe that XPIR sends d vectors containing $\sqrt[d]{n}$ ciphertexts each. Meanwhile, each ciphertext is made up of two polynomials of degree N. In contrast, SealPIR sends d ciphertexts, each of which is two polynomials of degree N (the ceiling operator is needed to account for the case where $\sqrt[d]{n} > N$, which requires more than one FV plaintext to encode the query).

5.9 Summary and future work

In this chapter we presented SealPIR, a new PIR library that balances computational and network costs in practice. SealPIR significantly reduces the network cost of XPIR, while introducing only modest computational overheads. The key technique is a new oblivious expansion procedure that allows a client to send a compressed query that the server can efficiently decompress and use in the rest of the XPIR protocol.

While the cost of query expansion is small, there are several opportunities to reduce CPU costs further. For example, observe that when the database dimension (d) is 2 (see Section 5.1.1) the first step of the computation consists of a matrix-vector product (the matrix is the database and the vector is v_{col} which specifies the desired column). The server could aggregate $\sqrt{n}$ column vectors (v_{col}) sent by different users into a $\sqrt{n} \times \sqrt{n}$ matrix, and compute a single matrix-matrix multiplication using an algorithm with subcubic computational complexity (for example, Strassen's [208]).

Chapter 6

Reducing computational costs with PBCs

Section 4.4 discusses how Pung's server can use batch codes to amortize its computational costs when processing a batch of PIR queries from the same client. Nevertheless, while existing batch code constructions achieve good computational amortization, they also introduce very high network overhead. In this chapter, we propose a new construction that weakens the guarantees of batch codes, and in return reduces network overhead by orders of magnitude. Before describing our construction, we first highlight the costs of using existing batch codes in PIR.

6.1 Costs of PIR with existing batch codes

Recall from Section 4.4.2 that a batch code that provides availability to sets (or multisets) of k input elements takes a collection of n elements as input, and outputs m codewords that are spread out (not necessarily evenly) across b buckets. Figure 6.1 depicts the relationship between the number of codewords (m) and the number of buckets b, as a function of the input collection size (n) and the batch size (k) for

This chapter contains material from two previously published works: "Unobservable communication over fully untrusted infrastructure" (OSDI '16) by Sebastian Angel and Srinath Setty [27], and "PIR with compressed queries and amortized query processing" (S&P '18) by Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty [25]. Sebastian contributed to all aspects of the design, implementation, and experimental evaluation of the probabilistic batch codes described in this chapter.

batch code	**codewords (m)**	**buckets (b)**
subcube ($\ell \geq 2$) [127, §3.2]	$n \cdot ((\ell+1)/\ell)^{\log_2(k)}$	$(\ell+1)^{\log_2(k)}$
combinatorial $\left(\binom{r}{k-1} \leq n/(k-1)\right)$ [177, §2.2]	$kn - (k-1) \cdot \binom{r}{k-1}$	r
Balbuena graphs [189, §IV.A]	$2(k^3 - k \cdot \lceil n/(k^3-k) \rceil)$	$2(k^3-k)$
3-way reverse cuckoo hashing* (**this work**, §6.5)	$3n$	$1.5k$

Figure 6.1: Cost of existing batch codes and the probabilistic batch code (PBC) construction given in Section 6.5. n indicates the number of elements in the database DB. k gives the number of elements that can be retrieved from DB by querying each bucket in $\beta(DB)$ at most once, where β is the batch code. Building a multi-query PIR scheme from any of the above constructions leads to computational costs to the server linear in m, and network communication linear in b. We list batch codes that have explicit constructions and can amortize CPU costs for multi-query PIR. Other batch codes have been proposed (for example, [154, 204, 205, 220]), but they either have no known constructions, or they seek additional properties (such as tolerate data erasures, or optimize for the case where $n = b$, support multisets) that introduce structure or costs that makes them a poor fit for multi-query PIR.
*Unlike other schemes, the reverse cuckoo hashing PBC can fail with small probability ($\approx 2^{-40}$ for large k).

several constructions. In multi-query PIR, the client issues one query to each of the b buckets and therefore receives b responses. To answer these b queries, the server computes over all m codewords exactly once; lower values of m lead to less computation, and lower values of b lead to lower network costs. Since $m < k \cdot n$, the total computation done by the server is lower than running k parallel instances of PIR. The drawback is that existing batch codes produce many buckets (see the third column in Figure 6.1). As a result, they *introduce* significant network overhead over not using a batch code at all. This makes batch codes unappealing in practice.

Our key observation is that batch codes' guarantees are actually too conservative for Pung. Specifically, batch codes guarantee *perfect completeness* (that is, clients can retrieve *any* k items). Meanwhile, Pung does not require that clients can *always* retrieve all k messages during a given round: since messages in Pung are long-lived (§7), if for some reason a client cannot get a particular set of k messages, the client can simply retry the next round. In fact, this behavior is inevitable in systems

resistant to traffic analysis such as Pung: recall that clients send and retrieve messages at some rate; any client who receives messages in excess of this rate must wait at least two rounds to get all the messages. Below we describe an alternative that works well for larger k, but is probabilistic. That is, a client can sometimes only retrieve a subset of the k messages that it wished to retrieve in a given round.

6.2 Probabilistic batch codes (PBC)

A *probabilistic batch code* (PBC) differs from a traditional batch code in that it fails to be *complete* with probability p. That is, there might be no way to recover a specific set of k elements from a collection encoded with a PBC by retrieving exactly one codeword from each bucket. The probability of encountering one such set (when the elements are uniformly chosen) is p. In the example of Section 4.4.2, this would mean that under a PBC, a client may be unable to retrieve both x_1 and x_2 by querying buckets at most once (whereas a traditional batch code guarantees that this is always possible). In practice, this is seldom an issue: our construction has parameters that result in roughly 1 in a trillion queries failing, which we think is a sufficiently rare occurrence. Furthermore, this is an easy failure case to address in PIR since a client learns whether or not it can get all of the elements before issuing any queries.

In addition to the above relaxation, the specific PBC construction described in this chapter introduces another weakening (this is not fundamental to PBCs). Our construction provides availability (§4.4.2) only to sets of elements in the input collection (not to multisets). In other words, our construction only allows for the retrieval of distinct elements. We emphasize that this is not a limitation for us. While multisets are common in non-PIR applications of batch codes (for example, distributed storage, network switches), this is not the case for multi-query PIR: if a client truly wishes to retrieve multiple copies of the same element, this can be trivially done without multiset support. Specifically, the client can download the element once, and then locally make as many copies as needed.

Definition 6.2.1 (PBC). A (n, m, k, b, p)-PBC is given by three polynomial-time algorithms (Encode, GenSchedule, Decode):

- $(C_0, \ldots, C_{b-1}) \leftarrow \mathsf{Encode}(DB)$: Given an n-element collection DB, output a b-tuple of buckets, where $b \geq k$, each bucket contains zero or more codewords, and the total number of codewords across all buckets is $m = \sum_{i=0}^{b-1} |C_i| \geq n$.

- $\{\sigma, \perp\} \leftarrow \mathsf{GenSchedule}(L)$: Given a set of k labels L corresponding to tuples in DB, output a *schedule* $\sigma : L \rightarrow \{\{0, \ldots, b-1\}^+\}^k$. The schedule σ gives, for each label $\ell \in L$, the index of one or more buckets from which to retrieve a codeword that can be used to reconstruct element $DB[\ell]$. GenSchedule outputs $\perp$ if it cannot produce a schedule where each $\ell \in L$ is associated with at least one bucket, and where no bucket is used more than once. This failure event occurs with probability p.

- *element* $\leftarrow \mathsf{Decode}(W)$: Given a set of codewords W, output the corresponding element $\in DB$.

In the sections ahead we describe an efficient PBC construction. Our key idea is as follows. Batch codes spread out elements such that retrieval requests are load balanced among different buckets. Relatedly, many data structures and networking applications use different variants of hashing—consistent [128], asymmetric [217], weighted [211], multi-choice [30, 166], cuckoo [29, 174], and others [53, 94]—to achieve a similar goal. While there is no obvious way to use these hashing schemes to implement multi-query PIR directly, we can do it indirectly: we first build a PBC from a simple technique that we call *reverse hashing* (§6.4), and then use the PBC to implement multi-query PIR (§6.6).

6.3 Randomized load balancing

A common use case for (non-cryptographic) hash functions is to build data structures such as hash tables or dictionaries. In a hash table, the insert procedure consists of computing one or more hash functions on the label of the item being inserted. Each application of a hash function returns an index into an array of buckets in the hash table. The item is then placed into one of these buckets following an allocation algorithm. For example, in multi-choice hashing [30, 166], the item is placed

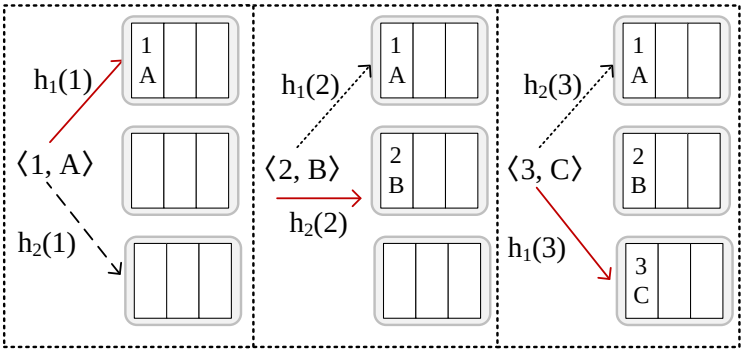


Figure 6.2: Logic for two-choice hashing [30] when allocating three key-value tuples to buckets: $\langle 1, A\rangle, \langle 2, B\rangle, \langle 3, C\rangle$. Tuples are inserted into the bucket least full. Arrows represent the choices for each tuple based on different hashes of the tuple's key (here we depict an optimistic scenario). The red solid arrow indicates the chosen mapping.

in the bucket least full among several candidate buckets. In Cuckoo hashing [174], items may move around following the Cuckoo hashing algorithm (we explain this algorithm in Section 6.5).

An ideal allocation results in items being assigned to buckets such that all buckets have roughly the same number of items (since this lowers the cost of lookup). In practice, there is load imbalance where some buckets end up having more elements than others; the extent of the imbalance depends on the allocation algorithm and the random choices that it makes. To look up an item by its label, one computes the different hash functions on the label to obtain the list of buckets in which the item could have been placed. One then scans each of those buckets for the desired item. An example of the insertion process for two-choice hashing is given in Figure 6.2.

Abstract problem: balls and bins. In the above example, hashing is used to solve an instance of the classic n balls and b bins problem, which arises during insertion. The items to be inserted into a hash table are the n balls, and the buckets in the hash table are the b bins; using w hash functions to hash a label to w candidate buckets approximates an independent and uniform random assignment of a ball to w bins. The number of collisions in a bucket is the load of a bin, and the highest load across all bins is the *max load*. In the worst case, the max load is n/w (all balls map to the

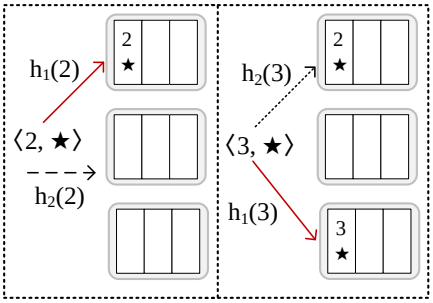
same w candidate buckets), but there are much smaller bounds that hold with high probability [30].

Interestingly, if we examine other scenarios abstracted by the balls and bins problem, a pattern becomes clear: the allocation algorithm is typically executed during data placement. In the hash table example, the allocation algorithm determines where to insert an element. In the context of a transport protocol [135], the allocation algorithm dictates on which path to send a packet. In the context of a job scheduler [173], the allocation algorithm selects the server on which to run a task. The result is that the load balancing effect is achieved at the time of "data placement". However, to build a PBC, we must do it at the time of "data retrieval". Reverse hashing achieves this.

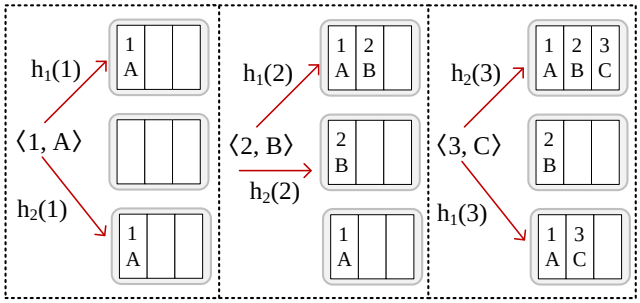
6.4 Reverse hashing

We start by introducing two principals: the *producer* and the *consumer*. The producer holds a collection of n items where each item is a key-value tuple. It is in charge of data placement: taking each of the n elements and placing them into buckets based on their label (the key in the key-value tuple) following some allocation algorithm. The consumer holds a set of k labels ($k \leq n$), and is in charge of data retrieval: it fetches items by their labels from the buckets that were populated by the producer. The goal is for the consumer to get all k items by probing each bucket as few times as possible. In other words, the consumer has an instance of a k balls and b bins problem, and its goal is to reduce the instance's max load.

Note that the consumer is not inserting elements into buckets (that is the job of the producer). Instead, the consumer is in some sense placing "retrieval requests" into the buckets. The challenge is that any clever allocation chosen by the consumer must be *compatible* with the actions of the producer (who populates the buckets). That is, if the consumer, after running its allocation algorithm (for example, multi-choice hashing) decides to retrieve items x_1, x_2, and x_3, from buckets 2, 3, and 7, it better be the case that the producer previously placed those elements in those exact buckets. We describe how we guarantee compatibility below.



(a) consumer's simulation



(b) producer's allocation

Figure 6.3: Example of two-choice reverse hashing. (a) shows the consumer's simulation when inserting two tuples $\langle 2, \star \rangle, \langle 3, \star \rangle$. The $\star$ indicates that the value is not known, so an arbitrary value is used. (b) shows a modification to two-choice hashing where the producer stores the tuple in all possible choices. This ensures that the final allocation is always compatible with the consumer's simulation.

Protocol. The consumer starts by imagining that it is a producer with a collection of k elements. In particular, the consumer converts its k labels into k key-value tuples by assigning a dummy value to each tuple (since it does not know actual values). In this simulation, the consumer follows a specific allocation algorithm (for instance, 2-choice hashing, cuckoo hashing) and populates the b buckets accordingly. The result is an allocation that balances the load of the k elements among the b buckets (as we discuss in Section 6.3). The consumer then ends its simulation and uses the resulting allocation to fetch the k elements from the buckets that were populated by the real producer.

Guaranteeing that the consumer's allocation is compatible with the producer's actions is challenging. One reason is that the consumer's simulation is acting on k items whereas the real producer is acting on n items. If the allocation algorithm being used (by the consumer and the producer) is randomized or depends on prior choices (this is the case with multi-choice hashing schemes), the allocations will be different. For example, observe that if a producer generates the allocation in Figure 6.2 it would not be compatible with the consumer's simulation in Figure 6.3(a) despite both entities using the same algorithm (since the producer places the item under label "2" in the middle bucket, but the consumer's simulation maps it to the top bucket).

To guarantee compatibility we employ a simple solution: the producer follows the same allocation algorithm as the consumer's simulation (for example, 2-choice hashing) on its n elements but stores the elements in *all* candidate buckets. That is, whenever the algorithm chooses one among w candidate buckets to store an element, the producer stores the element in all w buckets. This ensures that regardless of which k elements are part of the consumer's simulation or which non-deterministic choices the algorithm makes, the allocations are always compatible (Figure 6.3(b)). Of course this means that the producer is replicating elements, which defeats the point of load balancing. However, PBCs only need load balancing during data retrieval.

6.5 A PBC from reverse cuckoo hashing

We give a construction that uses Cuckoo hashing [174] to allocate balls to bins. However, the same method can be used with other algorithms (for example, multi-choice Greedy [30], LocalSearch [136]) to obtain different parameters, and can even be hybridize with traditional batch codes to achieve even better amortization (§6.5.2). We give a brief summary of Cuckoo hashing's allocation algorithm below.

Cuckoo hashing algorithm. Given n balls, b buckets, and w independent hash functions $h_0, \ldots, h_{w-1}$ that map a ball to a random bucket, compute w candidate buckets for each ball by applying the w hash functions. For each ball x, place x in any empty candidate bucket. If none of the w candidate buckets are empty, select one at random, remove the ball currently in that bucket (x_{old}), place x in the bucket, and re-insert x_{old}. If re-inserting x_{old} causes another ball to be removed, this process continues recursively for a maximum number of iterations.

Construction. Let H be an instance (producer, consumer) of reverse hashing where the allocation algorithm is Cuckoo hashing with w independent hash functions and b bins (we discuss concrete values for w and b later in this section). We construct a (n, m, k, b, p)-PBC as follows.

Encode(DB). Given a collection DB of n key-value tuples, follow H's producer algorithm to allocate the n elements to the b buckets. This results in $m = wn$ total elements distributed (not necessarily evenly) across the b buckets. Return the buckets.

GenSchedule(L). Given a set of k labels L, follow H's consumer algorithm to allocate the k labels to the b buckets. Return the mapping of labels to buckets. If more than one label maps to the same bucket (that is, if there are collisions), return $\perp$ instead.

Decode(W). Since Encode performs only replication, all codewords are elements in DB and require no decoding. Furthermore, σ, which is returned by GenSchedule, has only one entry for each label (since there is no need to combine codewords). As a result, W contains only one codeword. Decode returns that codeword.

6.5.1 Concrete parameters

Analyzing the exact failure probability of Cuckoo hashing, and determining the constant factors, remains an open problem (see [103] for recent progress). However, several works [66, 91, 182] have estimated this probability empirically for different parameter configurations. Following the analysis in [91, Appendix B], we choose $w = 3$ and $b = 1.5k$. In this setting, the failure probability is estimated to be $p \approx 2^{-40}$ for $k > 300$ (for smaller k it is around 2^{-20}). Figure 6.1 compares this result with existing batch codes.

6.5.2 Lowering the failure probability further

For some applications, one query failing out of 2^{20} (or even 2^{40}) queries might not be acceptable. In the above construction, one can lower p by either increasing the number of replicas (w), increasing the number of buckets (b), or both. More replicas lead to higher computational costs (due to PIR's costs being linear in the total number of elements), while more buckets lead to higher communication costs in PIR (due to clients having to query each bucket to maintain privacy). Demmler et al. [91, §5.2] show that increasing b leads to roughly a linear decrease in the failure probability of Cuckoo hashing. The effect of increasing w on the failure probability appears less significant [91, Appendix B], though these findings are based on limited experiments (only values of $w = 2$, 3, and 4 were used).

There is, however, another way to lower the probability of failure: increasing the number of collisions that every bucket can handle. We describe this technique here for completeness, but we note that we do not use this in Pung, since Pung can tolerate a failure probability of 2^{-20}. In the above scheme, a failure occurs whenever the allocation resulting from the consumer's algorithm has a max load greater than 1 (in other words, there is at least 1 collision). However, we can extend PBCs to support a max load of t. The key idea is to encode each bucket i with a (n, m, k, b)-batch code, where $n = |C_i|$ (the number of elements in bucket i), $k = t$ (the desired max load that the bucket should support), and m and b are parameters that depend on the batch code (see Figure 6.1).

Understanding how all of these parameters interplay in the context of our reverse cuckoo hashing PBC remains a challenging problem due to the complexity of analyzing cuckoo hashing's failure probability. However, Appendix B describes and evaluates the failure probability for several other PBC constructions; these are not as efficient as the Cuckoo PBC described in this chapter, but we include them for comparison and completeness.

6.6 Multi-query PIR from PBCs

In this section we describe how we can combine any PBC with a PIR protocol to obtain a multi-query PIR scheme. We give the pseudocode in Figure 6.4. At a high level, the server encodes its database by calling the PBC's Encode procedure. This produces a set of buckets, each of which can be treated as an independent database on which clients can perform PIR. A client who wishes to retrieve elements with labels $L = \{\ell_0, \ldots, \ell_{k-1}\}$ can then locally call $\mathsf{GenSchedule}(L)$ to obtain a schedule σ. This schedule states, for each label, the bucket from which to retrieve an element using PIR. Because of the semantics of GenSchedule it is guaranteed that no bucket is queried more than once (or $\sigma = \perp$). As a result, the client can run one instance of PIR on each bucket. However, a challenge is determining *which* index to retrieve from each bucket: by assumption (of PIR) the client knows the index in *DB*, but this has no relation to the index of that same element in each bucket. To address this, we introduce an oracle $\mathcal{O}$ that provides this information (we discuss it below). If the client has nothing to retrieve from a given bucket, the client simply queries a random index for that bucket.

Constructing the oracle $\mathcal{O}$. There are several ways that the client can construct $\mathcal{O}$. The simplest solution is to obtain the mapping from each index in *DB* to the corresponding indices in each bucket. For example, Pung clients can obtain this mapping in a succinct Bloom filter (§4.3.4). Another option, which Figure 6.4 uses, is for the client to fetch elements using BST-Retrieval (§4.3.3). In this case, the client need not know the index of a particular tuple in *DB* or in the corresponding buckets.

```
1:  function SETUP(DB)
2:      // run by the server
3:      (C_0, ..., C_{b-1}) ← Encode(DB)
4:      for j = 0 to b − 1 do
5:          SETUP(C_j)                                  // See Figure 5.1, Line 1
6:
7:  function MULTIRETRIEVAL(pk, sk, L, {|C_0|, ..., |C_{b-1}|})
8:      σ ← GenSchedule(L)
9:      if σ ≠ ⊥ then
10:         // get an element for each bucket
11:         // pick a random label if the bucket is not used in σ
12:         for j = 0 to b − 1 do
13:             ℓ_j ← label for bucket j (based on σ)
14:             // run BST-Retrieval on bucket j
15:             cw_j ← BST-RETRIEVAL(pk, sk, ℓ_j, |C_j|)        // see Figure 4.4
16:
17:         // select codewords from cw that are relevant to each label in L
18:         for i = 0 in k − 1 do
19:             W ← codewords from cw (based on σ[L_i])
20:             e_i ← Decode(W)
21:         return (e_0, ..., e_{k-1})
22:
23:     else deal with failure (see §6.6.2)
```

Figure 6.4: Multi-retrieval PIR protocol that combines the CPIR protocol of Figure 5.1, the BST retrieval protocol of Figure 4.4, and a PBC $(\mathsf{Encode}, \mathsf{GenSchedule}, \mathsf{Decode})$. L is the set of k desired labels and $|C_i|$ is the size of bucket i.

6.6.1 Selective failure attacks

In Pung, the server is not in charge of assigning labels to tuples. Instead, clients generate these labels before sending the tuples to the server (§4.2). Since these labels are pseudorandom, the server cannot control or predict ahead of time in which buckets a particular tuple will end up after calling Encode. This prevents the server from using the probabilistic nature of a PBC to conduct *selective failure attacks* (we discuss these attacks below). Of course the server could choose to not follow the Encode protocol, but this would simply lead to a denial of service (in the same way that the server could choose not to store a tuple in its collection during single-retrieval).

The more interesting case occurs in other applications of multi-query PIR (besides Pung) where the server does get to assign labels to tuples. In such cases, the server has full control over where to place elements by nature of its ability to assign labels. As a result, the server could assign labels to specific elements in a way that these elements cannot be retrieved together (that is, at sets of labels where GenSchedule returns $\perp$). This opens the door to attacks where the server selectively makes certain combinations of elements not retrievable in hopes of observing a client's reaction and breaking the privacy guarantees. Note that a similar attack already exists in the single-query PIR case: the server can selectively place an incorrect element (or garbage) at a particular index and can wait to see if a client complains or not (thereby learning that the index that the client requested was one that contained garbage or not). To address "selective failure" attacks, additional mechanisms are needed. A common solution is to ensure that a client's reaction remains independent of whether or not queries succeed. This guarantees that the attack violates availability instead of privacy—which a malicious server could violate anyway by not answering queries.

6.6.2 Dealing with failures in Pung

If the PBC that Pung uses has $p > 0$, then it is possible that for a client's choice of labels, $\sigma = \perp$. In this case, the client is unable to fetch all k messages privately. Notice, however, that the client learns of this fact before issuing any PIR query (see Figure 6.4, Line 9). As a result, the client has a few options. First, the client can ad-

just its set of labels (meaning the client can choose different elements to retrieve). This is possible in Pung whenever the client needs to retrieve more than k messages. Second, the client can retrieve a subset of the elements. In Pung, this would mean that the client would not retrieve all unread messages in the same round. However, since Pung messages are stored for several rounds, the client can try again the next round (presumably with a new set of labels). Lastly, the client can simply ignore the unretrievable messages. In our implementation we use this last option since we implement a reliable transport layer on top of Pung that ensures the sender retransmits messages until the recipient acknowledges them.

6.7 Summary and future work

In this chapter we discussed a relaxation of batch codes called PBC. We then used PBCs to build a multi-query PIR scheme that amortizes computational costs (§6.6) while introducing low network overhead. PBCs use PIR as a black box and therefore work with both XPIR and SealPIR. The key building block of PBCs is an allocation algorithm (we use hashing schemes) that assigns balls into bins in a way that minimizes the number of collisions. In this dissertation we study allocation algorithms that are typically used for online load balancing (that is, when balls arrive one at a time). In the future, we could consider algorithms that optimize for the offline setting in which all balls are available at the same time (which is the case in PBCs). In this setting, Czumaj and Stemann [81] show that the allocation process can be phrased in terms of orienting the edges of undirected graphs in order to obtain directed graphs with minimum in-degree. Optimal solutions for this problem can be computed in polynomial time [67], and linear time approximations also exist [57, 81, 101].

Chapter 7

Implementation

This chapter discuss the prototype implementations of SealPIR, *mPIR* (our multi-query PIR library), and Pung. All the code is available at: `https://github.com/pung-project`.

SealPIR implements XPIR's protocol [20] atop the SEAL homomorphic encryption library [11] (version 2.3.0-4) and extends it with our new query compression and Expand procedure (§5). This is around 2,000 lines of C++ and Rust (on top of SEAL). The most intricate component is Expand (Figure 5.3) which requires the substitution homomorphic operation (§5.4). We implement this operation in SEAL by porting the Galois group actions algorithm from Gentry et al. [108, §4.2]. We discuss this in detail in Appendix A.1.

SealPIR exposes the API in Figure 5.1. A difference with XPIR is that substitution requires auxiliary cryptographic material to be generated by the client and be sent to the server (see Appendix A.1). However, a client can reuse this material across all of its requests, and it is relatively small (2.9 MB per client).

Our multi-query PIR library, mPIR, implements the scheme described in Chapter 6.6, instantiated with the reverse Cuckoo hashing PBC described in Chapter 6.5. This library works transparently on top of both XPIR and SealPIR, and is written in 1,700 lines of Rust.

Details of Pung. We express the server side components of Pung as a series of nodes in a dataflow graph; we use Naiad's timely dataflow model [169]. Front-end

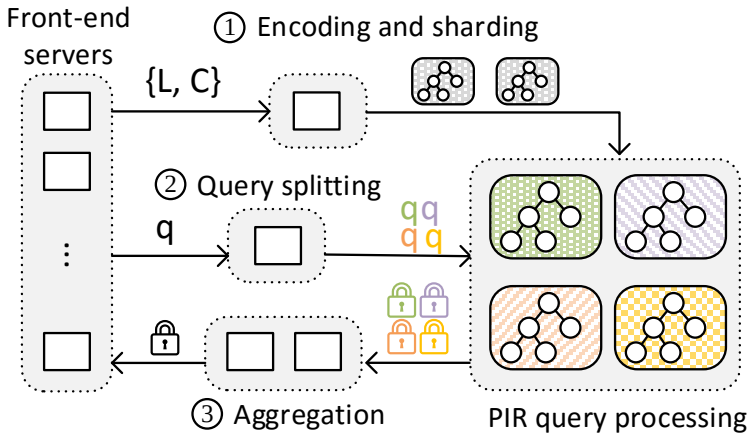


Figure 7.1: Pung's server-side architecture. Front-end servers receive requests from users and forward them to workers that process these requests. The collection of $(label, c)$-tuples is sharded at the granularity of buckets, and these shards are replicated across multiple workers that process PIR queries (see text for details). q represents a query, and the lock symbol signifies a PIR answer.

servers receive requests directly from users, aggregate these requests, and load balance them across dataflow workers. These requests cause workers either to modify the database of messages by inserting new tuples, or to read through the database to produce external output (for example, respond to a PIR query).

We depict a round of Pung in Figure 7.1. In the send phase (§4.1), clients send their $(label, c)$-tuples to front-end servers (we do not depict clients in the figure). Front-end servers hand these tuples to workers that transform the existing collection (Figure 7.1, step ①) by sorting the tuples, and encoding them with a batch code or PBC that is compatible with BST retrieval (§4.4.3). The workers then shard the resulting collection at the granularity of buckets and pass each shard to a PIR processing worker node. To support long-lived messages and to allow users to retrieve messages sent to them during past rounds, workers maintain a sliding window of messages. Pung's dataflow workers implement this window by mixing new and existing messages, garbage collecting the messages that outlive the sliding window, and reconstructing buckets and BSTs every round.

During retrieval, clients send to front-end servers multi-queries, which are

requests that contain queries going to multiple buckets (§4.4). Front-end servers forward these multi-queries to nodes that split and route individual queries to the PIR processing nodes holding the appropriate bucket (step ②). Aggregation workers collect the responses from PIR processing nodes and combine them to form the answer that is ultimately sent to front-end servers and then to clients (step ③).

We implement Pung in 5,800 lines of Rust and C++ bindings (not counting cryptographic operations, PIR, and PBC). We implement the server-side computation with the Timely Dataflow library [160] that creates, runs, and coordinates dataflow workers. We use mPIR to encode the stored messages with a PBC, and SealPIR (§5) to answer PIR queries. When the number of messages is small, clients use a Bloom filter (§4.3.4) rather than BST retrieval (§4.3.3). To construct the Bloom filter, our implementation uses MetroHash [10] as the non-cryptographic hash function, and it targets a false positive probability of 2^{-18}. Finally, we derive keys from secrets with HKDF [141], generate labels with HMAC-SHA256, and encrypt messages with ChaCha20-Poly1305. All of these operations are supported by the Rust-Crypto library [6].

We also implement the add-friend and dialing protocols of Section 4.5. These protocols run on a parallel service and are independent of the above Pung implementation. To implement dialing (§4.5.2), we use HMAC-SHA256 as the PRF, and MetroHash for the Bloom filter. To implement add friend (§4.5.3), we use the Cramer-Shoup public key cryptosystem [80] with 3072-bit keys, and the Edwards-curve Digital signature algorithm (EdDSA) [42] with curve 25519 [41] implemented in Rust's Ring library [16].

Chapter 8

Evaluation

This chapter evaluates the prototype implementations of SealPIR, mPIR, and Pung described in the previous chapter. Figure 8.1 summarizes the main results.

8.1 Experimental setup and concrete parameters

We run our experiments using Microsoft's Azure instances in three data centers: West US, South India, and West Europe. We run the PIR servers on H16 instances (16-core 3.6 GHz Intel Xeon E5-2667 and 112 GB RAM), and clients on F16s instances (16-core, 2.4 GHz Intel Xeon E5-2673 and 32 GB RAM), all running Ubuntu 16.04. We compile all our code with Rust's nightly version 1.25. For XPIR, we use the publicly available source code [21] and integrate it into our testing framework using Rust wrappers. We report all network costs measured at the application layer. We run each experiment 10 times and report averages from those 10 trials. Unless otherwise noted, standard deviations are less than 10% of the reported means. Finally, we run all microbenchmarks using Rust's criterion library [14].

Security parameters. We choose security parameters for FHE following XPIR's latest estimates [12], which are based on the analysis and tools by Albrecht et al. [22]. We set the degree of ciphertexts' polynomials to 2048, and the size of the coefficients to 60 bits (N and q in Section 5.4). Specifically, SEAL uses a value of $q = 2^{60} - 2^{18} + 1$,

SealPIR reduces the size of queries by 274× over XPIR.	§8.2.1
The end-to-end response time to privately fetch an element from a server on an average network is 42% lower with SealPIR than with XPIR.	§8.2.2
SealPIR achieves 23% lower throughput than XPIR (with $d = 2$), but 50% higher throughput than XPIR (with $d = 3$).	§8.2.3
mPIR reduces the computational costs of processing a batch of 256 queries by 40.5× (over processing each query individually).	§8.3
Pung can support 131K users with 3-minute communication rounds on 60 VMs.	§8.4.2

Figure 8.1: Summary of the main evaluation results.

whereas XPIR uses $q = 2^{61} - i \cdot 2^{14} + 1$, for various values of i [13].

Each database element is 288 bytes. We choose this size since we use 288-byte messages in Pung (§8.4). Unless otherwise stated, SealPIR uses a plaintext modulus t=2^{23}. A larger t leads to lower network and computational costs, but might cause noise to grow too much, preventing ciphertexts from decrypting successfully (we lower t in some experiments to ensure that we can always decrypt the result). For XPIR, we use $\alpha = 14$, meaning that we pack α elements into a single XPIR plaintext, thereby reducing the number of elements stored in the database by a factor of α. For 288-byte elements and our security parameters, setting $\alpha = 14$ in XPIR has the same effect as setting $t = 2^{23}$ in SealPIR without the optimization to Expand discussed in Section 5.7.1. With the optimization, SealPIR packs fewer data into plaintexts than XPIR (since it depends on t' rather than t, and $t' < t$); XPIR therefore ends up processing over a database with fewer total elements than SealPIR. Note that it does not make sense to use a lower value of α for XPIR to try to match the effect that t has on the optimized variant of SealPIR since that would not be fair to XPIR (lower values of α lead to higher computational costs).

8.2 Evaluating SealPIR

In this section we evaluate SealPIR in isolation, since it is a general-purpose PIR library that can be used in contexts beyond Pung. We answer two main questions:

1. What are the concrete resource costs of SealPIR, and how do they compare to the baseline of XPIR?
2. What is the throughput and latency achieved by SealPIR under different deployment scenarios?

8.2.1 Cost and performance of SealPIR

To evaluate SealPIR, we run a series of microbenchmarks to measure: (i) the time to generate, expand, and answer a query; (ii) the time to extract the response; and (iii) the time to preprocess the database. We study several database sizes and repeat the same experiment for XPIR using two different dimension parameters d (§5.7). Figure 8.2 tabulates our results.

CPU costs. We find that the computational costs of query generation are an order of magnitude lower under SealPIR than under XPIR. This is because the client in SealPIR generates d ciphertexts as a query rather than $d\sqrt[d]{n}$ ciphertexts as in XPIR (§5.7). When it comes to the server, SealPIR's Expand procedure introduces CPU overheads of 11% to 38% (over answering a query vector directly using SealPIR's code base). While this is high, it results in significant network savings (which we discuss below). Furthermore, even with the overhead of Expand, the cost of answering a query in SealPIR is comparable to XPIR.

We note that larger values of d lead to more computation for the server for two reasons. First, structuring the database as a d-dimensional hyperrectangle often requires padding the database with dummy plaintexts to fit all dimensions. Second, as we discuss in Section 5.7, the ciphertext expansion factor effectively increases the size of the elements by a factor of F after processing each dimension, necessitating more computation.

	XPIR (d = 2)			XPIR (d = 3)			SealPIR (d = 2)		
database size (n)	65,536	262,144	1,048,576	65,536	262,144	1,048,576	65,536	262,144	1,048,576
client CPU costs (ms)									
QUERY	13.83	27.57	55.14	4.98	8.03	12.74	3.37	3.37	3.37
EXTRACT	0.34	0.29	0.30	2.47	2.49	2.57	1.37	1.39	1.69
server CPU costs (sec)									
SETUP	0.15	0.57	2.27	0.15	0.58	2.32	0.23	1.04	4.26
EXPAND	N/A	N/A	N/A	N/A	N/A	N/A	0.05	0.11	0.23
ANSWER	0.21	0.63	2.12	0.27	0.78	2.52	0.13	0.5	2.01
network costs (KB)									
query	4,384	8,768	17,536	1,632	2,560	4,064	64	64	64
answer	256	256	256	1,824	1,952	1,952	256	256	256

Figure 8.2: Microbenchmarks of CPU and network costs for XPIR (for two database dimensions) and SealPIR under varying database sizes (n). Elements are of size 288 bytes.

Network costs. For network costs, SealPIR enjoys a significant reduction owing to its query encoding and EXPAND procedure (§5.6). For larger databases, the query size reductions over XPIR are 274× when $d = 2$, and 63× when $d = 3$.

8.2.2 SealPIR's response time

While microbenchmarks are useful for understanding how SealPIR compares to XPIR, another important axis is understanding how these costs affect response time and throughput. This section discusses response time, and the next section discusses throughput. To measure response time, we run experiments where we deploy a PIR server in Azure's US West data center, and place a PIR client under four deployment scenarios. We then measure the time to retrieve a 288-byte element using SealPIR, XPIR, and `scp` (the secure copy command line tool). We use `scp` to represent a client downloading the entire database (naive PIR).

Deployment scenarios

Intra-DC. The client and the server are both in the US West data center. The bandwidth between the two VMs is approximately 3.4 Gbps (measured using the `iperf` measurement tool). This scenario is mostly pedagogical since it makes little sense to use PIR inside two VMs in the same data center controlled by the same operator, but it gives an idea of the performance that PIR schemes could achieve if network bandwidth were plentiful.

Inter-DC. The client is placed in the South India data center. The bandwidth between the two VMs is approximately 800 Mbps. This scenario represents clients who deploy their applications in a data center (or well-provisioned proxy) that they trust, and access content from an untrusted data center.

Home network. The client is placed in the South India data center. We use the `tc` traffic control utility to configure the Linux kernel packet scheduler in both VMs to maintain a 20 Mbps send rate. We choose this number as it is slightly over the mean

download speed in the U.S. (18.7 Mbps) according to Akamai's latest connectivity report [8, §4]. This scenario is optimistic to XPIR since it ignores the asymmetry present in home networks where the uplink bandwidth is typically lower (meanwhile in XPIR, the queries are large). Nevertheless it gives a rough estimate of a common PIR use case in which a client accesses an element from their home machine.

Mobile network. The client is placed in the South India data center. We use `tc` to configure VMs to maintain a 10 Mbps send rate. We choose this number as it approximates the average data speed achieved by users across all U.S. carriers according to OpenSignal's 2017 State of Mobile Networks report [9] and Akamai [8, §8]. As with the home network, this scenario is optimistic (for XPIR) as it ignores the discrepancy between download and upload speeds. It represents the use of PIR from a mobile or data-limited device.

Results. Figure 8.3 depicts the results. At very high speeds (intra-DC), naive PIR (`scp`) is currently the best option, which is not surprising given the computational costs introduced by PIR. In this regime, SealPIR is competitive with both instances of XPIR, although our implementation falls behind on the largest database size. The primary issue is that, for a database with $n = 2^{22}$ elements, our optimization of Expand makes the plaintext modulus very small ($t' = 2^{12}$, see Equation 5.1 in Section 5.7.1). This causes SealPIR to use many more plaintexts than XPIR. For even larger databases, since we must use a higher dimension anyway (§5.8), the difference in the number of plaintexts between XPIR and SealPIR (for the same d) becomes less prominent until n is large enough that the second operand in Equation 5.1 approaches $\log(t)$ again.

With lower network speeds, XPIR and SealPIR significantly outperform `scp`. As bandwidth decreases (home, mobile), SealPIR's lower network consumption and competitive CPU costs yield up to a 42% reduction in response time.

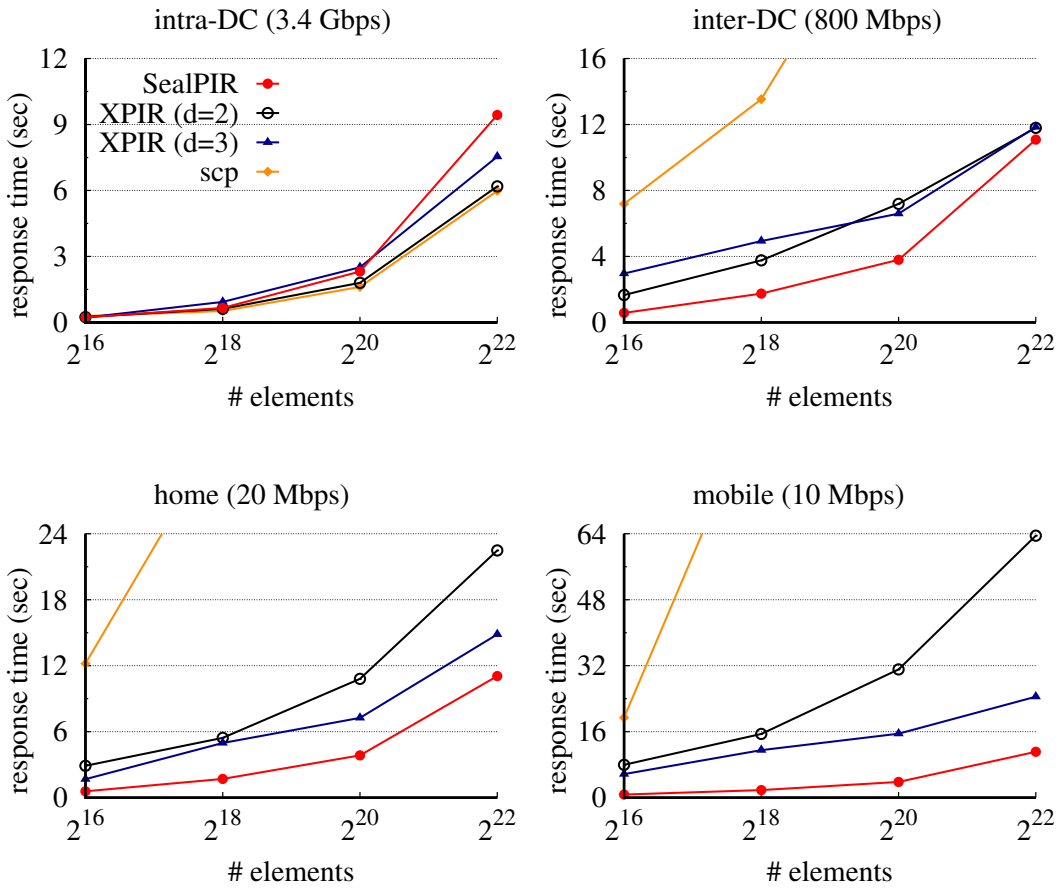


Figure 8.3: Mean response time experienced by a client under different deployments (see text for a description of network conditions) with different PIR schemes. When the network bandwidth is plentiful (intra-DC), downloading the entire database (`scp`) achieves the lowest response time. However, when the network bandwidth is limited (home, mobile), SealPIR achieves the lowest response time.

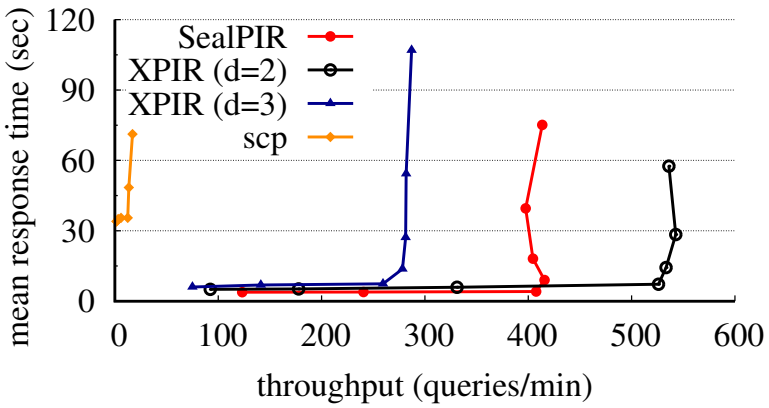


Figure 8.4: Comparing throughput vs. mean response time under SealPIR and XPIR (with $d = 2$ and $d = 3$) when using a database with 2^{20} elements where each element is 288 bytes long. We find that XPIR with $d = 2$ saturates at 9 requests/second whereas SealPIR saturates at 7 requests/second (a 23% reduction in throughput). When XPIR uses $d = 3$, SealPIR achieves about 50% higher throughput.

8.2.3 SealPIR's throughput

We deploy the PIR server in Azure's US West data center, but access it with an increasing number of concurrent PIR clients deployed across the South India and EU West data centers. We then measure the number of requests serviced per minute at the server, and the request completion times at the clients. Figure 8.4 depicts the results of running from 4 to 256 clients each requesting one 288-byte element from a database with 2^{20} entries. In our experiments, we ensure that the bottleneck is the server's CPU or WAN network connection, and not the clients or some link between specific data centers.

We find that SealPIR achieves a 50% higher throughput than XPIR with $d = 3$, but a 23% lower throughput than XPIR with $d = 2$. Most of the difference can be attributed to Expand, but we believe that with further engineering we can close this gap (since SealPIR is comparable to XPIR according to microbenchmarks). Compared to naive PIR via `scp`, SealPIR and XPIR achieve over 20× higher throughput since the primary bottleneck in naive PIR is network bandwidth and not CPU (which is the bottleneck for both SealPIR and XPIR).

	single-query	mPIR		
batch size (k)	1	16	64	256
client CPU costs (ms)				
MultiQuery	3.07	6.45	5.26	4.92
MultiExtract	2.51	3.26	3.25	2.70
server CPU costs (sec)				
MultiSetup	6.1	1.50	0.38	0.12
MultiAnswer	3.24	0.69	0.23	0.08
network costs (KB)				
query	64	96	96	96
answer	384	480	480	384

Figure 8.5: Per-request (amortized) CPU and network costs of mPIR on a database consisting of 2^{20} elements, with varying batch sizes. The second column gives the cost of retrieving a single element (no amortization). The underlying PIR library is SealPIR with $t = 2^{20}$ and elements are 288 bytes.

8.3 Evaluating mPIR

In this section we evaluate how using a PBC to build a multi-query PIR scheme as described in Chapter 6 can help amortize computational costs, and what kind of network overhead it introduces. In particular, we evaluate our multi-query PIR library, mPIR, which we use in Pung, but which can also be used more generally. Our experiment consists of repeating the microbenchmark in Section 8.2.1, but this time we use mPIR. The baseline of this experiment is performing k parallel instances of PIR, which is the naive way to retrieve k elements using single-retrieval. In this experiment we use SealPIR with $t = 2^{20}$ and $d = 2$ as the underlying PIR library.

Figure 8.5 gives the results. We find that mPIR reduces computational costs over the baseline up to 40.5× for a batch of $k = 256$ queries and a database with $n = 2^{20}$ elements. This is a direct effect of mPIR performing fewer operations at the server. In particular, recall from Section 6.5 that the Cuckoo PBC (which is what mPIR uses) generates $1.5k$ buckets and $3n$ total codewords. This results in $1.5k = 384$ buckets, each of which contains on average 2^{13} elements (each element is replicated 3 times

and each replica is assigned to a random bucket). mPIR then queries each bucket exactly once. By contrast, the baseline requires the server to answer k queries on a database with 2^{20} elements, which results in 85× more operations.

Observe that at $k = 256$, mPIR's download costs are the same as the baseline. This is counterintuitive since mPIR results in 50% more answers (the extra answers are dummies that hide which buckets are of interest to the client; see Section 6.6). However, each answer in mPIR contains fewer ciphertexts because of the interaction between SealPIR and mPIR. Recall from Section 5.1.1 that if $d > 1$, the number of ciphertexts in an answer is F^{d-1}; F is the cryptosystem's expansion factor, which in our case is $F = 2\log(q)/\log(t')$. Furthermore, Equation 5.1 (§5.7.1) shows that t' is larger for smaller databases. Indeed, for the original 2^{20}-entry database, $t' = 2^{10}$ (resulting in $F = 12$), whereas for the average bucket of 2^{13} entries, $t' = 2^{15}$ (resulting in $F = 8$). Consequently, for our choice of parameters, the total download communication ends up being the same: $256 \cdot 12 = 384 \cdot 8$ ciphertexts.

Note that this parity in download cost is not true in general; it is a result of the particular parameters used in this case. In fact, because of Equation 5.1 (§5.7.1), we can even achieve *lower* amortized download costs. Without Expand's optimization, this would not be the case: in some sense, the optimization introduces communication overhead to fetching elements from databases with many entries and mPIR amortizes that overhead. As an aside, Equation 5.1 does not affect upload costs; these costs increase by 50% since the client is sending 50% more queries.

8.4 End-to-end evaluation of Pung

In this section we focus on three main evaluation questions:

- *How many users can Pung support, and how does it compare to prior systems?* We answer this question by measuring throughput and latency as we vary the number of users in our end-to-end deployment (§8.4.1 and §8.4.2).
- *What are the benefits of using mPIR?* We discuss the throughput benefits and also the corresponding network overheads of clients using mPIR to retrieve multiple messages from Pung's servers (§8.4.3).

- *How expensive is Pung for clients?* We measure the CPU and network costs incurred by all operations performed by clients (§8.4.4).

We answer these questions in the context of the following setup and baselines.

Setup and metrics. We deploy Pung's server logic on timely dataflow workers running on the H16 VMs (§8.1). Our performance metrics are throughput (in messages/minute) and end-to-end latency (in seconds). Note that all entities run on the same data center, so our results do not capture the effects of wide area networking.

We run clients and dataflow workers in a closed loop and let round duration be as low as possible: a new round starts as soon as all current requests are fulfilled. To keep the number of messages constant across rounds, we configure Pung's garbage collection window to be the number of messages sent in one round (§7).

Baselines. We compare Pung to two prior systems: Dissent [77] and Vuvuzela [216]. Vuvuzela is the state-of-the-art in private communication under the *anytrust model* [226], whereas Dissent is the state-of-the-art in the no-trust model. The anytrust model states that out of a set of servers one is assumed to be correct, but clients need not know which is the correct one. We want to emphasize that our comparison to Dissent is not entirely fair: Dissent achieves an additional privacy property—sender anonymity (§3)—that Pung does not provide. However, we are not aware of any system that provides the same guarantees as Pung under the same threat model.

8.4.1 How many users can Pung support?

The number of users that Pung can support depends on the performance that users are willing to tolerate and the financial cost that the provider is willing to shoulder. We therefore focus on measuring the impact of the number of users on two end-to-end metrics: latency observed by a client and throughput achieved by Pung's servers. Here we test the version of Pung that we describe in Section 4 with SealPIR. Note that since our goal is to determine the number of users that Pung can support, we set the batch size used by mPIR to $k = 1$ (in other words, Pung gets no amortization). We

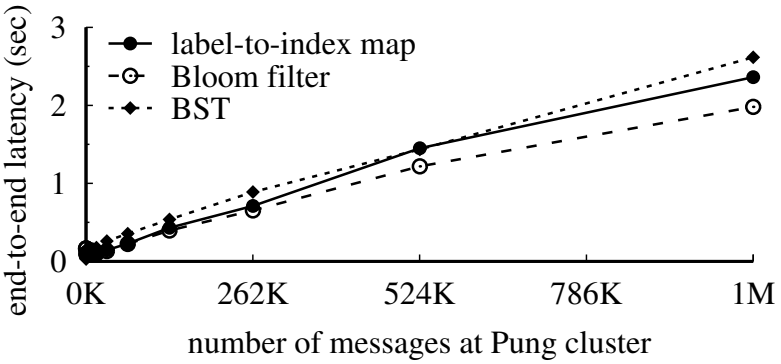


Figure 8.6: The end-to-end latency of sending and retrieving one message when Pung's server is under-utilized is up to 1.3 seconds (when the server stores 1 million messages). This figure gives the round-trip time of the three different retrieval methods discussed in Chapter 4.

do this because mPIR only helps to amortize the cost when clients retrieve multiple messages, but it does not actually help Pung support more clients.

Latency. Our goal is to understand what is the lowest round duration that Pung could possibly support given enough hardware resources. To do this, we measure the end-to-end latency perceived by a client in Pung when its request is handled by a dedicated dataflow worker (that is, by a dedicated CPU core). We have the client send its message and perform a retrieval. To experiment with large collection sizes we populate the server with up to 1 million 288-byte tuples. We experiment with three different methods that the client can use to retrieve its desired tuple from the server. The first has the client explicitly download all the label-to-index mappings prior to retrieval, look up the index of the corresponding label locally, and perform PIR with this index. The second downloads a Bloom filter that succinctly encodes the label-to-index mappings (§4.3.4), and performs the same steps as above. The last performs the BST retrieval procedure given in Figure 4.4.

Figure 8.6 depicts the results. As we expect from our microbenchmarks of SealPIR (§8.2.1), the client latency grows linearly with the number of messages at the server. Also, our low-latency network allows us to confirm that the server-side

CPU costs associated with BST retrieval are comparable to explicitly fetching the label-to-index mapping. This is due to our ability to express the complete BST as a contiguous array which requires no padding or auxiliary elements (§4.3.3). However, in wide area networks we expect to see added latency due to $\log(n)$ round trips. The Bloom filter's checks (§4.3.4) also incur little CPU overhead to the client, and its size is up to 10.4× smaller than the associated label-to-index mapping. However, for large databases, the Bloom filter consumes more network resources than the BST retrieval (see the discussion in Section 4.3.4)

Finally, note that our prototype performs request-level—rather than data-level—parallelism, so these latencies could be reduced further by having dataflow workers process fractions of a request. This is possible because PIR answers are ciphertexts from an additively homomorphic cryptosystem, so workers can generate partial answers that can then be aggregated (§5.2). However, the current latencies (assuming enough computational resources) are already much better than those achieved by Vuvuzela, where even a two-client deployment requires 20 to 30 second rounds due to the addition and serial processing of cover traffic (adding more machines does not reduce this latency).

Throughput. To measure Pung's peak throughput, we run experiments where clients send and retrieve a 256-byte message per round, for a total of 10 rounds. We then vary the number of clients (n) and measure the number of messages processed per minute. We distribute 64 timely dataflow workers across 4 VMs to run Pung's server-side computation. Since we cannot run tens of thousands of clients in our infrastructure, we employ a combination of real and simulated clients. We configure 512 real clients across 8 H16 VMs (4 clients per core). We then have each client send a single message and instruct dataflow workers to make up the difference by injecting the remaining messages (n–512) at the end of the send phase, simulating additional clients. Finally, during the retrieve phase, each real client fetches a message from a random mailbox.

We also run both baselines in our cluster, with 256-byte messages. Since Dissent is a peer-to-peer system and does not use servers, we spread out its peers across

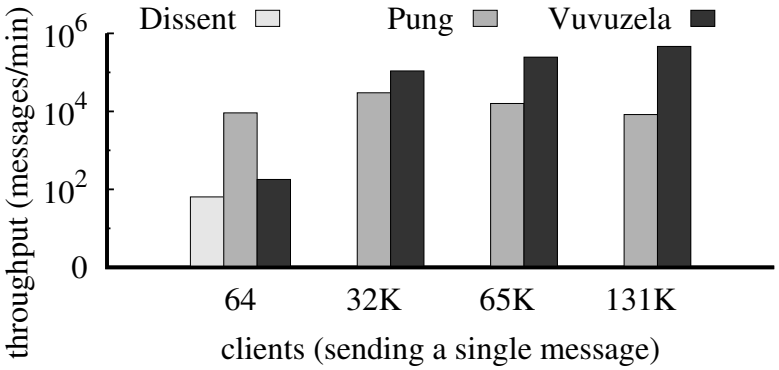


Figure 8.7: Pung can handle significantly more messages and clients than Dissent but its throughput at 131K clients (on the same hardware) is 55.7× lower than Vuvuzela's. We do not report Dissent's throughput past 64 users (see text for details).

our VMs. We run only its shuffle protocol as that is more efficient than the full Dissent protocol for small fixed-sized messages [77, §3].

For Vuvuzela, we set up a 3-server chain in addition to the entry server that proxies client requests, which mirrors the arrangement evaluated by its authors [216, §7]. A caveat is that our VMs have fewer CPU cores. We also use the same parameters that characterize the distribution from which Vuvuzela servers draw noise (μ = 300,000 and b = 13,800). We run 512 Vuvuzela clients and modify the entry server [7] to make up for the remaining messages (similar to how Pung's dataflow workers inject messages).

Figure 8.7 depicts our results for 64, 32K, 65K, and 131K clients. We show Dissent's throughput only with 64 clients because at higher peer counts it is less than one message per minute with the prototype we use [5]. Pung and Vuvuzela achieve relatively low throughput—far below their capacity—at very low client counts. This is due to lack of work, since only 64 clients are sending and retrieving messages in a given round. As a result, Pung workers sit idle most of the time, while Vuvuzela servers continue to generate and process significant cover traffic, delaying the start of the next round. However, at higher client counts, there is enough work to make long rounds a non-issue for Vuvuzela. Indeed, Vuvuzela's throughput is 55.7× higher than Pung at 131K clients, and this gap grows larger with more clients.

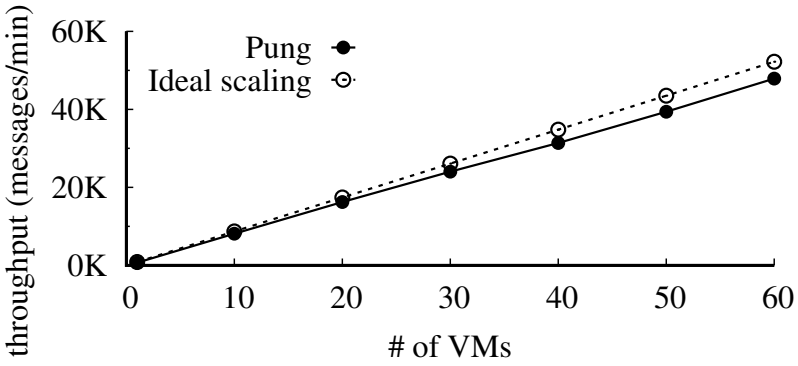


Figure 8.8: Throughput of Pung for 131K clients (sending and retrieving a single message) with varying number of VMs. Pung achieves near-linear horizontal scalability. For the ideal line, we choose the fastest single-server performance across all 60 VMs, and multiply it by the number of servers.

8.4.2 Can Pung scale out to support more users?

In the above throughput experiment, we keep the number of VMs that Pung and Vuvuzela use at 4. This artificial limit is due to Vuvuzela's architecture not scaling out with more servers (although Stadium [213] proposes a way to extend Vuvuzela to achieve horizontal scalability). In contrast, Pung benefits greatly from additional computational resources owing to its bottleneck being computation (specifically answering PIR queries). To understand Pung's scaling, we rerun the above throughput experiment but we increase the number of VMs running Pung's servers. Figure 8.8 shows the results.

We find that a Pung deployment of 60 VMs and 131K users can complete a full round of communication (process all sent messages and all retrieval requests) in 2.8 minutes (a max throughput of around 47.7K requests/min). Assuming that a 3-minute round duration is acceptable to end users, this result supports our claim that a deployment of Pung can handle hundreds of thousands of users.

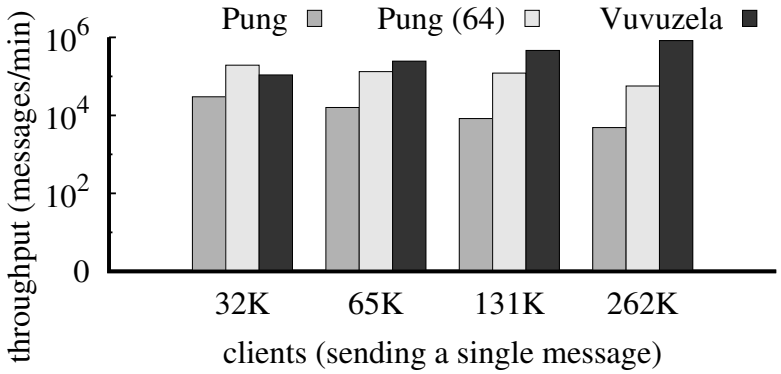


Figure 8.9: Throughput of Pung and Vuvuzela on 4 VMs. When clients retrieve multiple messages in Pung, its throughput increases by up to 16.1×. Pung (64) represents an instance of Pung where clients retrieve k=64 messages simultaneously using mPIR (§7). At 262K clients, Vuvuzela handles 169.8× and 10.7× more messages than Pung and Pung (64), respectively.

8.4.3 What are the benefits and costs of using mPIR?

Recall that in the previous section we used mPIR with a batch size of $k = 1$ (since our goal was to determine the number of users that Pung could support, and larger batch sizes do not help). We now discuss how a larger batch size impacts Pung in terms of throughput and network resources for a given number of users. To measure throughput, we run the same experiment described in Section 8.4.1, but configure Pung's servers and clients to use mPIR with varying batch sizes.

Throughput. We depict the throughput benefits of having clients retrieve a batch of $k = 64$ messages in Figure 8.9. We find that this offers a throughput boost of up to 16.1× over single retrieval. Given that the maximum theoretical gain that one can expect from using mPIR over retrieving messages one by one from a database with $n = 2^{18}$ elements is 21.3× for $k = 64$ (since the number of codewords produced by mPIR is $3n$, which is 21.3× lower than kn), our implementation achieves 76% of this gain. This is expected, since our end-to-end throughput measures not only message retrieval but also Pung's send phase—including the expensive PIR setup step (§8.2.1).

Nevertheless, Pung's multi-retrieval throughput is high enough that it can

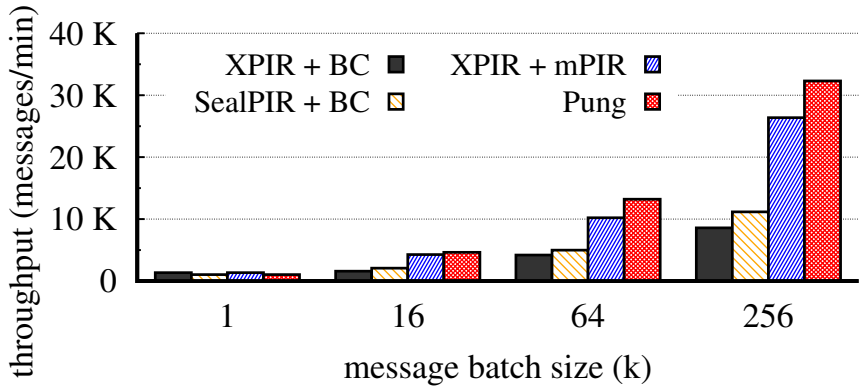


Figure 8.10: Throughput of Pung and some variants that act as baselines (variants have the same architecture as Pung but use different cryptographic building blocks) on one VM with 256K users, each retrieving k 288-byte messages per round. The XPIR + BC baseline refers to a system that uses XPIR [21] as the PIR library and a different PBC construction that we describe in Appendix B instead of mPIR. The SealPIR + BC baseline refers to a system that uses SealPIR as the PIR library as well as the PBC construction in Appendix B. The XPIR + mPIR baseline refers to a system that uses XPIR as the PIR library and mPIR to amortize the retrieval of multiple messages.

support group communication with rounds that are only a few minutes. We also experiment with values of k ranging from 4 to 128, and find throughput gains between 1.2×–34.1×.

Alternatives. To better understand the impact of the primitives introduced in this dissertation, namely SealPIR and PBCs (mPIR), we compare Pung to several baselines that share the same general architecture as Pung but use other building blocks (that is, a different PIR scheme, and a different PBC). We measure throughput by running the same experiment described in Section 8.4.1. The baselines are:

- *XPIR + mPIR*: This baseline has the same architecture as Pung, but clients use XPIR ($d = 2$) instead of SealPIR to privately retrieve messages from the servers.
- *SealPIR + BC*: This baseline has the same architecture as Pung, but clients do not use mPIR (which implements the PBC based on Cuckoo hashing described in

Section 6.5). Instead, clients use a multi-query PIR protocol based on a different construction of a PBC, which we describe in detail in Appendix B; at a high level, it combines the subcube batch code described in Section 4.4.2 with a PBC.

- *XPIR + BC*: This baseline has the same architecture as Pung, but clients use XPIR ($d = 2$) instead of SealPIR to retrieve messages from the server, and use a different multi-query PIR scheme instead of mPIR.

Figure 8.10 shows the throughput in messages per minute that Pung and the baselines achieve on a single VM. Pung's throughput is higher than that achieved by the baselines for all batch sizes greater than 1. There are three reasons for this. First, mPIR produces 50% fewer codewords than the alternate construction used by the baselines ("BC"), and fewer codewords translate directly into lower computational costs. Second, BC produces 7× more buckets than mPIR. This means that the XPIR + BC baseline has to run the XPIR protocol on many small databases that contain an average of 500 to 8,000 elements (depending on the batch size), which exacerbates XPIR's high fixed costs.

Last, even though SealPIR incurs additional CPU costs than XPIR ($d = 2$) on large databases as we show in Section 8.2.1 (this is also why the baselines that use XPIR have higher throughput than Pung when the batch size is 1 in Figure 8.10), SealPIR is slightly faster when the database is small (see the column with 65,536 elements in Figure 8.2). Ultimately, we find that if clients retrieve $k = 64$ messages, the throughput of Pung is 3.1× higher than that of the XPIR + BC baseline. This supports the claim that the optimizations introduced in this dissertation are important for Pung's architecture to perform well.

Network costs. To understand how Pung compares to the baselines in terms of network overheads, we plot the total communication costs (including uploading and downloading messages) for a single client resulting from one round of communication of the prior throughput experiment. We give the results in Figure 8.11.

We find that for single retrievals ($k = 1$), a user requires 380 KB (SealPIR) and 8.5 MB (XPIR) for sending and receiving a message. This cost is 3–5 orders of magnitude higher than sending and retrieving messages through Vuvuzela (or

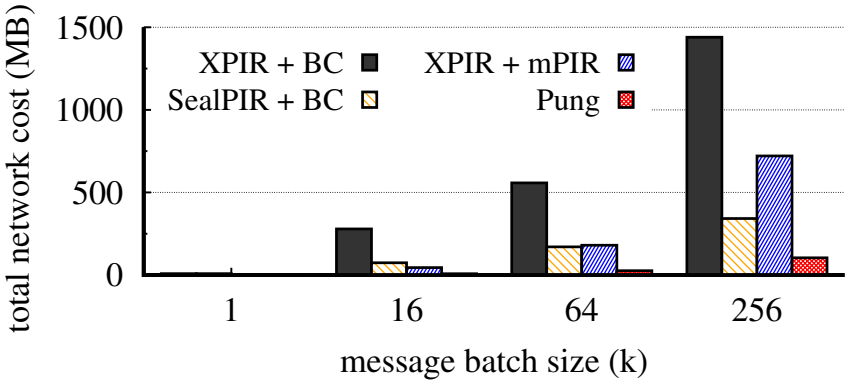


Figure 8.11: Per-user total network cost (upload and download) of Pung and several baselines with 256K users. Each user retrieves k 288-byte messages. See Figure 8.10 for an explanation of the baselines.

through a non-private service). However, compared to downloading the entire collection (which would also meet our privacy goals), it is 194× (SealPIR) and 8.6× (XPIR) lower. Nevertheless, we admit that in absolute terms these costs are high and are currently the main limitation of Pung's design.

For multi-query PIR, the benefits of Pung over the baselines are considerable. We find that the compressed queries that SealPIR produces and the fewer buckets that result from mPIR's encoding result in savings of over 36× (over the XPIR + BC baseline). In particular, the per-client communication costs are cut down to 7.7 MB per round for $k = 16$ in Pung, versus 279 MB in the XPIR + BC baseline. Compared to k-parallel instances of PIR, the network overheads of Pung are consistent with the microbenchmarks in Figure 8.5, and the accompanying discussion (§8.3).

8.4.4 What costs does Pung impose on clients?

Since Pung's client applications participate in every round to ensure privacy (§4.1), clients incur CPU and network costs regardless of whether the user is idle or not. In the previous section we discussed the total network costs that clients incur during one round of communication. These costs are the result of submitting $(label, c)$-tuples, PIR queries, and downloading bloom filters (or using BST-Retrieval). We

	CPU time	costs scale linearly with
Conversation (§4.1)		
Key derivation	6.05 μs	N/A
Label generation	1.60 μs	retrieval rate (k)
Message encryption	1.56 μs	send rate (s)
Message decryption	1.37 μs	retrieval rate (k)
Dialing (§4.5.2)		
Token generation	3.11 μs	dialing rate
Token lookup	0.07 μs	# friends
Add friend (§4.5.3)		
Signing key generation	18.18 μs	N/A
Encryption key generation	17.99 ms	N/A
Sign friend request	18.13 μs	friend request rate
Verify request signature	53.11 μs	# real friend requests
Encrypt request	43.9 ms	friend request rate
Decrypt request	5.25 ms	# total friend requests in Pung

Figure 8.12: Microbenchmarks for Pung's client operations. The cost of PIR operations are given in Figure 8.2. The second column gives the CPU time required to compute one operation. The last column describes what parameters cause these costs to increase linearly.

now turn our attention to the computational costs incurred by clients. We give the results in Figure 8.12.

For a round of communication, the CPU costs to the client are only a few milliseconds—including the cost to generate a PIR query which we show in Figure 8.2. Most of these costs scale with the send rate (s) and retrieval rate (k) that clients choose. Specifically, since a client needs to maintain these rates regardless of whether they are idle or not, the client always generates $s + k$ labels to send and retrieve messages, encrypts s real or dummy messages, generates a PIR multi-query that supports the retrieval of k elements, and decrypts k of the retrieved elements. Nevertheless, even for high send and retrieval rates (≈ 64), these costs are at most hundreds of milliseconds: the costliest operation is the PIR multi-query, which for $k = 64$ would cost 336 ms for the client to generate (Figure 8.2).

We now discuss the less frequent operations, namely dialing (§4.5.2) and

adding friends (§4.5.3). The costs of these operations should be taken in context: whereas a round of communication in Pung occurs every few minutes (§4.1), we expect dialing to be done every tens of minutes, and friend addition to be done once or twice a day (Vuvuzela [216] and Alpenhorn [147] assume similar frequencies). Dialing imposes negligible computational costs on a client, and the Bloom filter that the client downloads for 131K dial tokens is 384 KB. As a result, dialing's overhead is likely in line with the rest of Pung's cost, which we view as undesirable but not onerous.

The add friend operation, on the other hand, is expensive on all axes. First, each add friend request is 1.6 KB, and clients must download all of the friend requests submitted by all users during a friend request round. At 131K users, this results in each client downloading 209 MB worth of friend requests. Furthermore, clients must try to decrypt each of these friend requests to determine which ones are meant for them. Since the cost of each decryption is 5.37 ms, decrypting 131K friend requests would require 12 CPU minutes. While this operation is highly parallelizable and can be performed slowly throughout the day (since adding friends occurs once a day), the computational costs are high for a weak client such as a mobile device.

Chapter 9

Summary, limitations, and next steps

This dissertation studied how users can communicate over the Internet without anybody else learning that the communication has taken place, which is a property called metadata privacy (§3). The motivation is that metadata is itself sensitive; for example, if one learns that someone has made repeated calls to a doctor with a particular specialty, one may infer something about the caller's health. More vividly, the former director of the NSA and the CIA admits that inspecting metadata at large scale suffices for these agencies to conduct their operations [72]. This degree of mass surveillance endangers civil liberties and the expression of dissenting opinions, especially in regimes where other forms of communication would be perilous.

As we discussed in Chapter 2 the goal of hiding metadata is not new; however, most prior systems assume the existence of some trusted infrastructure. Meanwhile, our desire to consider strongly adversarial conditions led to a qualitatively different and stronger variant of the problem in which all infrastructure is untrusted. To hide metadata in this setting, this dissertation proposed a messaging system called Pung, and a series of refinements that made Pung less expensive in practice.

Hiding metadata brings several challenges; one of the most difficult is breaking the link between a message's sender and its recipient. Pung breaks this link by having senders deposit messages into an untrusted server, and having recipients fetch these messages using private information retrieval, or PIR (§4.3.1). PIR allows recipients to retrieve a message from the server without revealing to the server which

message was retrieved. While PIR is a powerful theoretical tool, using it in practice poses some challenges: PIR has a narrow interface, and its computational and network costs are very high.

To make PIR usable in practice, this dissertation made three key contributions. First, it leveraged and extended an efficient oblivious binary search procedure that augments PIR with basic search functionality (§4.3.3). This allows users to search the server for their incoming messages. Without this procedure, users would need to use some other method to find the location of their messages prior to fetching them with PIR. Second, it introduced the first mechanism to compress and obliviously decompress queries in PIR (§5). Perhaps surprisingly, both procedures are computationally inexpensive, and compressing queries cuts their size by up to 273× (§8.2). Last, this dissertation proposed the use of probabilistic batch codes (PBC) as a way to amortize PIR's computational costs (§6). While related data encodings, namely batch codes (§4.4.2), achieve the same type of amortization, their use results in onerous network costs (§6.1). In contrast, PBCs achieve orders of magnitude lower network costs (§8.3), but in exchange they introduce a small probability of failure (§6.6.2). However, the nature of the failure is that a client can retrieve only a subset of his or her desired messages, not that privacy is compromised. In Pung, this is not an issue since clients can retry at a later time.

With these improvements, our experimental evaluation showed that a small deployment of our Pung prototype can provide metadata-private communication to hundreds of thousands of users (§8.4).

Challenges remaining. Pung has many limitations; chief among them are its high network and computational costs. To support hundreds of thousands of users exchanging one message, the communication costs between a client and Pung's servers are hundreds of kilobytes per round. While for 3-minute rounds this comes down to about 16 kbps (low given today's network speeds), clients must be constantly online to preserve their privacy; on a monthly basis this cost is north of 5.4 GB, which is higher than many cellular data plans. Furthermore, adding friends in Pung (§4.5.3) requires clients to incur hundreds of megabytes of network communication and

spend over ten minutes of CPU time. Finally, to prevent against attacks where the adversary compromises a client's friends (§4.6), Pung must give up any type of dialing mechanism (preventing a client from starting a new communication with a friend) or incur very high costs (§4.6.5).

In addition to costs, Pung suffers from its inability to hide usage: an adversary in Pung's threat model knows that a user is part of Pung. This makes censorship possible—and in particular targeted censorship where only Pung's communication is blocked. While this is already true of deployed systems like Tor [96], Pung's centralized architecture makes this issue worse.

Next steps. There are several avenues for improving Pung. While Pung's current architecture appears hard to scale to billions of users, it might be possible to devise a hierarchical version in which smaller communities interact with each other. This would enable incremental deployments akin to the Internet and several social networks: cities, universities, or companies can roll out their own deployments of Pung for their citizens, students, and employees. Over time, these deployments can be interconnected with a deployment of Pung that bridges across institutions. Nevertheless, it remains an open question how to instantiate such a hierarchical architecture, how to ensure that communities are large enough to provide meaningful privacy, and how to route messages that cross organizations without leaking metadata.

A second path is to investigate ways to instantiate privacy-preserving answering machines. We described a few ideas and the corresponding challenges in Section 4.6; while it presently seems hard to devise an answering machine that is both privacy-preserving and efficient, we hope that we can improve the existing tradeoff.

Last, Pung and related systems require clients to be constantly online to preserve privacy. An exciting direction is devising ways to incentivize clients to stay online. One possibility is through the allocation of tokens that hold financial value. For example, the act of participating in the system produces a token that can be used to reduce the monthly fee of using Pung.

It might even be possible to go a step further and redesign Pung to be fully decentralized. After all, none of Pung's servers need to be trusted for privacy or in-

tegrity, and Pung's throughput scales linearly with the number of servers. In a decentralized setting, playing the role of a Pung server and processing users' send and retrieval requests could generate tokens. As a result, we could bootstrap an entire economy where clients buy tokens from servers or other clients, and use these tokens in order to send and receive messages in the system. Finally, clients who participate for long enough periods of time, in addition to receiving Pung's privacy guarantees, also receive tokens as compensation. Of course, this design requires solving many technical challenges, including developing ways for clients and servers to prove that they are participating in the system, techniques to prevent or detect malicious servers that deny service, and mechanisms for transferring and assigning tokens.

Appendix A

Details and correctness proofs of SealPIR

A.1 Substitution operator

We now give details on how the substitution operator is implemented. Let Φ_i be the i-th cyclotomic polynomial.[1] As we discuss in Section 5.4, we pick $\Phi_i = x^N + 1$, where N is a power of two (hence $i = 2N$). Recall from that same section that FV plaintexts are polynomials in the ring $R_t = \mathbb{Z}_t[x]/\Phi_i(x)$, and ciphertexts are two polynomials, each in the ring $R_q = \mathbb{Z}_q[x]/\Phi_i(x)$. The secret key sk is a randomly sampled polynomial in R_2.

Let $p(x)$ be the plaintext encrypted by ciphertext $c = (c_0, c_1)$. Our goal is to substitute in $p(x)$ every instance of x with x^k for some integer k, by operating directly on c. Gentry et al. [108, §4.2] show that if $k \in \mathbb{Z}_i^*$ (that is, k is odd so that it is coprime with i), performing the substitution directly on the ciphertext polynomials (c_0, c_1) and the secret key achieves this goal.

Specifically, let $c^{(k)}$ be the result of replacing every instance of x in the ciphertext polynomials c_0 and c_1 with x^k. Similarly, let $sk^{(k)}$ be the result of replacing every instance of x in the secret key sk with x^k. The result of decrypting $c^{(k)}$ with $sk^{(k)}$ is therefore $p(x^k)$—which is exactly what we want.

One issue with the above is that Expand (Figure 5.3) uses the output cipher-

[1] The i-th cyclotomic polynomial is the unique irreducible polynomial with integer coefficients that is a factor of $x^i - 1$ but not of $x^j - 1$ for any $j < i$.

text after substitution, $c^{(k)}$, and adds it to the input ciphertext c in each iteration of the inner loop (see Lines 10 and 11). This operation is not well defined since both ciphertexts are encrypted under different keys (substitution essentially changes the key under which the ciphertext is encrypted). To address this, we perform an operation called *key switching* [50], which allows us to transform an encryption of $c^{(k)}$ under some public key associated with $sk^{(k)}$, to an encryption of $c^{(k)}$ under some public key associated with the original key sk (which is the key under which c is also defined).

Note that the server needs some auxiliary information in order to perform key switching. In particular, the server needs a key-switching matrix showing how to go from $sk^{(k)}$ to sk (see [108, Appendix D] for details), which the client must generate. Since in Expand substitution is called for different values of k (notice that in Line 10 and 11 in Figure 5.3 the value of k depends on j), the client must provide a key-switching matrix for each of them. However, this only needs to be done once and it depends only on the size of the database.

The above allows the server to compute Expand: the server first does the substitution followed by the appropriate key switch, and finally performs the addition in the inner loop.

A.2 Correctness of query expansion

Below we prove that Expand (Figure 5.3) correctly expands one ciphertext into a vector of n ciphertexts with the desired contents. The following theorem makes this formal.

Theorem A.2.1. Let N be a power of 2, $N \geq n$, and $query = Enc(x^i)$ be the client's encoding of index i. The n output ciphertexts $o_0, \ldots, o_{n-1}$ of $\text{Expand}(query)$ satisfy, for all $0 \leq k \leq n-1$:

$$o_k = \begin{cases} Enc(1) & \text{if } i = k \\ Enc(0) & \text{otherwise} \end{cases}$$

Proof. It suffices to prove the case for $n = 2^\ell$. For $j = \{0, 1, \ldots, \ell - 1\}$, we claim that

after the j^{th} iteration of the outer loop, we have *ciphertexts* = $[c'_0, \ldots, c'_{2^{j+1}-1}]$ such that

$$ciphertexts[k] = \begin{cases} Enc(2^{j+1}x^{i-k}) & \text{if } i \equiv k \pmod{2^{j+1}} \\ Enc(0) & \text{otherwise} \end{cases}$$

We prove the claim by induction on j. The base case $j = 0$ is explained in the main text of Section 5.6. Suppose the claim is true for some $j \geq 0$. Then in the next iteration, we compute an array *ciphertexts*′.

For the first half of the array, meaning for values of k where $0 \leq k < 2^{j+1}$, we have $ciphertexts'[k] = ciphertexts[k] + \mathsf{Sub}(ciphertexts[k], N/2^{j+1} + 1)$. If $i \neq k \pmod{2^{j+1}}$, then $ciphertexts'[k]$ is an encryption of 0; otherwise, there is an integer r such that $i-k = 2^{j+1} \cdot r$, and $\mathsf{Sub}(ciphertexts[k], N/2^{j+1}+1) = Enc(2^{j+1}x^{(N/2^{j+1}+1)(2^{j+1}r)}) = Enc(2^{j+1}(-1)^r x^{i-k})$. Hence, if r is odd, then $ciphertexts'[k]$ is an encryption of 0; otherwise, $ciphertexts'[k]$ is an encryption of $2^{j+2}x^{i-k}$. So the claim follows because r is even if and only if $i \equiv k \pmod{2^{j+2}}$.

We now prove the claim for the second half of the array *ciphertexts*′. The only interesting case is $i \equiv k - 2^{j+1} \pmod{2^{j+1}}$. In this case, we see that $ciphertexts'[k]$ is again $Enc(2^{j+1}(-1)^{(i-k)/2^{j+1}} x^{i-k})$. So the same argument applies.

Finally, with the above claim we show that after the outer loop in Expand, we have an array of 2^ℓ ciphertexts such that:

$$ciphertexts[k] = \begin{cases} Enc(2^{\ell}x^{i-k}) & \text{if } i \equiv k \pmod{2^{\ell}} \\ Enc(0) & \text{otherwise} \end{cases}$$

However, note that $i < n = 2^\ell$, so $i \equiv k \pmod{2^\ell}$ implies $i = k$. Hence $ciphertexts[k]$ is either an encryption of 0 or an encryption of 2^ℓ. To obtain an encryption of 0 or 1, we multiply $ciphertexts[k]$ by the inverse of 2^ℓ modulo t in the last step (Figure 5.3, Line 16). □

A.3 Noise growth of query expansion

One advantage of our query expansion technique over the straw man FHE solution given in Section 5.3 (besides the one mentioned in that section) is that our approach has *much* smaller noise growth. We bound the noise growth of Expand (Figure 5.3) in the theorem below. Before stating the theorem, we give some background on noise. See the SEAL manual [65] for a more detailed explanation. We have that the noise of the addition of two ciphertexts is the sum of their individual noises. Plain multiplication by a monomial x^j (for some j) with coefficient 1 does not change the noise, and plain multiplication by a constant α multiplies the noise by α. Substitution adds a constant additive term B_{sub} to the noise, which depends on the FV parameters.

One interesting case is Figure 5.3, Line 9, in which the exponent of the monomial is negated (x^{-2^j}), and hence the coefficient is $-1 \equiv t - 1 \pmod{t}$ (rather than 1 as state above). Even in this case, the noise does not change. This is because SEAL adjusts the coefficients of the monomial to be mod q rather than mod t: $-1 \equiv t - 1 \pmod{t}$ becomes $q - 1 \pmod{q}$. When $q - 1$ multiplies the noise, the q-multiplies are irrelevant (mod q), so the absolute value of the noise stays the same.

Theorem A.3.1. Let v_{out} be the output noise of Expand, and v_{in} be the input noise. Let t denote the plaintext modulus in Expand, and let $k = \lceil \log(n) \rceil$. We have that

$$v_{out} \leq t \cdot \left(2^k (v_{in} + 2B_{sub})\right)$$

Proof. Let v_i be the noise after the i^{th} iteration in Expand (setting $v_0 = v_{in}$). Then $v_i = 2(v_{i-1} + B_{sub})$. Carrying out the sum, we get

$$v_k = 2^k v_0 + 2(2^k - 1)B_{sub} < 2^k (v_0 + 2B_{sub})$$

Since inverse $\leq t$, the final plain multiplication results in $v_{out} \leq t v_k$. This completes the proof. □

Appendix B

Two-choice hashing and batch code hybrid PBC

As part of mPIR, we also implement 4 other PBC constructions (besides the one described in Chapter 6.5) based on reverse hashing (§6.4) with different allocation algorithms: replication, single-hashing, two-choice hashing, hybrid of two-choice hashing and the subcube batch code [127]. The scheme that we use as a baseline in Section 8.4.3 is the hybrid scheme.

The replication PBC simply creates k buckets and places a replica of each element in each bucket. This is the most naive batch code and achieves no computational amortization, but also incurs no network overhead (since it is equivalent to the baseline which is performing k parallel instances of PIR).

The single-hashing PBC uses a single hash function to map elements to k buckets. Since this is the standard balls-and-bins scenario considered in the literature, we can bound the number of tuples that fall in any bucket (the max load) by $\frac{3\ln(k)}{\ln(\ln(k))}$ [167, Lemma 5.1]; this bound fails to hold with probability $\leq \frac{1}{k}$. When using this PBC with PIR, we configure clients to always issue $\frac{3\ln(k)}{\ln(\ln(k))}$ queries to each bucket to ensure that they can retrieve all of their messages with high probability.

The two-choice hashing PBC is similar to the single-hashing PBC, but it applies the finding of Azar et al. [30]: in a k balls and k bins scenario, if each ball maps to w random bins ($w > 1$), and balls are placed in the bin least full, the highest load in any bin is bounded by $\frac{\ln(\ln(k))}{\ln(w)} + \Theta(1)$ with high probability. In other words, by using two hash functions instead of one, the bound on the max load decreases exponen-

batch size (k)	max CPU gains	network overhead	scheme
4	1.78×	9×	$(n, \frac{9}{4}n, 4, 9)$-subcube
16	4.21×	9×	hybrid
32	8.41×	9×	hybrid
64	14.2×	9×	hybrid
128	28.4×	9×	hybrid
256	56.9×	9×	hybrid

Figure B.1: Theoretical CPU gains and network overheads of the hybrid PBC-based multi-query PIR scheme over retrieving elements one by one using PIR. For $k = 4$, the hybrid scheme defaults to a single bucket encoded with a $(n, \frac{9}{4}n, 4, 9)$-subcube batch code, since two-choice hashing yields no benefit for such small k.

tially. Again, when using this PBC with PIR, we configure clients to issue $\frac{\ln(\ln(k))}{\ln(2)} + 1$ queries to each bucket (we find the constant 1 to be sufficient).

Since the two-choice hashing PBC reduces the number of collisions on any bucket significantly (usually to 3 or 4), we consider a hybrid where we encode each of the buckets with a subcube batch code [127] that can handle the retrieval of any 4 elements (§6.5.2). This "hybrid PBC" construction achieves better computational amortization than the two-choice hashing PBC alone, albeit at a higher network cost.

Figure B.1 gives the maximum theoretical gains that can be achieved by using a multi-query PIR scheme based on the hybrid PBC (over k parallel instances of PIR). In all cases, the network overhead is 9× since the PBC splits up the collection into k buckets, and each bucket is encoded with a $(n_b, \frac{9}{4}n_b, 4, 9)$-subcube batch code (where n_b is the number of elements in each bucket). As a result, the total number of buckets is $9k$. Note that if one wishes to use a very large k ($k > 2980$), then one needs to use a subcube batch code that supports more than 4 queries. This is because for $k > 2980$, the expected number of collisions on any bucket after using two-choice hashing is greater than 4.

The next subcube batch code is the $(n_b, \frac{27}{16}n_b, 16, 27)$-subcube batch code which supports up to 16 queries, but introduces a network overhead of 27×. Hybridizing a two-choice hashing PBC with this subcube batch code results in a PBC that supports up to $k = e^{32,768}$, which is likely sufficient for most applications.

PBC	codewords (m)	buckets (b)
k-way replication	kn	k
Single hashing	$3n\ln(k)/\ln(\ln(k))$	$3k\ln(k)/\ln(\ln(k))$
Multi-choice hashing	$2n\rho$	$k\rho$
Multi-choice + subcube batch code hybrid	$2n \cdot ((\ell+1)/\ell)^{\log_2(\rho)}$	$k \cdot (\ell+1)^{\log_2(\rho)}$
Cuckoo hashing	$3n$	$1.5k$

Figure B.2: Parameters of the different PBC schemes described in Section B. When used to implement multi-query PIR, CPU costs are proportional to the number of codewords (m), and network costs are proportional to the number of buckets (b). The initial number of elements in the database (before encoding) is n. In all cases, k of the elements retrieved are useful, and the rest are overhead introduced by the PBC. For multi-choice hashing, $\rho = \ln(\ln(k))/\ln(w) + 1$, $w \geq 2$, and $\ell \geq 2$ (we use $d = 2$ and $\ell = 2$).

B.1 Understanding the costs of different PBCs

In the previous section we discuss four different PBC constructions, in addition to the Cuckoo PBC described in Section 6.5. In Figure B.2 we give a cost model that compares the different variants. This model suggests that replication, single-hashing, and multi-choice are simply not competitive with the other two schemes, yielding more codewords and buckets (which translate into higher computational and network costs when used for PIR). The two-choice hashing PBC hybridized with a subcube batch code has the ability to produce fewer codewords (by setting the parameter ℓ to a larger value) than the Cuckoo hashing PBC (§6.5), but it results in many more buckets. If the goal is to achieve good computation and reasonable network costs, then a good choice is to set $\ell = 2$.

To better understand how the Hybrid PBC performs in practice versus the Cuckoo PBC, we repeat the experiment in Section 8.3. Figure B.3 gives the results. We find that mPIR-Cuckoo does a better job than the mPIR-Hybrid at amortizing CPU costs across all batch sizes. This is a direct effect of the Cuckoo PBC producing fewer total codewords (see Figure 6.1), since computational costs are proportional to the number of elements after encoding (m). At $k = 256$ and 288-byte elements, mPIR-Cuckoo achieves a 2.6× reduction in CPU cost for the server when answering

	single-query	mPIR-Hybrid			mPIR-Cuckoo		
batch size (k)	1	16	64	256	16	64	256
client CPU costs (ms)							
MultiQuery	3.07	29.03	28.50	28.58	6.45	5.26	4.92
MultiExtract	2.51	20.00	16.27	16.36	3.26	3.25	2.70
server CPU costs (sec)							
MultiSetup	6.1	2.02	0.64	0.30	1.50	0.38	0.12
MultiAnswer	3.24	1.37	0.49	0.21	0.69	0.23	0.08
network costs (KB)							
query	64	577	577	577	96	96	96
answer	384	2,885	2,308	2,308	480	480	384

Figure B.3: Per-request (amortized) CPU and network costs of two multi-query PIR schemes on a database consisting of 2^{20} elements, with varying batch sizes. The schemes are mPIR-Hybrid, which uses a two-choice hashing PBC (§B) combined with a subcube batch code [127], and mPIR-Cuckoo, which is based on the Cuckoo hashing PBC described in Section 6.5. The second column gives the cost of retrieving a single element (no amortization). The underlying PIR library is SealPIR with $t = 2^{20}$ and elements are 288 bytes.

queries over the mPIR-Hybrid. Over running k parallel instances of PIR, the per-request CPU cost of mPIR-Cuckoo is 40.5× lower.

The difference in network costs between mPIR-Hybrid and mPIR-Cuckoo is more pronounced. This owes to the hybrid PBC building on the subcube batch code of Ishai et al. [127] which creates a large number of buckets (see Figure 6.1); to preserve privacy, clients must issue a PIR query to each bucket. In terms of concrete savings, mPIR-Cuckoo is 6× more network efficient (upload and download) than mPIR-Hybrid.

The last axis on which to compare these two PBCs is failure probability. We find that mPIR-Cuckoo also has a lower failure probability (2^{-40} to 2^{-20} depending on k, compared to mPIR-Hybrid's 2^{-20} to 2^{-16}). This suggests that mPIR-Cuckoo is superior on all axis, which justifies our use of this PBC as the default in Pung. We discuss how we measure the failure probability of the Hybrid PBC below.

k	bucket collisions (99th-percentile, max)	failure rate
4	4, 4	0
16	3, 5	1.1×10^{-5}
32	3, 6	6.2×10^{-6}
64	3, 5	2.4×10^{-6}
128	3, 6	2.2×10^{-6}
256	3, 5	1.0×10^{-6}

Figure B.4: Probability of failure for the Hybrid PBC. The number of bucket collisions after 10 million runs when clients derive 2 labels for each tuple is typically 3 (for $k > 4$), but can be up to 6 with small probability. More than 4 collisions in a bucket prevents a client from retrieving all k tuples. The last column quantifies the probability of this event.

B.2 Probability of failure for hybrid PBC

Recall that PBCs are probabilistically complete. That is, there exists some small probability that a client will be unable to retrieve all of its desired elements (§6.6.2). We now *empirically* quantify this probability. To do this we run an experiment where a client derives k pairs of labels randomly. We then count the number of bucket collisions (that is, the maximum load on each bucket) when the client uses the following greedy algorithm. For each pair of labels, the client chooses the label within the pair that maps to the bucket least full. The client then places the label in its matching bucket, and continues with the next pair of labels. The client processes the pairs of labels sequentially. At the end, we count the number of labels in each bucket. We repeat this experiment 10 million times; the results are in Figure B.4.

We find that the probability that a client fails to retrieve at least one item is relatively small: 1 in 100K for small k; even lower for larger k. While this is high enough that it will occasionally affect clients, recall from Section 6.6.2 that clients can simply retry again. Critically, this is not an *unexpected failure*: since the client knows *before* issuing the multi-retrieval request that it cannot retrieve one or more of its elements.

Appendix C

Pung's security analysis

Section 3.1 provides an informal description of metadata privacy. We now expand on this description. In particular we target *relationship unobservability* (UO) [179], which states that aside from the sender and the recipient, no other party can observe the existence of a communication. To formalize UO, we use and (minimally) extend the framework and security games presented by Gelernter and Herzberg [106] (other works provide related frameworks [31, 202]). We cannot use their framework directly since it assumes that messages that are sent are automatically received by users. In contrast, users in Pung fetch messages *explicitly*, possibly over many rounds (§4.3).

We start by introducing an abstract protocol π that models communication through a round-based mailbox service. π exposes four functions: $\textsc{Init}(\cdot)$, $\textsc{Round}(r_{in})$, $\textsc{Send}(i, j, m, r)$, and $\textsc{Retr}(i, j, num, r)$. Init initializes the state of the protocol and loads the shared keys between every pair of clients that intends to communicate (that is, we assume that the system has been securely bootstrapped). $\textsc{Round}(r_{in})$ takes as input an untrusted round of communication r_{in}, and outputs a client-local r_{local} round to be used by the subsequent Send and Retr procedures. Send takes the sender's id i, the recipient's id j, a plaintext message m ($\perp$ if the sender is idle), and the client-local round of communication r; it outputs a key-value pair that is sent to the mailbox service. Retr generates a retrieval request for a message sent to j by i during the client-local round r. *num* is an untrusted value that Retr uses to construct the retrieval request.

C.1 UO under explicit retrieval

We define our privacy notion, *relationship unobservability under explicit retrieval* (UO-ER), based on the following security game.

Security Game for UO-ER. The game consists of a setup-simulation-guess protocol played by a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. An instance of the game,

$$G^{b}_{\mathcal{A},\pi,n,t}(1^{\lambda}) = b'$$

is parameterized by the actions of the adversary, $\mathcal{A}$; the abstract protocol, π; the number of correct users n; the security parameter, λ; the number of rounds for which π runs, t; and the correct output of the game, b. The actual output of the game is the adversary's guess b'. The adversary wins the game if his guess is correct: $b' = b$.

Outline. The game is a standard indistinguishability game. In the *setup* phase, $\mathcal{A}$ specifies two scenarios, M^0 and M^1, that describe the behavior of users: what round they use, what messages they send (and to whom), and what messages they retrieve (and from whom). In the *simulation* step $\mathcal{C}$ chooses one scenario at random (M^b) and simulates the actions of correct users under this scenario. $\mathcal{C}$ does so by translating all round, send, and retrieve instructions in M^b to the corresponding Round, Send and Retr functions in π, and providing all the resulting application-layer packets to $\mathcal{A}$. At the end of the simulation, $\mathcal{A}$ chooses to either move forward onto the *guess* step and issue an answer or it can ask for the setup and simulation steps to be rerun from scratch (meaning that $\mathcal{A}$ specifies two new scenarios M^0 and M^1, and $\mathcal{C}$ simulates the new scenario M^b). If after a polynomial number of iterations $\mathcal{A}$ has not issued a guess, the default value of $b' = 0$ is assigned to the game's output.

Setup. $\mathcal{A}$ specifies two scenarios, M^0 and M^1, each containing a total of $n \cdot t$ entries. Each entry corresponds to a tuple of two actions: *send*, and *retr*. These actions are performed by a correct user during a round of the protocol. The values for these actions for user i in scenario b during simulation round r are: $M^b[i,r].send = \{i, j^b, m^b_{i\to j}\}$

and $M^b[i,r].retr = \{i, j^b, r_R^b\}$. i, j^b are the ids of users communicating in scenario b (not necessarily distinct); $m_{i\to j}^b$ is the plaintext message sent from i to j^b, which could be $\perp$ to indicate that the user does no send a message in round r of scenario b; and r_R^b is the round from which to retrieve a message (recall that clients can also retrieve messages from prior rounds). To model clients sending and retrieving k messages per round, one can extend the scenarios to contain $n \cdot k \cdot t$ entries. Each client can then be thought of as k logical clients; $\mathcal{A}$ can thus distinguish between M^0 and M^1 if it can distinguish the communication between any of the k logical clients, which is the expected behavior.

There are three restrictions on the ability of $\mathcal{A}$ to construct scenarios. First, both scenarios have the same number of entries describing the actions of each user in every round. Furthermore, we assume that each user is the recipient of at most k messages in any given round. This means that the attack described in Section 4.6 is out of the scope of this model. This is reasonable since k can be made large enough to be higher than the maximum number of friends that any user can have (recall that strangers cannot send messages to users in Pung since retrieving a message requires pre-sharing a secret). We also assume that clients have already bootstrapped their communication and have agreed out-of-band on a schedule to use to communicate (this schedule is captured by the chosen scenario M^b).

The second restriction on how $\mathcal{A}$ constructs the scenarios is that M^0 and M^1 describe only the actions of *correct* users. This is because malicious users do not follow π, and cannot be simulated by $\mathcal{C}$. The last restriction is that if scenario M^b has an honest user i sending or retrieving a message from a compromised user j during round r, then i must also send or retrieve a message from j during round r in scenario M^{1-b} (we have no such restriction when both i and j are honest). This restriction is consistent with our goal of relationship unobservability, which provides meaningful privacy only if both the sender and the recipient of a given message are correct (§3). A similar approach is taken by Gelernter and Herzberg [106].

Simulation. $\mathcal{A}$ provides the two scenarios that it generates to $\mathcal{C}$. If this is the first iteration, $\mathcal{C}$ flips a random coin and obtains bit b. Otherwise, $\mathcal{C}$ continues to use

```
1:  function SIMULATE(𝒜, π, n, t, M^b)
2:      requests ← []                                  // requests generated by clients
3:      responses ← []                                 // responses received by clients
4:      for i = 0 to n − 1 do
5:          π_i ← π.INIT(·)                            // setup an instance of π for user i
6:      for r = 0 to t − 1 do
7:          for i = 0 to n − 1 do
8:              round_i ← 𝒜.GetRound(i, r)
9:              rlocal_i ← π_i.ROUND(round_i)
10:             requests ←(insert)— π_i.SEND(M^b[i, r].send, rlocal_i)
11:         for i = 0 to n − 1 do
12:             num_i ← 𝒜.GetNumMessages(i, r, requests)
13:             req_i ← π_i.RETR(M^b[i, r].retr, num_i)
14:             requests ←(insert)— req_i
15:             responses ←(insert)— 𝒜.GetResponse(i, r, req_i)
16:     return (requests, responses)
```

Figure C.1: Simulation performed by challenger $\mathcal{C}$. $\mathcal{A}$ is the adversary's algorithm; π is an explicit retrieval protocol; M^b is the scenario to simulate; n is the number of correct users in the scenario; and t is the total number of rounds for which to run π.

the previously derived bit b. $\mathcal{C}$ then chooses scenario M^b and follows the protocol in Figure C.1. Note that $\mathcal{C}$ calls $\mathcal{A}$ as an oracle through three functions: *GetRound*, *GetNumMessages*, and *GetResponses*. $\mathcal{A}$ can return arbitrary values for these functions; this allows $\mathcal{A}$ to drop, reorder, replay, and insert messages (by adding or removing tuples or storing them in a different order), modify messages from correct users (by returning bogus results for *GetResponses*), and force clients out of sync (by returning different values for *GetRound*). During the oracle calls to $\mathcal{A}$.*GetNumMessages* and $\mathcal{A}$.*GetResponses*, $\mathcal{C}$ exposes the requests that clients generate to $\mathcal{A}$. These requests (and the corresponding responses) are also given to $\mathcal{A}$ at the end of the simulation.

Once the simulation is over, $\mathcal{A}$ can either issue an answer or ask for a rerun with new scenarios (but b is kept unchanged). This allows $\mathcal{A}$ to adapt its strategy across iterations. This process can repeat a number of times that is polynomial in the security parameter λ, after which the game automatically outputs 0 as $\mathcal{A}$'s guess.

Guess. $\mathcal{A}$ outputs a guess b' indicating that scenario $M^{b'}$ was simulated. $\mathcal{A}$ wins the game if it guesses $b' = b$.

Definition C.1.1. Protocol π provides UO-ER if given security parameter λ, for all probabilistic polynomial time algorithms $\mathcal{A}$, for any polynomial number of rounds t and correct users n, there exists a negligible function negl such that:

$$|\Pr[G^0_{\mathcal{A},\pi,n,t}(1^\lambda)=1] - \Pr[G^1_{\mathcal{A},\pi,n,t}(1^\lambda)=1]| \leq \text{negl}(1^\lambda)$$

where the probability is define over the random coins of $\mathcal{C}$. This definition states that if π provides UO-ER, then an adversary gains no meaningful advantage from observing network packets. In other words, the probability of $\mathcal{A}$ distinguishing between Alice communicating with Bob, and Alice communicating with Charlie (or not communicating at all) is negligibly better than a random guess (or any prior $\mathcal{A}$ may have obtained through other channels).

Theorem C.1.1. Pung provides UO-ER (Definition C.1.1).

Proof. Recall that Pung uses three primitives (§3.2): authenticated encryption (AE),

PRF, and a computational private information retrieval scheme (CPIR). Each has a notion of indistinguishability, which we leverage in a series of hybrid games (we summarize CPIR's indistinguishability game in Section C.2). We discuss the hybrid games below.

- **Game 0** is the original game $G^b_{\mathcal{A},\pi,n,t}(1^\lambda)$, where π = Pung. In this game Pung's ROUND procedure for a client i during simulation round r takes as input an untrusted value $round_i$ (provided by $\mathcal{A}$) and outputs $rlocal^b_i$. As we describe in Section 4.1, Pung monotonically increases the client's local round ($rlocal^b_i$, initially set to 0) and sets it to $max(rlocal^b_i + 1, round_i)$, which prevents round re-use.
  Pung's SEND procedure for a client i during simulation round r takes as input $(i, j^b, m^b_{i\to j}) = M^b[i, r].send$ and $rlocal^b_i$; it outputs a $(label_s, c)$ tuple. $label_s$ is computed using a PRF keyed with a uniformly random key k_L shared between client i and j^b (generated during INIT), $label_s = PRF_{k_L}(rlocal^b_i || j^b)$, where j^b is the recipient specified in scenario M^b. The ciphertext c is computed as $c = AE_{k_E}(m^b_{i\to j}, rlocal^b_i)$, where k_E is a uniformly random key shared between i and j^b (also generated during INIT), $rlocal^b_i$ is used as a non-secret nonce, and $m^b_{i\to j}$ is the plaintext message specified in scenario M^b.
  Pung's RETR procedure for a client i during round r takes as input $(i, j^b, r^b_R) = M^b[i, r].retr$, and num, which is an untrusted value supplied by $\mathcal{A}$. Pung's RETR procedure outputs a PIR query $q = \text{QUERY}(label_r, num)$. The label is computed as $label_r = PRF_{k_L}(r^b_R || i)$, where k_L is the same random key shared between i and j^b that was used during the SEND procedure, and r^b_R is the specified round from which to retrieve the message in scenario M^b (possibly in the past). Pung uses this label as input to a PIR-by-keywords scheme [69] (for example, BST-RETRIEVAL in Figure 4.4) to produce $q = \text{QUERY}(label_r, num)$.
- **Game 1** is the same as Game 0 except that $label_s$, which is produced by SEND, is the output of a truly random function evaluated on input $rlocal^b || j^b$.
- **Game 2** is the same as Game 1 except that the ciphertext c which is produced by SEND is an encryption of a random message (of the same size as $m^b_{i\to j}$) using AE with a random nonce and key K_E.

- **Game 3** is the same as Game 2 except that the RETR procedure uses a $label_r$ which is generated uniformly at random.

Let S_0 be the event that $b = b'$ in Game 0, where M^b is the chosen scenario and b' is $\mathcal{A}$'s guess. Similarly, let S_1 be the event that $b = b'$ in Game 1, S_2 be the event that $b = b'$ in Game 2, and S_3 be the event that $b = b'$ in Game 3.

Lemma C.1.1. $\Pr[S_3] = 1/2$.

Proof. Observe that when $\mathcal{C}$ and $\mathcal{A}$ play Game 3, all the requests and responses produced by $\mathcal{C}$ are independent of b: SEND produces a tuple of a random label and an encryption of a random message using a random nonce, and RETR issues a PIR query for a random label. As a result, the ability of $\mathcal{A}$ to correctly guess b after inspecting all requests and responses is no better than a random coin flip. □

Lemma C.1.2. $|\Pr[S_2] - \Pr[S_3]| \leq \epsilon_{CPIR}$, where ϵ_{CPIR} is the advantage of an efficient algorithm that distinguishes CPIR queries.

Proof. Observe that in Game 2, $\mathcal{C}$ outputs a CPIR query using $label_r$ (which depends on b, and which $\mathcal{A}$ might have seen from some other client during the send phase), whereas in Game 3, $\mathcal{C}$ outputs a CPIR query using a random label. Given the assumption that the CPIR scheme is secure, the advantage of an efficient algorithm to distinguish which of the two labels is requested is ϵ_{CPIR}, which is negligible. □

Lemma C.1.3. $|\Pr[S_1] - \Pr[S_2]| \leq \epsilon_{AE}$, where ϵ_{AE} is the advantage of an efficient algorithm that distinguishes between two messages encrypted with an AE scheme.

Proof. Observe that in Game 1, $\mathcal{C}$ outputs an encryption of $m^b_{i\to j}$ under AE_{k_E} using $rlocal^b$ as a non-secret nonce, whereas in Game 2, $\mathcal{C}$ outputs an encryption of a random message of the same size under AE_{k_E} using a random nonce. Given that Pung's ROUND procedure guarantees that rounds are never repeated and hence the nonce is unique, and given the assumption that the AE scheme is IND-CCA2 and the key k_E is secret, the advantage of an efficient algorithm to distinguish which of these two encryptions is outputted is ϵ_{AE}, which is negligible. Note that if during a particular round r, M^b specifies the communication between an honest client and a malicious

client (and hence $\mathcal{A}$ knows k_E, violating the secrecy assumption) this gives no information to $\mathcal{A}$ about b: recall from the game's setup that any communication between an honest and a compromised client is the same in both M^b and M^{1-b}. □

Lemma C.1.4. $|\Pr[S_0] - \Pr[S_1]| \leq \epsilon_{PRF}$, where ϵ_{PRF} is the advantage of an efficient algorithm that distinguishes the output of a pseudorandom function from that of a truly random function.

Proof. Observe that in Game 0, $\mathcal{C}$ outputs a $label_s$ which is a PRF on an input that depends on b whereas Game 1 outputs a random label that is independent of b. By definition, the advantage of an efficient algorithm to distinguish the output of a secure PRF on a given input from the output of a truly random function is ϵ_{PRF}, which is negligible. □

By combining Lemmas C.1.1, C.1.2, C.1.3, and C.1.4, we see that:

$$|\Pr[S_0] - 1/2| \leq \epsilon_{PRF} + \epsilon_{AE} + \epsilon_{CPIR}$$

and this is negligible. This completes the proof of Theorem C.1.1. □

C.2 Security of multi-query PIR

We now show that the multi-query PIR scheme (§6.6) retains the privacy guarantees provided by the underlying PIR scheme. In conjunction with Theorem C.1.1, this shows that Pung with mPIR continues to provide UO-ER (Definition C.1.1).

Security game for PIR. We use a standard computationally indistinguishability game for PIR [227, §2.3]. We describe it here for completeness. The game is between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. We denote an instance of this game by

$$PG^b_{\mathcal{A},\pi,n}(1^\lambda) = b'$$

where $\pi = (\textsc{Query}, \textsc{Answer}, \textsc{Extract})$ is a PIR protocol, and λ is the security parameter.

1. $\mathcal{A}$ selects a collection DB consisting of n items of equal length, and sends n to $\mathcal{C}$.
2. $\mathcal{A}$ then specifies two indices, i and j, to $\mathcal{C}$ such that $0 \leq i, j < n$.
3. $\mathcal{C}$ then generates a random bit $b \in_R \{0, 1\}$; it selects i if b is 0 and j otherwise.
4. $\mathcal{C}$ then generates a PIR query for the chosen index using π's Query procedure and sends them to $\mathcal{A}$.
5. $\mathcal{A}$ then outputs its guess of $\mathcal{C}$'s choice, $b' \in \{0, 1\}$.

The adversary $\mathcal{A}$ wins the security game if $b = b'$.

Definition C.2.1. A PIR protocol π is secure if for all probabilistic polynomial time adversaries $\mathcal{A}$ and DB sizes n:

$$|\Pr[PG^0_{\mathcal{A},\pi,n}(1^\lambda)=1] - \Pr[PG^1_{\mathcal{A},\pi,n}(1^\lambda)=1]| \leq \text{negl}(1^\lambda)$$

where λ is a security parameter and negl is a negligible function of the security parameter λ.

Assumption 1. We denote the PIR protocol used by Pung as π_{cpir}. We take it as a given that π_{cpir} is secure under Definition C.2.1.

Multi-retrieval. We now formalize multi-query PIR and provide the corresponding security game. A multi-query PIR protocol π' works over m collections $(DB_1, \ldots, DB_m)$ held by a server, where the ith collection, DB_i, consists of n_i elements of equal size. We denote the size of the server's collection by $N = (n_1, \ldots, n_m)$. Similar to a PIR protocol, π' supports three procedures: Multi-Query, Multi-Answer, Multi-Extract.

The Multi-Query(I, N) procedure is run by the client; I is a vector of length k, and encodes the indices of items that the client is interested in retrieving from the server's collections. Each entry in this vector is a tuple, (ℓ, p), where ℓ specifies the collection ($1 \leq \ell \leq m$) and p states the index in the ℓth collection ($1 \leq p \leq n_\ell$). Multi-Query outputs a set of queries Q that encodes theses indices. The Multi-Answer$(Q, (DB_1, \ldots, DB_m))$ procedure is run by the server; it returns a vector of encrypted responses A, where each entry encodes the element requested by the client at the corresponding position in Q. The Multi-Extract(A) procedure is

run by the client; it decrypts each entry in A to recover the desired element in each of the collections $(DB_1, \ldots, DB_m)$.

Security game for multi-query PIR. We extend PIR's security game and Definition C.2.1 to the case of PIR retrievals from multiple collections.[1] Our generalization to multiple collections is similar in spirit to indistinguishable multiple encryptions in the case of public key cryptosystems [129, Chapter 10]. We denote an instance of such a generalized game by

$$MG^{b}_{\mathcal{A},\pi',N,k}(1^{\lambda}) = b'$$

where π'=(Multi-Query, Multi-Answer, Multi-Extract) is a multi-query PIR protocol, and λ is the security parameter.

1. $\mathcal{A}$ generates m collections where the ith collection contains n_i elements of the same size; it then communicates $N = (n_1, \ldots, n_m)$ to the challenger $\mathcal{C}$.
2. $\mathcal{A}$ specifies two vectors of length k, I and J. Each entry in these vectors is a tuple (ℓ, p) where ℓ selects one of the collections ($1 \leq \ell \leq m$) and p selects one index from the ℓ^{th} database ($1 \leq p \leq n_\ell$). Additionally, we require that in both vectors, the number of retrievals from a particular collection must the same.
3. $\mathcal{C}$ then flips a random coin $b \in_R \{0, 1\}$ to select one of the two vectors (as in the single retrieval game). It then generates PIR queries, Q, using Multi-Query and sends them to $\mathcal{A}$.
4. $\mathcal{A}$ then outputs its guess of $\mathcal{C}$'s coin, b'.

The adversary $\mathcal{A}$ wins the security game if $b = b'$.

Definition C.2.2. A multi-query PIR protocol, π', is secure if for all probabilistic polynomial adversaries $\mathcal{A}$,

$$|\Pr[MG^{0}_{\mathcal{A},\pi',N,k}(1^{\lambda}) = 1] - \Pr[MG^{1}_{\mathcal{A},\pi',N,k}(1^{\lambda}) = 1]| \leq negl(1^{\lambda})$$

where λ is the security parameter and negl is a negligible function of the security

[1] This is different from multi-database PIR, which is a term often associated with IT-PIR schemes.

parameter.

Observe that Pung's multi-query PIR protocol, including all of the variants presented in Chapter 6, adhere to the above formalism. For instance, in Pung's single-hashing PBC scheme (Appendix B), the server simply splits its collection (DB) into B sub-collections $DB_1, \ldots, DB_B$ (representing the buckets) using a static partitioning scheme; clients issue PIR queries to each sub-collection independently. Furthermore, the number of PIR queries that clients need to issue to a particular sub-collection is given to clients at the end of Pung's SEND phase (which occurs prior to clients issuing any PIR queries). Our hybrid scheme (Appendix B) simply splits each bucket (DB_i) into further sub-collections based on the batch code used.

We now show that Pung's multi-query PIR protocol is secure under Definition C.2.2 given Assumption 1.

Theorem C.2.1. Pung's multi-query PIR scheme (§6.6) is secure under Definition C.2.2.

Proof. Suppose Pung's multi-query PIR scheme is not secure; then there exists a probabilistic polynomial time adversary, $\mathcal{A}_M$, that violates Definition C.2.2. We can show that one can use such an adversary to construct another adversary, $\mathcal{A}_P$, that violates definition C.2.1. Since Pung's multi-query PIR scheme employs π_{cpir}, which is assumed secure under Definition C.2.1, this leads to a contradiction.

The reduction proof is identical to the proof of reduction from indistinguishable multiple encryptions to indistinguishable single encryption for a public key cryptosystem [129, Theorem 10.10]. □

Bibliography

[1] Bleep. http://www.bleep.pm.

[2] ChatSecure. https://chatsecure.org.

[3] Open Whisper Systems. https://whispersystems.org.

[4] Telegram. https://telegram.org.

[5] Dissent: Provably anonymous overlay. https://github.com/dedis/Dissent/tree/95f73, Apr. 2010.

[6] Rust-crypto. https://github.com/dagenix/rust-crypto/, 2016.

[7] Vuvuzela: Private messaging system that hides metadata. https://github.com/davidlazar/vuvuzela, Sept. 2016.

[8] Akamai state of the internet connectivity report. https://www.akamai.com/fr/fr/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf, May 2017.

[9] Opensignal state of mobile networks: Usa. https://opensignal.com/reports-data/national/data-2017-08-usa/report.pdf, Aug. 2017.

[10] Rust MetroHash. https://github.com/arthurprs/metrohash-rs, Dec. 2017.

[11] Simple encrypted arithmetic library — SEAL. https://sealcrypto.org, 2017.

[12] XPIR: NFLLWE security estimator. https://github.com/XPIR-team/XPIR/blob/master/crypto/NFLLWESecurityEstimator/NFLLWESecurityEstimator-README, June 2017.

[13] XPIR NFLParams. https://github.com/XPIR-team/XPIR/blob/master/crypto/NFLParams.cpp, June 2017.

[14] Criterion: Statistics-driven microbenchmarking in rust. https://github.com/japaric/criterion.rs, Mar. 2018.

[15] Internet providers with data caps. https://broadbandnow.com/internet-providers-with-data-caps, Jan. 2018.

[16] Ring: Safe, fast, small crypto using rust. https://github.com/briansmith/ring, Mar. 2018.

[17] M. Abdalla, M. Bellare, and G. Neven. Robust encryption. In *Proceedings of the Theory of Cryptography Conference (TCC)*, Feb. 2010.

[18] M. Abdalla, M. Bellare, and P. Rogaway. The oracle diffie-hellman assumptions and an analysis of DHIES. In *Proceedings of the RSA Conference Cryptographer' Track (CT-RSA)*, Apr. 2001.

[19] D. Agrawal and D. Kesdogan. Measuring anonymity: The disclosure attack. *IEEE Security & Privacy*, 1(6), Nov. 2003.

[20] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[21] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. https://github.com/xpir-team/xpir/, 2016.

[22] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3), Oct. 2015.

[23] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *Proceedings of the USENIX Security Symposium*, Aug. 2017.

[24] K. Ali, A. X. Liu, W. Wang, and M. Shahzad. Keystroke recognition using WiFi signals. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MOBICOM)*, Sept. 2015.

[25] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2018.

[26] S. Angel, D. Lazar, and I. Tzialla. What's a little leakage between friends? In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Aug. 2018.

[27] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[28] J. Angwin, C. Savage, J. Larson, H. Moltke, L. Poitras, and J. Risen. AT&T helped U.S. spy on Internet on a vast scale. `http://goo.gl/Jfsm18`, Aug. 2015. The New York Times.

[29] Y. Arbitman, M. Naor, and G. Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 2010.

[30] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1994.

[31] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi. AnoA: A framework for analyzing anonymous communication protocols. In *Proceedings of the IEEE Computer Security Foundations Symposium*, June 2013.

[32] J. Ball. GCHQ captured emails of journalists from top international media. `http://goo.gl/YzXnYK`, Jan. 2015. The Guardian.

[33] J. Bamford. Shady companies with ties to Israel wiretap the U.S. for the NSA. `http://goo.gl/bdi7w4`, Apr. 2012. Wired.

[34] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Journal of the ACM*, 59(2), 2012.

[35] A. Beimel and S. Dolev. Buses for anonymous message delivery. *Journal of Cryptology*, 16(1), Jan. 2003.

[36] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Nov. 2002.

[37] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2000.

[38] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-privacy in public-key encryption. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Dec. 2001.

[39] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Dec. 2000.

[40] M. Bellare and P. Rogaway. Minimizing the use of random oracles in authenticated encryption schemes. In *Proceedings of the International Conference on Information and Communication Security (ICICS)*, Nov. 1997.

[41] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Proceedings of the International Conference on Theory and Practice of Public Key Cryptography (PKC)*, Apr. 2006.

[42] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Sept. 2011.

[43] O. Berthold, H. Federrath, and S. Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobserbability*, July 2000.

[44] O. Berthold and H. Langos. Dummy traffic against long term intersection attacks. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, Mar. 2002.

[45] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), July 1970.

[46] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2001.

[47] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2015.

[48] E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can we access a database both locally and privately? In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.

[49] E. Boyle, Y. Ishai, and A. Polychroniadou. Limits of practical sublinear secure computation. Cryptology ePrint Archive, Report 2018/571, June 2018. `http://eprint.iacr.org/2018/571.pdf`.

[50] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, Jan. 2012.

[51] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 2011.

[52] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2011.

[53] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen. Horton tables: Fast hash tables for in-memory data-intensive computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2016.

[54] S. Buttar. Dragnet NSA spying survives: 2015 in review. `https://goo.gl/JsNgS7`, Dec. 2015. Electronic Frontier Foundantion.

[55] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[56] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.

[57] J. A. Cain, P. Sanders, and N. Wormald. The random graph threshold for *k*-orientability and a fast algorithm for optimal multiple-choice allocation. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Jan. 2007.

[58] R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.

[59] Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *Proceedings of the Australasian Conference on Information Security and Privacy*, July 2004.

[60] D. Chaum, F. Javani, A. Kate, A. Krasnova, J. de Ruiter, and A. T. Sherman. cMix: Anonymization by high-performance scalable mixing. Cryptology ePrint Archive, Report 2016/008, Jan. 2016. `http://eprint.iacr.org/2016/008.pdf`.

[61] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

[62] D. L. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1), 1988.

[63] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. HORNET: High-speed onion routing at the network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2015.

[64] C. Chen, D. E. Asoni, A. Perrig, D. Barrera, G. Danezis, and C. Troncoso. TARANET: Traffic-analysis resistant anonymity at the network layer. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, Apr. 2018.

[65] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine. Simple encrypted arithmetic library v2.3.0-4. `https://sealcrypto.org`, Dec. 2017.

[66] H. Chen, K. Laine, and P. Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2017.

[67] L. T. Chen and D. Rotem. Optimal reponse time retrieval of replicated data. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, May 1994.

[68] R. Cheng, W. Scott, B. Parno, I. Zhang, A. Krishnamurthy, and T. Anderson. Talek: a private publish-subscribe protocol. Technical Report UW-CSE-16-11-01, University of Washington Computer Science and Engineering, Nov. 2016.

[69] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, Feb. 1998. `http://eprint.iacr.org/1998/003`.

[70] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1995.

[71] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobserbability*, July 2000.

[72] D. Cole. We kill people based on metadata. http://goo.gl/LWKQLx, May 2014. The New York Review of Books.

[73] T. Cook. A message to our customers. http://www.apple.com/customer-letter/, Feb. 2016.

[74] D. A. Cooper and K. P. Birman. Preserving privacy in a network of mobile computers. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 1995.

[75] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1987.

[76] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[77] H. Corrigan-Gibbs and B. Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2010.

[78] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford. Proactively accountable anonymous messaging in Verdict. In *Proceedings of the USENIX Security Symposium*, Aug. 2013.

[79] Council of Europe. European Convention on Human Rights: Article 8. http://www.echr.coe.int/Documents/Convention_ENG.pdf, Nov. 1950.

[80] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secures against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1), 2003.

[81] A. Czumaj and V. Stemann. Randomized allocation processes. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[82] G. Danezis. Statistical disclosure attacks. In *Proceedings of the IFIP Information Security Conference*, May 2003.

[83] G. Danezis, C. Diaz, and P. Syverson. Systems for anonymous communication. `https://securewww.esat.kuleuven.be/cosic/publications/article-1335.pdf`, Aug. 2009.

[84] G. Danezis, C. Diaz, and C. Troncoso. Two-sided statistical disclosure attack. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, June 2007.

[85] G. Danezis, C. Diaz, C. Troncoso, and B. Laurie. Drac: An architecture for anonymous low-volume communications. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2010.

[86] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2003.

[87] G. Danezis and A. Serjantov. Statistical disclosure or intersection attacks on anonymity systems. In *Proceedings of the International Workshop on Information Hiding*, May 2004.

[88] G. Danezis and C. Troncoso. Vida: How to use bayesian inference to de-anonymize persistent communications. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, Aug. 2009.

[89] D. Das, S. Meiser, E. Mohammadi, and A. Kate. Anonymity trilemma: Strong anonymity, low bandwidth, low latency—choose two. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2018.

[90] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, Nov. 2014.

[91] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu. PIR-PSI: Scaling private contact discovery. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2018.

[92] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[93] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.

[94] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 1990.

[95] R. Dingledine. Did the FBI pay a university to attack Tor users? `https://goo.gl/NB3hSR`, Nov. 2015. Tor Project.

[96] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[97] C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2014.

[98] J. R. Douceur. The sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[99] C. Egger, J. Schlumberger, C. Kruegel, and G. Vigna. Practical attacks against the I2P network. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Nov. 2013.

[100] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, Mar. 2012. `https://eprint.iacr.org/2012/144.pdf`.

[101] D. Fernholz and V. Ramachandran. The k-orientability thresholds for $G_{n,p}$. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Jan. 2007.

[102] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2002.

[103] A. Frieze, P. Melsted, and M. Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM Journal on Computing*, 40(2), Mar. 2011.

[104] J. Gardiner and S. Nagaraja. Blindspot: Indistinguishable anonymous communications. arXiv:1408/0784v2, Aug. 2014. `http://arxiv.org/abs/1408.0784`.

[105] S. Garg, E. Miles, P. Mukherjee, A. Sahai, A. Srinivasan, and M. Zhandry. Secure obfuscation in a weak multilinear map model. In *Proceedings of the Theory of Cryptography Conference (TCC)*, Oct. 2016.

[106] N. Gelernter and A. Herzberg. On the limits of provable anonymity. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Aug. 2013.

[107] N. Gelernter, A. Herzberg, and H. Leibowitz. Two cents for strong anonymity: The anonymous post-office protocol. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[108] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 2012.

[109] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 2005.

[110] I. Goldberg. Improving the robustness of private information retrieval. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2007.

[111] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4), Oct. 1986.

[112] S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1982.

[113] P. Golle and A. Juels. Dining cryptographers revisited. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2004.

[114] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 58(11), Nov. 2012.

[115] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.

[116] A. Greenberg. Whatsapp just switched on end-to-end encryption for hundreds of millions of users. `http://www.wired.com/2014/11/whatsapp-encrypted-messaging/`, Nov. 2014.

[117] G. Greenwald and R. Gallagher. New Zealand launched mass surveillance project while publicly denying it. `https://goo.gl/UwNpwV`, Sept. 2014. The Intercept.

[118] G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others. `http://goo.gl/qETWUq`, June 2013. The Guardian.

[119] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, May 2010.

[120] C. Gülcü and G. Tsudik. Mixing E-mail with Babel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 1996.

[121] E. Gün Sirer, S. Goel, M. Robson, and D. Engin. Eluding carnivores: File sharing with strong anonymity. In *Proceedings of the ACM SIGOPS European Workshop*, Sept. 2004.

[122] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[123] R. Henry. Polynomial batch codes for efficient IT-PIR. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[124] R. Henry, Y. Huang, and I. Goldberg. One (block) size fits all: PIR and SPIR with variable-length records via multi-block queries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2013.

[125] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.

[126] A. Houmansadr, C. Brubaker, and V. Shmatikov. The parrot is dead: Observing unobservable network communications. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[127] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, June 2004.

[128] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1997.

[129] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007.

[130] J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *Proceedings of the Fast Software Encryption Workshop (FSE)*, Apr. 2000.

[131] D. Kesdogan, D. Agrawal, V. Pham, and D. Rautenbach. Fundamental limits on the anonymity provided by the MIX technique. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2006.

[132] D. Kesdogan, J. Egner, and R. Büschkes. Stop-And-Go-MIXes providing probabilistic anonymity in an open system. In *Proceedings of the International Workshop on Information Hiding*, Apr. 1998.

[133] D. Kesdogan, D. Mölle, S. Ritchter, and P. Rossmanith. Breaking anonymity by learning a unique minimum hitting set. In *Proceedings of the International Computer Science Symposium in Russia (CSR)*, Aug. 2009.

[134] D. Kesdogan and L. Pimenidis. The hitting set attack on anonymity protocols. In *Proceedings of the International Workshop on Information Hiding*, May 2004.

[135] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, May 2007.

[136] M. Khosla. Balls into bins made faster. In *Proceedings of the European Symposium on Algorithms (ESA)*, Sept. 2013.

[137] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal rate private information retrieval from homomorphic encryption. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2015.

[138] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Journal of Random Structures and Algorithms*, 33(2), Sept. 2008.

[139] L. Kissner, A. Oprea, M. K. Reiter, D. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, June 2004.

[140] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi. Anonymity-preserving public-key encryption: A constructive approach. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2013.

[141] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2010.

[142] K. Kurosawa and Y. Desmedt. A new paradigm of hybrid encryption scheme. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2004.

[143] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[144] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[145] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2017.

[146] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[147] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[148] S. Le Blond, D. Choffnes, W. Caldwell, P. Druschel, and N. Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2015.

[149] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2013.

[150] R. Lenzner. ATT, Verizon, Sprint are paid cash by NSA for your private communications. `http://goo.gl/x7Czlm`, Sept. 2013. Forbes.

[151] B. N. Levine, M. K. Reiter, C. Wang, and M. Wright. Timing attacks in low-latency mix systems. In *Proceedings of the International Financial Cryptography Conference*, Feb. 2004.

[152] H. Lipmaa. First CPIR protocol with data-dependent computation. In *Proceedings of the International Conference on Information, Security and Cryptology (ICISC)*, Dec. 2009.

[153] H. Lipmaa and K. Pavlyk. A simpler rate-optimal CPIR protocol. In *Proceedings of the International Financial Cryptography Conference*, Apr. 2017.

[154] H. Lipmaa and V. Skachek. Linear batch codes. In *Proceedings of the International Castle Meeting on Coding Theory and Applications*, Sept. 2014.

[155] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the International Financial Cryptography and Data Security Conference*, Jan. 2015.

[156] N. Mallesh and M. Wright. The reverse statistical disclosure attack. In *Proceedings of the International Workshop on Information Hiding*, June 2010.

[157] N. Mathewson and R. Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, May 2004.

[158] J. Mayer, P. Mutchler, and J. C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 113(20), May 2016.

[159] D. A. McGrew and J. Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In *Proceedings of the International Conference on Cryptology in India (INDOCRYPT)*, Dec. 2004.

[160] F. McSherry. Timely dataflow. `https://github.com/frankmcsherry/timely-dataflow/`, 2016.

[161] J. Menn. Yahoo secretly scanned customer emails for U.S. intelligence. `https://goo.gl/KZuUYo`, Oct. 2016. Reuters.

[162] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are who you know: Inferring user profiles in online social networks. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, Feb. 2010.

[163] P. Mittal and N. Borisov. Information leaks in structured peer-to-peer anonymous communication systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.

[164] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2011.

[165] P. Mittal, M. Wright, and N. Borisov. Pisces: Anonymous communication using social networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2013.

[166] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), Oct. 2001.

[167] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Jan. 2005.

[168] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2005.

[169] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[170] A. Nambiar and M. Wright. Salsa: A structured approach to large-scale anonymity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2006.

[171] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1990.

[172] L. Nguyen and R. Safavi-Naini. Breaking and mending resilient mix-nets. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, Mar. 2003.

[173] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[174] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proceedings of the European Symposium on Algorithms (ESA)*, Aug. 2001.

[175] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[176] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2011.

[177] M. B. Paterson, D. R. Stinson, and R. Wei. Combinatorial batch codes. *Advances in Mathematics of Communications (AMC)*, 3(1), Feb. 2009.

[178] F. Pérez-González and C. Troncoso. Understanding statistical disclosure: A least squares approach. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2012.

[179] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. `http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf`, Aug. 2010.

[180] B. Pfitzmann. Breaking an efficient anonymous channel. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1995.

[181] B. Pfitzmann and A. Pfitzmann. How to break the direct RSA-implementation of mixes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 1989.

[182] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security*, 21(2), Jan. 2018.

[183] A. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The Loopix anonymity system. In *Proceedings of the USENIX Security Symposium*, Aug. 2017.

[184] A. Piotrowska, J. Hayes, N. Gelernter, G. Danezis, and A. Herzberg. AnNotify: A private notification service. Cryptology ePrint Archive, Report 2016/466, May 2016. `http://eprint.iacr.org/2016/466.pdf`.

[185] E. Protalinski. Facebook scans chats and posts for criminal activity. http://goo.gl/pfV9XE, July 2012. CNET.

[186] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 1991.

[187] L. Radaelli, P. Sapiezynski, F. Houssiau, E. Shmueli, and Y.-A. de Montjoye. Quantifying surveillance in the networked age: Node-based intrustions and group privacy. arXiv:1803/09007, Mar. 2018. http://arxiv.org/abs/1803.09007.

[188] A. S. Rawat, D. S. Papailiopoulos, A. G. Dimakis, and S. Vishwanath. Locality and availability in distributed storage. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, June 2014.

[189] A. S. Rawat, Z. Song, A. G. Dimakis, and A. Gál. Batch codes through dense graphs without short cycles. *IEEE Transactions on Information Theory*, 62(4), Apr. 2016.

[190] J.-F. Raymond. Traffic analaysis: Protocols, attacks, design issues, and open problems. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[191] J.-F. Raymond. Traffic analysis: Protocols, attacks, design issues and open problems. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[192] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1), Nov. 1998.

[193] M. Rennhard and B. Plattner. Introducing MorphMix: Peer-to-peer based anonymous Internet usage with collusion detection. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2002.

[194] R. L. Rivest. Chaffing and winnowing: Confidentiality without encryption. *CryptoBytes Technical Newsletter (RSA Laboratories)*, 4(1), July 1998.

[195] P. Rogaway. The moral character of cryptographic work. Cryptology ePrint Archive, Report 2015/1162, Dec. 2015. http://eprint.iacr.org/2015/1162.pdf.

[196] A. Rusbridger. The Snowden leaks and the public. http://goo.gl/VOQL86, Nov. 2013. The New York Review of Books.

[197] D. Rushe. Yahoo $250,000 daily fine over NSA data refusal was set to double 'every week'. http://goo.gl/FZGfTT, Sept. 2014. The Guardian.

[198] C. D. Salahub and W. Oldford. Interactive filter and display of Hillary Clinton's emails: A cautionary tale of metadata. https://www.researchgate.net/publication/315876309, Apr. 2017.

[199] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2005.

[200] B. Schneier. *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. W.W. Norton & Company, Mar. 2015.

[201] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. P5: A protocol for scalable anonymous communication. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2002.

[202] V. Shmatikov and M.-H. Wang. Measuring relationship anonymity in mix networks. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2006.

[203] V. Shmatikov and M.-H. Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2006.

[204] N. Silberstein. Fractional repetition and erasure batch codes. In *Proceedings of the International Castle Meeting on Coding Theory and Applications*, Sept. 2014.

[205] N. Silberstein and T. Etzion. Optimal fractional repetittion codes and fractional repetition batch codes. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, June 2015.

[206] A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2006.

[207] J. P. Stern. A new and efficient all-or-nothing disclosure of secrets protocol. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Oct. 1998.

[208] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4), Aug. 1969.

[209] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing attacks on privacy in Tor. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[210] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous connections and onion routing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 1997.

[211] K. Talwar and U. Wieder. Balanced allocations: the weighted case. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, June 2007.

[212] C. Troncoso, B. Gierlichs, B. Preneel, and I. Verbauwhede. Perfect matching disclosure attack. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2008.

[213] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2017.

[214] United Nations General Assembly. The Universal Declaration of Human Rights: Article 12. `http://www.un.org/en/universal-declaration-human-rights/`, Dec. 1948.

[215] United States Congress. Electronic Communications Privacy Act of 1986 (ECPA). `https://it.ojp.gov/privacyliberty/authorities/statutes/1285`, Oct. 1986.

[216] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[217] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4), 2003.

[218] M. Waidner and B. Pfitzmann. The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 1989.

[219] T. Wang and I. Goldberg. Improved website fingerprinting on Tor. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2013.

[220] Z. Wang, H. M. Kiah, and Y. Cassuto. Optimal binary switch codes with small query size. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, June 2015.

[221] Z. Wang, O. Shaked, Y. Cassuto, and J. Bruck. Codes for network switches. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, July 2013.

[222] S. Warren and L. Brandeis. The right to privacy. *Harvard Law Review*, 4(5), Dec. 1890.

[223] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. StegoTorus: A camouflage proxy for the Tor anonymity system. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2012.

[224] D. Wikström. Five practical attacks for "optimistic mixing for exit-polls". In *Proceedings of the Conference on Selected Areas in Cryptography (SAC)*, Aug. 2003.

[225] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[226] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Scalable anonymous group communication in the anytrust model. In *Proceedings of the European Workshop on System Security (EUROSEC)*, Apr. 2012.

[227] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5), May 2013.