EPFL

Thèse n° 7391

Integrity and Metadata Protection in Data Retrieval

Présentée le 26 novembre 2021

Faculté informatique et communications
Laboratoire de systèmes décentralisés et distribués
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Kirill NIKITIN

Acceptée sur proposition du jury

Prof. J.-P. Hubaux, président du jury
Prof. B. A. Ford, directeur de thèse
Prof. J. Cappos, rapporteur
Prof. S. Capkun, rapporteur
Prof. K. Argyraki, rapporteuse

École
polytechnique
fédérale
de Lausanne

2021

Моим родителям

Acknowledgements

During the six years of my PhD research, I had the great luck of being able to interact with and to learn from many amazing people: to learn concrete techniques and approaches for my research, to learn how to find good problems and how to handle unwanted ones, and to learn how to conduct research that I can be proud of or, at least, to not be ashamed of it. Of course, this learning would not have been sustainable or even possible without the support and kindness of these people around me. It is my pleasure to express to them the long-overdue gratitude for teaching me, showing me a good example, supporting me, or for simply being kind.

I begin by thanking Bryan Ford for being an advisor who let me explore various (often loosely connected) research topics and helped me enjoy my PhD years. Drawing from his research vision and intuition, I learned how to see the big picture and how to be optimistic about my ideas, well before others appreciate them. There was not a single time when he was too busy to have a chat with me or to give his advice on how to solve a problem that I was facing. I could always count on his support and his kindness. I would like to also thank Srinath Setty with whom I did an internship at Microsoft Research and later wrote a paper. Srinath, you showed me an example of how to conduct rigorous and goal-oriented systems research which applied to both methodology and writing. This shaped my perception of what great research should be. Now, I also sketch anticipated graphs with a pen and e-paper (as a reasonable alternative to ASCII-art graphs) before implementing evaluation scenarios, and I work out introduction before writing the rest of a paper. During my internships at Microsoft Research, I also was lucky to work with and learn from Santiago Zanella-Béguelin and Jonathan Lee. Henry Corrigan-Gibbs convinced me of the importance in choosing the right problem to work on and taught me how to separate “gold from sand” when making this choice. Henry, you are my research model and I strive to one day be as thorough and creative in my work as you are. I thank the jury of my oral exam, Katerina Argyraki, Srdjan Capkun, Justin Cappos, and Jean-Pierre Hubaux. They turned the exam into a joyful debate with insightful questions rather than a dreadful interrogation. Arjen Lenstra introduced me to research at EPFL by choosing me for the Summer@EPFL program and by later supporting my PhD application with his recommendation letter. I would not have reached this point without his endorsement.

DEDIS has been my research home during these last few years. I had numerous discussions, shared many wonderful lunches and dinners, and received feedback on my work countless times from present and former DEDIS members: Jeff Allen, Cristina Basescu, Pierluca Borsò,

Gaylor Bosson, Kelong Cong, Georgia Fragkouli, Nicolas Gailly, Linus Gasser, Philipp Jovanovic, Ismail Khoffi, Noémien Kocher, David Lazar, Stevens Le Blond, Louis-Henri Merino, Gaurav Narula, Antoine Rault, Pasindu Tennage, Haoqian Zhang, and others who passed via DEDIS. Thank you all, and I hope that we will stay in touch! I worked closely with Ludovic Barman and Simone Colombo. I thank you two for your hard work that also helped me to write this thesis and for bearing with me as a collaborator. The administrative assistants addressed all the bureaucratic hurdles for me and simplified my life at EFPL. Margaret Church, Angela Devenoge, Sandra Genolet, and Patricia Hjelt, you have been of invaluable help! Holly Cogliati-Bauereis edited countless pages of my work, including this thesis. Your guidance ~~allowed~~ enabled me to choose the right words and to improve my writing style. I am also thankful to the people of LDS and SPRING—our sister labs. In particular, from Wouter Lueks I learned that cryptographic proofs are, albeit magical, still a doable task. The research events and sport activities that I took part in, with David Froelicher, Mickaël Misbach, Christian Mouchet, and João Sá, were even more fun simply due to their presence.

Ceyhun Alp and Lefteris Kokoris-Kogias have been not only my lab colleagues but also dear friends. Cey, many times when we had to resolve problems and to complete challenging tasks, I never had even a tiny doubt that we would manage if we worked together. We shared spaces and experiences, visited countries, and tried various restaurants with Lefteris. I know that I can turn to you for advice and will always gain some new insight. Our “French club” meetings with Henry, Ludovic, and Sylvain Chatel gave us a chance to pair research discussions and life debriefings with delicious food. I hope that we will keep holding these meetings for many years to come. I could always count on the support and friendship of Iegor Rudnytskyi, and I hope that I have been a good friend to you too. Georgios Damaskinos, Patricia Egger, Lana Josipovic, Novak Kaluđerović, Marios Kogias, Hermina Petric Maretic, Helena Peic Tukuljac, and Irene Viola made my time outside research more uplifting and joyful. We spent many great evenings and weekends together skiing, playing games, barbecuing, and being ourselves. I am proud to be a friend and a collaborator of Leonardo Impett. It is always a pleasure to talk to you, and I am impressed by your erudition. Even though the areas of our research could not, at first glance, be further apart, we found a project that required the best from both of us. And it is inspiring to work with you since then. Dusan Kostic is the friend who understands me without words and whom I can unconditionally rely on. These six years would have been much more challenging without our conversations and mutual support.

В конце, я бы хотел выразить благодарность своим родным. Если бы у меня что-то до отчаяния не получалось в моей научной работе, то моя бабушка Лидия прочитала и разобралась бы во всех статьях, а потом помогла бы разобраться мне. Мои родители, Ольга и Анатолий, всегда поддерживали, помогали и гордились мной, что бы ни происходило. Без них я бы никогда не оказался там, где я сейчас. И за эту поддержку я безгранично благодарен.

Lausanne, August 17, 2021 K. N.

Abstract

Secure retrieval of data requires integrity, confidentially, transparency, and metadata-privacy of the process. Existing protection mechanisms, however, provide only partially these properties: encryption schemes still expose cleartext metadata, protocols for private information retrieval neglect data integrity, and data-distribution architectures forego transparency. In this dissertation, by designing new cryptographic primitives and security architectures that provide a more comprehensive protection, we improve on the current security and privacy practices in data retrieval. First, we propose a new format for encrypted data; it protects both content and all encryption metadata, such as the application, the intended recipients, and the algorithms used. The format comes with a cryptographically-agile encoding scheme that facilitates efficient decryption of such ciphertexts without cleartext markers. Second, to address the lack of integrity in privacy-preserving data-retrieval protocols, we introduce the concept of single-server verifiable private information retrieval. In contrast to existing solutions where, in some deployment scenarios, a malicious server can violate client privacy by selectively tampering with the data, our approach ensures that an honest client either correctly obtains the data from the system's server or detects server misbehavior and aborts. Finally, we present a software-update framework that reinforces software-distribution processes. Building on the concepts of decentralization and verifiability, our framework eliminates single points of failure, enforces transparency, and ensures integrity and authenticity of software releases. By implementing and experimentally evaluating our primitives and framework, we demonstrate that better protection is practical and incurs only a modest additional cost.

Keywords: privacy, integrity, security, metadata protection, private information retrieval, verifiable, software updates, transparency.

Résumé

Le téléchargement de données nécessite l'intégrité et la confidentialité des données obtenues, et la transparence et la protection des métadonnées du processus de téléchargement. Les systèmes existants ne garantissent que partiellement ces propriétés : par exemple, les processus de chiffrage exposent des métadonnées en clair, les protocoles de récupération privée de données[1] négligent l'intégrité de celles-ci, et les architectures de distribution de données ne fournissent pas de garantie de transparence quant aux données fournies. Dans cette dissertation, nous améliorons l'état de l'art concernant la sécurité et protection des données privées lors du téléchargement de données : pour ce faire, nous créons de nouvelles primitives cryptographiques et des architectures qui fournissent une protection plus globale. Premièrement, nous présentons un nouveau format pour les données chiffrées : il protège le contenu et les métadonnées (par exemple, les applications utilisées, les destinataires et les algorithmes utilisés) tout en garantissant des performances acceptables lors du déchiffrage pour les destinataires légitimes. Deuxièmement, pour palier au manque d'intégrité dans les protocoles de récupération privée de données, nous présentons une version monoserveur vérifiable. Contrairement aux alternatives actuelles, notre solution donne la garantie qu'un client obtient les données qu'il souhaite, ou qu'il détecte un problème d'intégrité et puisse agir en conséquence. Finalement, nous présentons une solution pour mises-à-jour logicielles qui améliore les processus actuels. Notre protocole utilise les concepts de décentralisation et de vérifiabilité publique et fourni une plus grande garantie en termes d'intégrité et d'authenticité pour les mises-à-jour logicielles. Nous implémentons et évaluons expérimentalement nos contributions et nous démontrons qu'il est possible d'obtenir une meilleure protection à un coût modeste.

Mots-clés : confidentialité, intégrité, sécurité, protection des métadonnées, récupération privée de données, vérifiable, mises à jour logicielles, transparence.

[1] Private information retrieval

Contents

List of Figures

List of Tables

1 Introduction

Data retrieval is a fundamental part of user interaction with the modern Internet. Consuming media, receiving e-mails, viewing Web pages, or installing software updates, essentially consist of users downloading data from remote servers and locally processing it. According to OpenVault, the average monthly download volume by broadband-Internet subscribers in the United States reached 344 GB in 2019 [144].

Due to its prevalence and direct effect on the users, the process of data retrieval becomes a lucrative attack-point for criminals, dishonest service providers, and oppressive state regimes. A hacker who watches network traffic in a coffee shop might be interested in banking information or personal messages that another shop's client receives. A mischievous service provider might seek to learn what content a subscriber consumes in order to sell this information to advertisement agencies. An oppressive regime could add a backdoor to a software update that a rights activist installs so that the regime gains the ability to spy on the activist.

The protection against such malicious actors comes primarily from the use of security and privacy mechanisms that researchers and practitioners have been developing for decades. Encryption schemes and protocols, such as OpenPGP [42] or TLS [159], ensure confidentiality of data in transit between communication parties. As of May, 2021, 72.6% of all websites on the Internet offer encrypted access to their users [191]. Private information retrieval [49], [115] is a mechanism that enables a user to retrieve data from a service provider without revealing which exact data she retrieves. It has become a building block in various (albeit mostly academic) privacy-preserving systems, e.g., in private video streaming [89], retrieval of security updates [43], and encrypted search [58]. Finally, the integrity and authenticity of data, i.e., that data have not been tampered with on the way to a user and that it indeed originate from a given content producer, respectively, are typically ensured with cryptographic signatures [68], [164] or message authentication codes [22]. The idea is that only the owner of a secret key can produce authentication information for a given piece of data, and other parties can use this information to validate the authenticity and integrity of the received data.

Unfortunately, malicious actors also continue to advance their techniques for exploiting mechanisms that previously not thought of requiring protection or that were only partially protected. Such an inventive approach represents a broad class of attacks, among which exploiting the metadata of communication [88], [129] and encryption [46], [95], [178], timing side-channels in cryptographic implementations [109],and information leakage in cloud computing [161]. In this dissertation, we are interested in how an adversary can exploit the metadata in encrypted communications. Specifically, by analyzing communication patterns and observing cleartext metadata of encrypted traffic, i.e., performing traffic analysis [56], an attacker might be able to identify users [72], [95], [148], [154], [162], the applications these users use [28], [192], the websites they visit [30], [32], [91], [147], [160], [183]–[185], and the actions they perform there [47], [132], [196], or to infer the exact length of their passwords [90], [180], and even to recover the content of their communication [48], [55], [167].

The standard security mechanisms can fail to ensure the end-to-end integrity and authenticity of data for users in some scenarios. The systems that distribute data to users often rely on a single signing key for data authentication, and such a key is prone to accidental exposure [100], [133] and theft [80], [157]. For example, in the context of software-update systems, an adversary who compromises a system's signing key gains the ability to present malicious, potentially harmful binaries as valid software updates that unsuspecting users will install [40], [80], [103], [195]. Furthermore, the typical lack of transparency and verifiability of a system could enable malicious [97], coerced [27], [79], [171], and/or compromised [44], [119] service providers to perform targeted attacks on their users, i.e., to maliciously alter the data before serving it to targeted users. As we demonstrate in Chapter 3, such data altering can enable a service provider to not only feed incorrect or malicious data to a user but also to use this for breaking the privacy of that user's choices in the context of private information retrieval.

1.1 Overview of Results

In this dissertation, we improve the status-quo in the security and privacy of data retrieval by designing new cryptographic primitives and building data-distribution systems with improved protection guarantees. Our contributions target different stages of data retrieval, hence several attacker types. In Chapter 2, we harden the protection against on-the-network eavesdroppers, such as coffee-shop hackers or Internet-service providers, by introducing a new format for encrypted data, which makes ciphertexts indistinguishable from random bit strings, and by developing an efficient encoding scheme for producing such ciphertexts in typical communication setups. In Chapter 3, we design a protocol for private information retrieval; it atomically ensures privacy and data integrity, even when a user interacts with a malicious data provider. Finally, in Chapter 4, to guard users against attacks via compromised software-vendors, we architect and build a system for secure and accountable distribution of software updates.

Chapter 2: Protecting Encryption Metadata

Most encrypted-data formats leak metadata, such as the format version, the encryption algorithms used, which application created it, the number of recipients who can decrypt the data, and even the recipients' identities, via their plaintext headers. This leakage can pose security and privacy risks to users, e.g., by revealing the full membership of a group of collaborators from a single encrypted e-mail or by enabling an eavesdropper to fingerprint the precise encryption-software version and configuration the sender used [72], [95], [123], [162].

We argue that this metadata exposure is unnecessary and propose that future encryption schemes and protocols improve security and privacy hygiene by producing *Padded Uniform Random Blobs* or PURBs: ciphertexts indistinguishable from random bit strings for anyone without a decryption key. A PURB's content does not leak any metadata, not even the application that created it, and is padded such that even its length reveals as little as possible.

Encoding and decoding ciphertexts with no cleartext markers present efficiency challenges. A PURB's recipient needs to know how to decrypt it, i.e., what cryptographic key, algorithm, and format version to use. Furthermore, if the PURB is addressed to multiple recipients, different parts of it could be decryptable with different keys. In order to address this challenge, we instantiate MsPURB, a cryptographically agile encoding scheme that enables legitimate recipients to efficiently decrypt a PURB, even when encrypted for any number of recipients' public keys and/or passwords, and when these public keys are of different cryptographic algorithms. At a high-level, a sender encrypts both content and metadata, and she organizes them in the ciphertext in such a way that a recipient can efficiently parse this ciphertext by using trial decryptions. MsPURB builds upon the ideas of key and data encapsulation [104, Chapter 11.3] and uses efficient data structures for ciphertext organization. Our experimental evaluation demonstrates that MsPURB gracefully performs with a high number of recipients and encryption algorithms used.

Chapter 3: Verifiable Single-Server Private Information Retrieval

Private information retrieval (PIR) is a powerful cryptographic primitive for retrieving an entry from a database, without disclosing which exact entry a user fetches. At a high level, in PIR, a user sends a blinded query to a service provider that applies this query to the whole database in order to compute the response. The user is then able to unblind the response to recover the data of interest. Examples of what a user could read from the database are messages [9], media files and cryptographic material [89], software updates [43], the status of another user's presence on a online service [37], [128], and a proof that the user's password is in a compromised database [6], [122].

There are two main types of PIR: single-server [115] and multi-server [49]. In single-server PIR, a client interacts with a single logical server, and the protocol security relies on the hardness of certain computational assumptions, e.g., the decisional composite residuosity

assumption [145] or discrete-logarithm problem [68], hence this PIR type is also referred to as computational. Multi-server PIR requires the service provider to consist of several (at minimum, two) independent servers and can provide information-theoretic security. The information-theoretic PIR is typically more efficient than its computational alternative but requires the trust assumption of at least one honest provider's server, i.e., if all the servers collude, they can break user privacy. In this dissertation, we are interested exclusively in single-server computational PIR.

A major barrier to naïvely using PIR in real systems is that, as a primitive, PIR does not provide any guarantees of data integrity. In all single-server PIR schemes that we are aware of [38], [41], [52], [115], [124], [150], a malicious server can choose the exact output that the client will receive by substituting all the database records with a chosen value, before processing the client's request. Consequently, in applications where data integrity matters standard single-server PIR schemes do not suffice.

Our first contribution in this chapter is to observe that naïvely adding standard integrity mechanisms, such as signing records in the database [136] or computing message authentication codes [58], introduces a side-channel that a malicious server can use to break user privacy. The malicious server can tamper with some entries of the database and observe whether the user accepts the response, i.e., whether the authentication check passes or fails. If the user accepts, the server learns that the user has *not* retrieved the tampered entries, or that the user *has* retrieved one of them, otherwise. We argue that the service provider can implicitly learn this client-acceptance outcome in many real-world scenarios. For example, in a social network with private presence-check [37] and hidden-recipient messaging, the acceptance bit comes from the mere fact that the client has sent a message to another unknown user, after checking the online-presence.

To address this side-channel problem, we introduce single-server *verifiable* private-information-retrieval that ensures that an honest client either obtains the "correct" data from the system's servers or detects server misbehavior and aborts. Our approach is to assume that the client can obtain—via some out-of-band means—a short digest of the database. Then, we specify that a PIR protocol satisfies integrity if the client accepts the protocol's output only if the output is consistent with the database that the digest represents. In some applications of PIR, e.g., to public-key directories, the client could obtain this database digest via a gossip mechanism, as in CONIKS [130]. In other applications of PIR (e.g., to video streaming [89]), there could be a database *owner*, distinct from the PIR servers, who can produce, sign, and distribute this digest.

We then construct a single-server verifiable PIR scheme, in which the client verifies the PIR server's answer against a constant-size database digest that the client obtains via some out-of-band means. Our verifiable-PIR protocol extends a textbook discrete-log-based PIR scheme with additional randomness that the client uses to authenticate the server's response. We prove the security of this construction in the generic-group model [170]. On database size n and

security parameter λ, our single-server verifiable-PIR scheme has communication cost $O(\sqrt{n})$, whereas non-verifiable PIR schemes have cost $O(\log n)$. (The big-O hides polynomials in the security parameter.) The heavy use of discrete-log-type cryptographic operations makes our scheme relatively costly in terms of computation. In future work, we hope to reduce both its computation and communication costs to match that of non-verifiable PIR and to generalize the approach to be compatible with a wider array of PIR schemes (e.g., lattice-based ones).

Chapter 4: Securing Retrieval of Software Updates

In the final section of this dissertation, we switch from designing cryptographic schemes and primitives to building a system that improves on the data-retrieval practice, in one concrete application. Specifically, we design a software-update framework that survives partial compromise, provides transparency and strong security guarantees to it users, and it can flexibly evolve.

As software-update systems are responsible for managing, distributing, and installing code that is eventually executed on end systems, they constitute valuable targets for attackers who try to subvert the update infrastructure in order to inject malware. A recent series of software supply-chain compromises [40], [50], [78], [100], [125], [195] illustrates that the current practices in software distribution do not meet modern security requirements.

To improve the status-quo, we first identify four primary flaws, which often make these systems vulnerable to compromise, in the currently typical design of software-update systems. The integrity and authenticity of releases in these systems traditionally depends on a single signing key that is prone to accidental exposure [100], [133] and theft [80], [157]. The dependency of a system's security on a single singing key is the first issue that we need to address. The lack of transparency mechanisms in the current infrastructure of software distribution leaves room for backdooring of updates by compromised [44], [119], coerced [27], [79], [171], and malicious [97] software vendors and distributors. A concrete example is a compromise of so-called build servers where the source code of a software release is compiled into a binary file for distribution to end users. Several recent large-scale security incidents, such as the attack on SolarWinds [78] and the NotPetya attack [125], were made possible precisely due to the compromise of a build server. Transparency and accountability at all the steps of the software-update process must be enforced for defending such stealthy attacks. The third issue is that the typical software-update system design leaves clients vulnerable to replay and freeze attacks [44], [177]. Hence, clients must be able to verify the timeliness of updates. Finally, revoking and renewing signing keys (e.g., in reaction to a compromise) and informing all their clients about these changes is usually cumbersome. A modern software-update system should provide efficient and secure means to evolve signing keys.

To address the above challenges, we propose CHAINIAC, a decentralized software-update framework that eliminates single points of failure, enforces transparency, and provides efficient verifiability of integrity and authenticity for software-release processes. First, CHAINIAC

introduces a two-step decentralized release-sign-off model where developers express their approval on a release by signing it with an individual key, and third-party servers, or *witnesses*, validate theses approvals and conformance to a release policy. The end result is a single collective signature [174] that is almost as compact and inexpensive to verify as a conventional digital signature. Although it improves security, this approach does not place a burden on clients who otherwise would have to verify multiple signatures per updated package.

Second, CHAINIAC introduces *collectively verified builds* to validate source-to-binary correspondence of software releases. CHAINIAC's verified builds are an improvement over reproducible builds [126], [153], in that they ensure that binaries are not only reproducible in principle, but have indeed been identically reproduced from the corresponding source release by multiple independent verifiers. Concretely, this task is handled by a subset of the witness servers, or *build verifiers*; they reproducibly build the source code of a release, compare the result with the binary provided by the developers, and attest this validation to clients.

Third, CHAINIAC increases transparency and ensures the accountability of the update process by implementing a public update-timeline that comprises a release log, freshness proofs, and key records. The timeline is maintained collectively by the witness servers such that each new entry can be added only—and clients will accept it only—if appropriate thresholds of the witnesses and build verifiers approve it. This mechanism protects clients from compile-time backdoors and malware by ensuring the source-to-binary binding, and it guarantees that all users have a consistent view of the update history, thus preventing adversaries from stealthily attacking targeted clients with compromised updates. Even if an attacker manages to slip a backdoor into the source code, the corresponding signed binary stays publicly available for scrutiny, thereby preventing secret deployment against targeted users.

Finally, to achieve tamper evidence, consistency, and search efficiency of the timeline, and to enable a secure rotation of signing keys, CHAINIAC employs skipchains [142] : authenticated data structures that enable clients to efficiently navigate arbitrarily long update timelines, both forward (e.g., to validate a new software release) and backward (e.g., to downgrade or verify the validity of older package-dependencies needed for compatibility). Due to skipchains, even resource-constrained clients (e.g., IoT devices) can obtain and efficiently validate binary updates, using a hard-coded initial software version as a trust anchor. Such clients do not need to continuously track a release chain, like a Bitcoin full-node does, but can privately exchange, gossip, and independently validate on-demand newer or older blocks due to the skipchain's forward and backward links being offline-verifiable.

Before diving into the detailed description of our results, we list the publications that constitute the basis for this dissertation and write down the notation that we later will use.

Bibliographic notes

This dissertation is based on parts of the following jointly authored publications:

Chapter 2: K. Nikitin, L. Barman, W. Lueks, M. Underwood, J.-P. Hubaux, and B. Ford, "Reducing metadata leakage from encrypted files and communication with PURBs", *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 4, Jul. 2019 [141],

Chapter 3: S. Colombo, K. Nikitin, B. Ford, and H. Corrigan-Gibbs, "Verifiable private information retrieval", *Under submission*, 2021 [51],

Chapter 4: K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds", in *USENIX Security Symposium*, 2017 [142].

Notation

We briefly survey the notation that we will use throughout this dissertation:

Sets and Operations. For a natural number n, $[n]$ denotes the finite set $\{1,\ldots,n\}$. $\mathbb{N}$ is the set of all natural numbers. When we write $x \in \{0,1\}^n$, we mean that x is a binary string of length n. We denote by $s \xleftarrow{\$} S$ the selection of the element s, uniformly at random from the finite set S. In general, we use \$ to indicate randomness. Given a set S, we denote with S^n (S^*, respectively) the set of all vectors of length n (arbitrary length, respectively) whose elements belong to S. For $a, b \in \{0,1\}^*$, we denote their concatenation as $a \parallel b$. The bit length of a is written as $|a|$.

Algorithms. We use $\leftarrow$ to indicate assignment and $=$ to indicate equality. We also use $\mathsf{negl}(\cdot)$ to denote a function whose inverse grows faster than any fixed polynomial. We write PPT as an abbreviation for probabilistic polynomial-time. The symbol $\perp$ is an output that indicates rejections.

Algebra. We use $\mathbb{G}$ to denote a cyclic finite group of prime order and write $1_{\mathbb{G}}$ to denote this group's identity element.

2 Protecting Encryption Metadata

Metadata can tell volumes. Knowing who called whom, at what time a message was sent, or the duration of an activity can sometimes suffice to uncover the callers' identities, the message contents, and the result of the activity. In encrypted communications, metadata come in multiple flavors: traffic patterns, packet lengths, cleartext headers with supplemental information and mere traffic volume; these flavors have all been used as information sources in various types of attacks, from deanonymizing users [148], [154] to identifying what these users do [47], [132], [196] and receive [48], [55], [167].

In this chapter, we consider one fundamental and arguably harmful type of metadata exposure that makes some of the above attacks possible: information leakage in the cleartext headers of ciphertexts. In the following sections, we first describe the problem of metadata exposure in cipheretxts. We then propose a new ciphertext format that does not have any unencrypted information in its headers. Finally, we design an encoding scheme that can efficiently produce or decode such ciphertexts, despite the absence of cleartext markers.

2.1 Metadata Exposure in Ciphertexts

Today, most applications use encryption schemes and protocols to provide data confidentiality to their users. Many of these schemes and protocols, however, leave some or all encryption metadata in cleartext. This metadata exposure is traditionally assumed to be important for the recipient's decryption efficiency and not security-sensitive. For example, OpenPGP [42] reveals in cleartext headers the public keys of the intended recipients, the algorithm used for encryption, the version of the packet format, and the actual length of the payload. Similarly, secure-communication protocols leak information during key and algorithm agreement. The TLS handshake [159], for example, exposes the protocol version, the chosen cipher suite, and the public keys of the communicating parties.

However, research has consistently shown that attackers can exploit metadata to infer sensitive information about communication parties and its content, and to decide on the attack vector.

For example, an attacker could be able to fingerprint users [72], [95], [148], [154], [162] by associating connection end points with cryptographic and compression algorithms that they support. Revealing the identities and/or the number of message recipients can enable an attacker to infer the group membership, e.g., of a minority or activists group, or even enable rubber-hose decryption attacks. The exposed encryption scheme or the application version could also suggest to the attacker to exploit implementation or cryptographic weaknesses, e.g., RC4-encrypted traffic in TLS [67] has been discovered vulnerable to the ciphertext-only attack [3]. Finally, revealing which application created the encrypted message can already be sensitive, because an attacker, e.g., an oppressive regime, could censor or collect the traffic of this application for incrimination.

The question that we pose is whether this metadata exposure is necessary. An application does not typically need to recognize a ciphertext because the attribution of ciphertexts to corresponding applications either occurs at a different level of the stack, e.g., by associating all packets coming from a given IP address to a given local port with application session, or the attribution is managed by the user, e.g., in encrypted e-mail. The only remaining purpose of cleartext metadata is to simplify the decryption by specifying the encryption algorithm, the version format, the payload length, and other encryption information. We argue that even this exposure is, in fact, unnecessary. In the following section, we propose a new ciphertext format without any cleartext metadata, and we demonstrate how to efficiently process such cipheretxts.

2.2 Padded Uniform Random Blobs

As a step towards countering the privacy threats of metadata exposure, we propose that encryption schemes and protocols should produce ciphertexts in the form of *Padded Uniform Random Blobs* (PURBs): an encrypted-data format that is designed to protect *all* encryption metadata. A PURB encrypts application content and metadata into a single blob that is indistinguishable from a random string, and is padded to minimize information leakage via its length. Unlike traditional formats, a PURB does not leak the encryption schemes used, who or how many recipients can decrypt it, or what application or software version created it.

Naturally, the surrounding context in which encrypted data appears, such as cleartext e-mail headers or the filename of an encrypted file, might still leak metadata that neither PURBs nor any format alone can hide. But that contextual leakage is typically much more obvious to users, and, hence, easier to reason about and to potentially mitigate than binary metadata embedded within ostensibly-encrypted data. A user who looks at the random-looking base64 blob in a PGP-encrypted e-mail might naturally assume that everything in that blob is encrypted. With PGP that assumption is unfortunately wrong, but with PURBs it would be correct.

While simple in concept, PURBs by definition contain *no* cleartext structure or markers, and, hence, encoding and decoding them presents practical challenges, i.e., of how to encrypt

them for multiple recipients and multiple cryptographic algorithms while retaining efficiency. Before diving in designing an efficient encoding scheme, let us describe the scenarios where PURBs can find application.

2.2.1 Potential Applications

E-mail protection. E-mail systems traditionally use PGP or S/MIME for encryption. Their packet formats [42], however, exposes format version, encryption methods, number and public-key identities of the recipients, and public-key algorithms used. In addition, the payload is padded only to the block size of a symmetric-key algorithm used. Using PURBs for encrypted e-mail could minimize this metadata leakage. Furthermore, as e-mail traffic is normally sparse, the moderate overhead that PURBs incur can easily be accommodated.

Initiation of cryptographic protocols. In many cryptographic protocols, initial cipher suite negotiation, handshaking, and key exchange are performed unencrypted. In TLS 1.3 [159], an eavesdropper who monitors a connection from the start can learn the cipher suites and public-key types that the user supports and the Server Name Indication (SNI). The unencrypted SNI enables an eavesdropper to determine which specific web site a client is connected to among the sites hosted by the same server. The eavesdropper can also fingerprint the client [72], [95], [123], [162] or distinguish censorship-circumvention tools that try to mimic TLS traffic [81], [93]. TLS 1.3 [159] takes a few protective measures: there is less unencrypted metadata during the handshake in comparison with TLS 1.2 [67], and an experimental extension for encrypted SNI [163]. These measures are only partial, however, and leave other metadata, such as the protocol version number, the cipher suites, and the application-layer protocol negotiation, still visible. PURBs could facilitate fully-encrypted handshaking from the start, provided a client already knows at least one public key and a cipher suite that the server supports. Clients might cache this information from prior connections, or obtain it out-of-band while finding the server, e.g. via DNS-based authentication [92].

Encrypted disk volumes. VeraCrypt [96] uses a block cipher to turn a disk partition into an encrypted volume where the partition's free space is filled with random bits. For plausible deniability and coercion protection, VeraCrypt supports so-called hidden volumes: an encrypted volume whose content and metadata is indistinguishable from the free space of a primary encrypted volume hosting the hidden volume. This protection is limited, however, because a primary volume can host only a single hidden volume. A potential coercer might therefore assume by default that the coercee has a hidden volume, and interpret a claim of non-possession of the decryption keys as a refusal to provide them. PURBs can enhance coercion protection by enabling an encrypted volume to contain any number of hidden volumes, facilitating a stronger "$N+1$" defense. Even if a coercee reveals up to N "decoy" volumes, the coercer cannot know whether there are any more.

2.3 Encoding scheme

This section addresses the challenges of encoding and decoding *Padded Uniform Random Blobs* in a flexible, efficient, and cryptographically agile way. We first cover notation, system and threat models, followed by a sequence of strawman approaches that address different challenges on the path towards the full MsPURB scheme. We start with a scheme where ciphertexts are encrypted with a shared secret and addressed to a single recipient. We then improve it to support public-key operations with a single cipher suite, and finally to multiple recipients and multiple cipher suites.

2.3.1 Preliminaries

Let $\Pi = (\mathscr{E}, \mathscr{D})$ be an IND$-CCA2-secure authenticated-encryption (AE) scheme [23] where $\mathscr{E}_K(m)$ and $\mathscr{D}_K(c)$ are encryption and decryption algorithms, respectively, given a message m, a ciphertext c, and a key K. Let $\mathsf{MAC} = (\mathscr{M}, \mathscr{V})$ be strongly unforgeable Message Authentication Code (MAC) generation and verification algorithms. An authentication tag generated by MAC must be indistinguishable from a random bit string.

Let $\mathbb{G}$ be a cyclic finite group of prime order p generated by the group element g where the gap-CDH problem is hard to solve (e.g. an elliptic curve or a multiplicative group of integers modulo a large prime). Let λ be a standard security parameter and $\texttt{Hide}: \mathbb{G}(1^\lambda) \to \{0,1\}^\lambda$ be a mapping that encodes a group element of $\mathbb{G}$ to a binary string that is indistinguishable from a uniform random bit string (e.g. Elligator [29], or Elligator Squared [10], [176]). Let $\texttt{Unhide}: \{0,1\}^\lambda \to \mathbb{G}(1^\lambda)$ be the counterpart to Hide that decodes a binary string into a group element of $\mathbb{G}$, such that $\texttt{Unhide}(\texttt{Hide}(X)) = X$ for all $X \in \mathbb{G}$.

Let $\mathrm{H}: \mathbb{G} \to \{0,1\}^{2\lambda}$ and $\hat{\mathrm{H}}: \{0,1\}^* \to \{0,1\}^{2\lambda}$ be two distinct cryptographic hash functions. Let $\texttt{PBKDF}: \{\text{salt}, \text{password}\} \to \{0,1\}^{2\lambda}$ be a secure password-based key-derivation function [31], [112], [152], a "slow" hash function that converts a salt and a password into a bit string that can be used as a key for symmetric encryption.

System Model

Let *data* be an application-level unit of data (e.g. a file or network message). A *sender* wants to send an encrypted version of data to one or more *recipients*. We consider two main approaches for secure data exchanges:

(1) Via *pre-shared secrets*, where the sender shares with the recipients long-term one-to-one passphrases $\hat{S}_1, ..., \hat{S}_r$ that the participants can use in a password-hashing scheme to derive ephemeral secrets $S_1, ..., S_r$.

(2) Via *public-key cryptography*, where sender and recipients derive ephemeral secrets $Z_i = \mathrm{H}(X^{y_i}) = \mathrm{H}(Y_i{}^x)$ using a hash function H. Here $(x, X = g^x)$ denotes the sender's one-time (private, public) key pair and $(y_i, Y_i = g^{y_i})$ is the key pair of recipient $i \in 1, ..., r$.

In both scenarios, the sender uses ephemeral secrets $S_1, ..., S_r$ or $Z_1, ..., Z_r$ to encrypt (parts of) the PURB header using an authenticated encryption (AE) scheme.

We refer to a tuple $S = \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$ used in the PURB generation as a *cipher suite*. This can be considered similar to the notion of a cipher suite in TLS [159]. Replacing any component of a suite (e.g., the group) results in a different cipher suite.

Threat Model and Security Goals

We will consider two different types of computationally bounded adversaries:

1. An *outsider* adversary who does not hold a private key or a password valid for decryption;
2. An *insider* adversary who is a "curious" and active legitimate recipient with a valid decryption key.

Both adversaries are adaptive.

Naturally, the insider adversary has more power, e.g. she can recover the plaintext payload. Hence, we consider different security goals given the adversary type:

1. We seek IND$-CCA2 security against the outsider adversary, i.e. the encoded content and *all metadata* must be indistinguishable from random bits under an adaptive chosen-ciphertext attack;
2. We seek recipient privacy [16] against the insider adversary under a chosen-plaintext attack, i.e. a recipient must not be able to determine the identities of the ciphertext's other recipients.

Recipient privacy is a generalization of the key indistinguishability notion [20] where an adversary is unable to determine whether a given public key has been used for a given encryption.

System Goals

We wish to achieve two system goals beyond security:

- The encoding scheme must provide cryptographic agility, i.e., it should accommodate both a single or multiple recipients, allow encryption for each recipient using a shared

password or a public key, and support different cipher suites. Adding support for new cipher suites must be seamless and must not affect or break backward compatibility with other cipher suites.

- PURBs' encoding and decoding must be "reasonably" efficient. In particular, the number of expensive public-key operations should be minimized.

2.3.2 Encryption to a Single Passphrase

We begin with a simple strawman PURB encoding format that enables a sender to encrypt data by using a single long-term passphrase $\hat{S}$ that has been shared with a single recipient (e.g., out-of-band via a secure channel). The sender and recipient use an agreed-upon cipher suite defining the scheme's components. As the first step, the sender generates a fresh symmetric session key K and computes the PURB payload as $\mathscr{E}_K(\text{data})$. The application data can be potentially large, hence we stipulate encrypting it with a symmetric-key algorithm for efficiency and to avoid ciphertext expansion that is typical for public-key algorithms.

The sender now needs to encapsulate the session key and related metadata such that only the recipient can recover them using the pre-shared passphrase. For that, the sender creates an *entry point* (EP) that contains the session key K, the payload's encryption metadata, and the position of the payload. An entry point is essentially an instantiation of the data encapsulation mechanism [104, Chapter 11.3]. The benefits of encoding a per-recipient entry point will become more evident when we move to the scenarios with multiple recipients and cipher suites. The sender then generates a random salt, uses it to derive the ephemeral secret $S = \texttt{PBKDF}(\text{salt}, \hat{S})$ from the passphrase and encrypts the entry point by using the authenticate-encryption scheme $\mathscr{E}$. Finally, the sender concatenates the random salt, the entry point and the payload to form the PURB as shown in Figure 2.1.

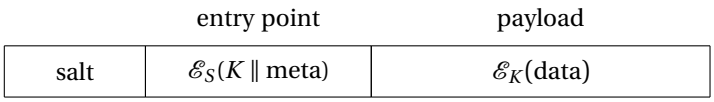

Figure 2.1: A PURB addressed to a single recipient and encrypted with a passphrase-derived ephemeral secret S.

2.3.3 Single Public Key, Single Suite

The challenge with the scheme described in the previous section is that sharing a passphrase in advance is cumbersome, and most of the interactions on the modern Internet use public-key cryptography, instead. Typically, the sender learns the public key of the other party via the Public-Key Infrastructure or the web of trust and encrypts data to that public key. When producing a ciphertext, the sender indicates in the file's cleartext metadata what public key this file is encrypted for. This approach, unfortunately, leaks the recipient's identity to an

eavesdropper. We address this use case with a second strawman PURB encoding format that enables decryption of an entry point EP using a private key.

To expand our scheme to the public-key scenario, we transform the data encapsulation instantiation to rely on the Diffie-Hellman computational problems [68] for derivation of the shared secret. The sender uses now a combination of public-key and symmetric-key cryptography to encapsulate the session key and metadata. Concretely, the new instantiation relies on the hardness of the gap-CDH assumption (see Appendix A.1.1 for details).

Let (y, Y) denote the recipient's key pair. The sender generates an ephemeral key pair (x, X), computes the ephemeral secret $Z = \mathrm{H}(Y^x)$, then proceeds as before, except she encrypts K and associated metadata with Z instead of S. The sender replaces the salt in the PURB with her encoded ephemeral public key $\texttt{Hide}(X)$, where $\texttt{Hide}(\cdot)$ maps a group element to a uniform random bit string. Whereas the salt can be random and uniform, an ephemeral public key X that the sender generates is typically not uniform. For example, elliptic-curve points, even in a compressed form, are distinguishable from random bits because they satisfy a curve equation, which can be validated by an eavesdropper. Hence, the hiding transformation is required for the PURB to remain indistinguishable from a random string. The resulting PURB format is shown in Figure 2.2.

encoded pk	entry point	payload
$\texttt{Hide}(X)$	$\mathcal{E}_Z(K \parallel \text{meta})$	$\mathcal{E}_K(\text{data})$

Figure 2.2: A PURB addressed to a single recipient that uses a public key Y, where X is the public key of the sender and $Z = \mathrm{H}(Y^x)$ is the ephemeral secret.

2.3.4 Multiple Public Keys, Single Suite

The ability to encrypt a message to several recipients is a requirement in multiple scenarios, e.g., in multicast communication or a mobile group chat. We hence add support for encrypting one message under multiple public keys that are of the same suite.

On the cryptographic side, we adopt the idea of multi-recipient public-key encryption [21], [114] where the sender generates a single key pair and uses it to derive an ephemeral secret with each of the intended recipients. (Combining the sender's private key with the public keys of different recipients yields different ephemeral secrets.) In our scheme, the sender creates now one entry point per recipient. These entry points contain the same session key and metadata—the encapsulated data are the same—but they are encrypted with different ephemeral secrets.

As a PURB's purpose is to prevent metadata leakage, including the number of the recipients, a PURB cannot reveal how many entry points exist in the header. Yet, a legitimate recipient needs to have a way to enumerate possible candidates for her entry point. Hence, the next

challenge is to find a space-efficient layout of entry points—with no cleartext markers—such that the recipients are able to efficiently find their segments.

Linear-table strawman. The most space-efficient approach is to sequentially place the entry points. In fact, OpenPGP suggests a similar approach for achieving better privacy [42, Section 5.1]. However, in this case, decryption is inefficient: the recipients have to sequentially attempt decryption of each potential entry point, before finding their own or reaching the end of the PURB.

Fixed-hash-table strawman. A more computationally-efficient approach is to use a hash table of a fixed size. The sender can create a hash table and place each encrypted entry point there, identifying the corresponding position by hashing an ephemeral secret. Once all the entry points are placed, the remaining slots are filled with random bits, hence a third-party is unable to deduce the number of recipients. The upper bound corresponding to the size of the hash table is public information. This approach, however, yields significant space overhead: in the common case of a single recipient, all the unpopulated slots are filled with random bits but still transmitted. Moreover, there is the downside of imposing an artificial limit on the number of recipients.

Our solution: Expanding hash tables. We therefore include not one but a sequence of hash tables whose sizes are consecutive powers of two. Immediately following the encoded public key, the sender encodes a hash table of length one, followed (if needed) by a hash table of length two, one of length four, etc., until all the entry points are placed. Unpopulated slots are filled with random bits. The sender can "lay out" the header and figure out how many hash tables are needed before she encrypts the entry points, thus the latter can still contain the payload's start position.

To decrypt a PURB, a recipient decodes the public key X, derives the ephemeral secret, computes the hash index in the first table (which is always zero), and tries to decrypt the corresponding entry point. On failure, the recipient moves to the second hash table, seeks the correct position and tries again, and so on, until either she finds a valid entry point or reaches the end of the PURB.

We now formalize this scheme. Let r be the number of recipients and $(y_1, Y_1), \ldots, (y_r, Y_r)$ be their corresponding key pairs. The sender generates a fresh key pair (x, X) and computes one ephemeral secret $k_i = \mathrm{H}({Y_i}^x)$ per recipient. The sender uses a second hash function $\hat{\mathrm{H}}$ to derive independent encryption keys as $Z_i = \hat{\mathrm{H}}(\text{"key"} \parallel k_i)$ and position keys as $P_i = \hat{\mathrm{H}}(\text{"pos"} \parallel k_i)$. Then, the sender encrypts the data and creates r entry points $\mathscr{E}_{Z_1}(K \parallel \text{meta})), \ldots, \mathscr{E}_{Z_r}(K \parallel \text{meta})$. The position of an entry in a hash table j is computed as $P_i \mod 2^j$. The sender iteratively attempts to place an entry point in HT0 (hash table 0), then in HT1, and so on, until placement succeeds (i.e., no collision occurs). If placement fails in the last existing hash table HTj, the sender appends another hash table HT$(j+1)$ of size 2^{j+1} and places the entry point there. An example of a PURB encrypted to five recipients is illustrated in Figure 2.3.

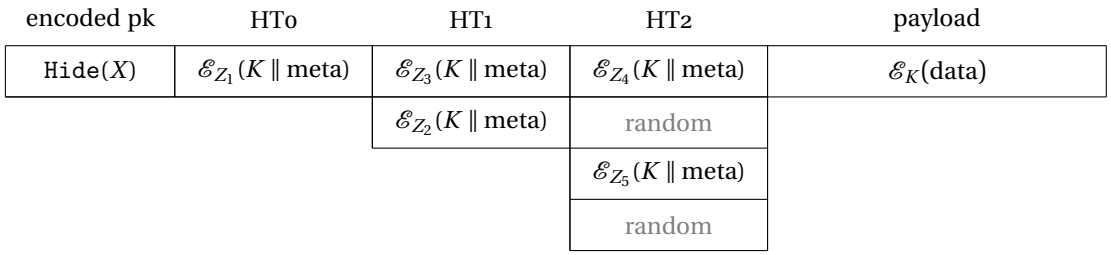

Figure 2.3: A PURB with hash tables of increasing sizes (HT0, HT1, HT2). The entry points occupy the total of five slots in all the hash tables, the rest is filled with random bits.

Although the logical format representation in Figure 2.3 makes it look like the hash tables stand out, in reality a PURB is a bit string where all the hash-table slots are placed sequentially one after another. Figure 2.4 shows how the logical presentation from Figure 2.3 flattens to a binary string.

Figure 2.4: An example of the flattened binary-string representation of a PURB with multiple entry-point hash tables.

Although the recipient does not initially know the number of hash tables in a PURB during decryption, she recipient needs to do only a single expensive public-key operation, and the rest are inexpensive symmetric-key decryption trials. In the worst case of a small message encrypted to many recipients, or a non-recipient searching for a nonexistent entry point, the total number of trial decryptions required is logarithmic in the PURB's size.

In the common case of a single recipient, only a single hash table of size 1 exists, and the header is compact. The scheme essentially falls back to the single-recipient encoding from Section 2.3.3. With r recipients, the worst-case compactness is having r hash tables (if each insertion leads to collisions with each previously inserted entry point), which happens with exponentially decreasing probability. The expected number of trial decryptions is $\log_2 r$.

2.3.5 Multiple Public Keys and Suites

In the real world, not all data recipients' keys might use the same cipher suite. For example, users might prefer different key lengths or might use public-key algorithms in different groups. Furthermore, we must be able to later add support for new cipher suites that might require larger and differently-structured keys and ciphertexts. These new additions must not violate interoperability and compatibility with old cipher suites. We therefore extend the previous strawman schemes to produce *Multi-Suite PURB* or MsPURB, which offers cryptographic agility by supporting the encryption of data for multiple different cipher suites.

When a PURB is multi-suite encrypted, the recipients need a way to learn whether a given suite has been used and where the encoded public key of this suite is located in the PURB. There are two obvious approaches to enabling recipients to locate encoded public keys for multiple cipher suites: to pack the public keys linearly at the beginning of a PURB, or to define a fixed byte position for each cipher suite. Both approaches incur undesirable overhead. In the former case, the recipients have to check all possible byte ranges, performing an expensive public-key operation for each. The latter approach results in significant space overhead and lack of agility, as unused fixed positions must be filled with random bits, and adding new cipher suites requires either assigning progressively larger fixed positions or compatibility-breaking position changes to existing suites.

Set of Standard Positions. To address this challenge, we introduce a *set* of standard byte positions per suite. These sets are public and standardized for all PURBs. The set refers to the positions where the suite's public key could be in the PURB. For instance, let us consider a toy suite `PURB_X25519_AES128GCM_SHA256`. We can define—arbitrarily for now—the set of positions as $\{0, 64, 128, 1024\}$. As the length of the encoded public key is fully defined by the suite (32 bytes here, as Curve25519 is used), the recipients will iteratively try to decode a public key at [0:32), then [64:96), etc.

If the sender wants to encode a PURB for two suites A and B, she needs to find one position in each set such that the public keys do not overlap. For instance, if $\text{set}_A = \{0, 128, 256\}$ and $\text{set}_B = \{0, 32, 64, 128\}$, and the public keys' lengths are 64 and 32, respectively, one possible choice would be to put the public key for suite A in [0:64), and the public key for suite B in [64:96). All suites typically have position 0 in their set, so that in the common case of a PURB encoded for only one suite, the encoded public key is at the beginning of the PURB for maximum space efficiency. Figure 2.5 illustrates an example encoding. With well-designed sets, in which each new cipher suite is assigned at least one position not overlapping with those assigned to prior suites, the sender can encode a PURB for any subset of the suites. We address efficiency hereunder, and provide a concrete example with real suites in Section 2.3.8.

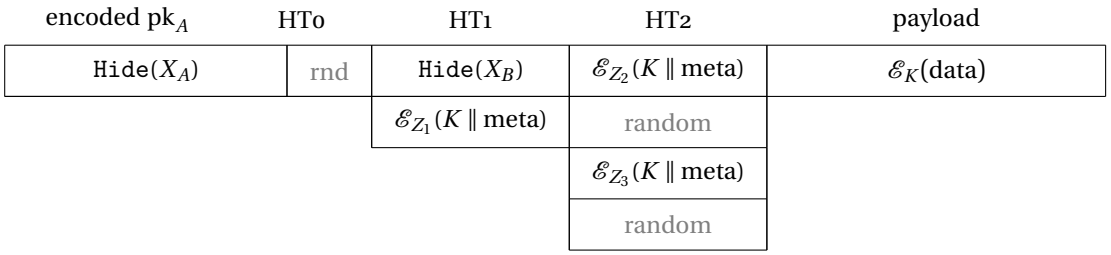

Figure 2.5: An example of a PURB encoded to three public keys in two suites (A and B). The sender generates one ephemeral key pair per suite (X_A and X_B). In this example, X_A is placed at the first allowed position, and X_B moves to the second allowed position (as the first position is taken by suite A). Those positions are public and fixed for each suite. HT0 cannot be used for storing an entry point, as X_A partially occupies it; HT0 is considered "full" and the entry point is placed in subsequent hash tables—here, HT1.

Overlapping Layers. One remaining challenge is that suites might indicate different lengths for both their public keys and entry points. A sender accommodates this requirement by processing each suite used in a PURB as an independent logical layer. Conceptually, each layer is composed of the public key and the entry-point hash tables for the recipients that use a given suite, and all suites' layers overlap. To place the layers, a sender first initializes a byte layout for the PURB. Then, she reserves in the byte layout the positions for the public keys of each suite used. Finally, she fills the hash tables of each suite with corresponding entry points. She identifies whether a given hash-table slot can be filled by checking the byte layout; the bytes might already be occupied by an entry point of the same or a different suite or one of the public keys. The hash tables for each suite start immediately after the suite public key's first possible position. Thus, upon reception of a PURB, a decoder knows exactly where to start decryption trials. The payload is placed right after the last encoded public key or hash table, and its start position is recorded in the meta in each entry point.

Decoding Efficiency. We have not yet achieved our decoding efficiency goal, however: the recipient must perform several expensive public-key operations for each cipher suite, one for each potential position until the correct position is found. We reduce this overhead to a single public-key operation per suite by removing the recipient's need to know in which of the suite positions the public key was actually placed. To accomplish this, a sender XORs bytes at all the suite positions and places the result into one of them. The sender first constructs the whole PURB as before, then she substitutes the bytes of the already-written encoded public key with the XOR of bytes at all the defined suite positions (if they do not exceed the PURB length), which could even correspond to encrypted payload. To decode a PURB, a recipient starts by reading and XORing the values at *all* the positions defined for a suite. This results in an encoded public key, if that suite was used in this PURB.

Encryption Flexibility. Although multiple cipher suites can be used in a PURB, so far these suites must agree on one payload encryption scheme, as a payload appears only once. To lift this constraint, we decouple encryption schemes for entry points and payloads. An entry-point encryption scheme is a part of a cipher suite, whereas a payload encryption scheme is indicated separately in the metadata "meta" in each entry point.

2.3.6 Non-malleability

Our encoding scheme MsPURB so far ensures integrity only of the payload and the entry point a decoder uses. If the entry points of other recipients or random-byte fillings are malformed, a decoder will not detect this. If an attacker obtains access to a decoding oracle, he can randomly flip bits in an intercepted PURB, query the oracle on decoding validity, and learn the structure of the PURB including the exact length of the payload. An example of exploiting malleability is the Efail attacks [155], which tamper with PGP- or S/MIME-encrypted e-mails to achieve exfiltration of the plaintext.

To protect PURBs from undetected modification, we add integrity protection to MsPURB using a MAC algorithm. A sender derives independent encryption $K_{enc} = \hat{\mathrm{H}}(\text{“enc”} \parallel K)$ and MAC $K_{mac} = \hat{\mathrm{H}}(\text{“mac”} \parallel K)$ keys from the encapsulated key K, and uses K_{mac} to compute an authentication tag over a full PURB as the final encoding step. The sender records the utilized MAC algorithm in the meta in the entry points, along with the payload encryption scheme that now does not need to be authenticated. The sender places the tag at the very end of the PURB, which covers the entire PURB including encoded public keys, entry point hash tables, payload ciphertext, and any padding required.

Because the final authentication tag covers the entire PURB, the sender must calculate it after all other PURB content is finalized, including the XOR-encoding of all the suites' public key positions. Filling in the tag would present a problem, however, if the tag's position happened to overlap with one of the public key positions of some cipher suite, because filling in the tag would corrupt the suite's XOR-encoded public key. If the authentication tag falls into any of the possible public key positions for the cipher suites in use, the sender simply pads the PURB until the overlap is resolved.

To encode a PURB, a sender prepares entry points, lays out the header, encrypts the payload, adds padding, and computes the PURB's total length. If any of the byte positions of the authentication tag to be appended overlap with public key positions, the sender increases the padding to next bracket, until the public-key positions and the tag are disjoint. The sender proceeds with XOR-encoding all suites' public keys, and with computing and appending the tag. Upon receipt of a PURB, a decoder computes the potential public keys, finds and decrypts her entry point, learns the decryption scheme and the MAC algorithm with the size of its tag. She then verifies the PURB's integrity and decrypts the payload.

2.3.7 Complete Construction

We summarize the encoding scheme in Construction 1, Construction 2, and Algorithm 1. We begin by defining the helper HdrPURB algorithms that encode and decode a PURB header's data for a single cipher suite. We then use these algorithms in defining the final MsPURB encoding scheme.

Recall the notion of a cipher suite $S = \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$, where $\mathbb{G}$ is a cyclic group of prime order p generated by g; `Hide` is a mapping: $\mathbb{G} \to \{0,1\}^{\lambda}$; $\Pi = (\mathscr{E}, \mathscr{D})$ is an authenticated-encryption scheme; and $\mathrm{H} : \mathbb{G} \to \{0,1\}^{2\lambda}$, $\hat{\mathrm{H}} : \{0,1\}^{*} \to \{0,1\}^{2\lambda}$ are two distinct cryptographic hash functions that we model as random oracles [24]. Let sk and pk be a private key and a public key, respectively, for $\langle \mathbb{G}, p, g \rangle$ defined in a cipher suite.

Construction 1 (HdrPURB)**.**

$\underline{\mathsf{HdrPURB.Encap}(R = \{pk_1 = Y_1, \ldots, pk_r = Y_r\}, S) \to (\tau, k_1, \ldots, k_r)}$

1. Sample a random value $x \xleftarrow{\$} \mathbb{Z}_p$ and compute the ephemeral public key $X \leftarrow g^x$.
2. Derive $k_1 \leftarrow \mathrm{H}(Y_1^x), \ldots, k_r \leftarrow \mathrm{H}(Y_r^x)$.
3. Encode X to a uniform string $\tau_X \leftarrow \texttt{Hide}(X)$.
4. Output $\tau \leftarrow \tau_X$ and $k_1, \ldots, k_r$.

$\underline{\mathsf{HdrPURB.Decap}(sk = y, \tau) \to k}$

1. Decode $X \leftarrow \texttt{Unhide}(\tau)$.
2. Output $k \leftarrow \mathrm{H}(X^y)$.

Theorem 1. *If for each cipher suite* $S = \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$ *used in a PURB we have that: the gap-CDH problem is hard relative to* $\mathbb{G}$*,* `Hide` *maps group elements in* $\mathbb{G}$ *to uniform random strings,* Π *is* IND$-CCA2*-secure, and* $\mathrm{H}, \hat{\mathrm{H}}$ *and* H' *are modeled as a random oracle; and moreover that* MAC *is strongly unforgeable with its MACs being indistinguishable from random, and the scheme for payload encryption* (Enc, Dec) *is* IND$-CPA*-secure, then MsPURB is* IND$-CCA2*-secure against an outsider adversary.*

Proof. See Appendix A.1.2. □

Theorem 1 also implies that an outsider adversary cannot break recipient privacy under an IND$-CCA2 attack, as long as the two possible sets of recipients N_0, N_1 induce the same distribution on the length of a PURB.

Theorem 2. *If for each cipher suite* $S = \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$*, used in a PURB we have that: the gap-CDH problem is hard relative to* $\mathbb{G}$*,* `Hide` *maps group elements in* $\mathbb{G}$ *to uniform random strings,* Π *is* IND$-CCA2*-secure,* H *and* $\hat{\mathrm{H}}$ *are modeled as a random oracle, and the order in which cipher suites are used for encoding is fixed; then MsPURB is recipient-private against an* IND$-CPA *insider adversary.*

Proof. See Appendix A.1.3. □

Construction 2 (MsPURB)**.**

MsPURB.Setup$(1^\lambda) \to S$

1. Initialize a cipher suite $S \leftarrow \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$.

MsPURB.KeyGen$(S) \to (sk, pk)$

1. Sample a random value $x \xleftarrow{\$} \mathbb{Z}_p$ and compute $X \leftarrow g^x$.
2. Output $(sk = x, pk = X)$.

MsPURB.Enc$(R = \{pk_1(S_1), \ldots, pk_r(S_r)\}, m) \to c$

1. Pick an appropriate symmetric-key encryption scheme (Enc, Dec) with key length λ_K, a MAC algorithm MAC $= (\mathcal{M}, \mathcal{V})$, and a hash function $\mathrm{H}' : \{0,1\}^* \to \{0,1\}^{\lambda_K}$ s.t. the key length λ_K matches the security level of the most conservative suite.
2. Group R into $R_1, \ldots, R_n$, s.t. all public keys in R_i share the same suite S_i. Let $r_i \leftarrow |R_i|$.
3. For each R_i:
   (a) Run $(\tau_i, k_1, \ldots, k_{r_i}) \leftarrow$ HdrPURB.Encap(R_i, S_i);
   (b) Compute entry-point keys $\mathsf{keys}_i \leftarrow (Z_1 = \hat{\mathrm{H}}(\text{“key”} \parallel k_1), \ldots, Z_{r_i} \leftarrow \hat{\mathrm{H}}(\text{“key”} \parallel k_{r_i}))$ and positions $\mathsf{aux}_i \leftarrow (P_1 = \hat{\mathrm{H}}(\text{“pos”} \parallel k_1), \ldots, P_{r_i} \leftarrow \hat{\mathrm{H}}(\text{“pos”} \parallel k_{r_i}))$.
4. Sample $K \xleftarrow{\$} \{0,1\}^{\lambda_K}$ and record (Enc, Dec), MAC and H' in meta.
5. Compute a payload key $K_{enc} \leftarrow \mathrm{H}'(\text{“enc”} \parallel K)$ and a MAC key $K_{mac} \leftarrow \mathrm{H}'(\text{“mac”} \parallel K)$.
6. Obtain $c_{\mathsf{payload}} \leftarrow \mathsf{Enc}_{K_{enc}}(m)$.
7. Run $c' \leftarrow$ Layout$(\tau_1, \ldots, \tau_n, \mathsf{keys}_1, \ldots, \mathsf{keys}_n, \mathsf{aux}_1, \ldots, \mathsf{aux}_n, S_1, \ldots, S_n, K, \mathsf{meta}, c_{\mathsf{payload}})$ (see Algorithm 1 on page 24).
8. Derive an authentication tag $\sigma \leftarrow \mathcal{M}_{K_{mac}}(c')$ and output $c = c' \parallel \sigma$.

MsPURB.Dec$(sk(S), c) \to m/\perp$

1. Look up the possible positions of a public key defined by S and XOR the bytes at all the positions to obtain the encoded public key τ.
2. Run $k \leftarrow$ HdrPURB.Decap(sk, τ).
3. Derive $Z \leftarrow \hat{\mathrm{H}}(\text{“key”} \parallel k)$ and $P \leftarrow \hat{\mathrm{H}}(\text{“pos”} \parallel k)$.
4. Parse c as expanding hash tables and, using the secret Z as the key, trial-decrypt the entries defined by P to obtain $K \parallel \mathsf{meta}$. If no decryption is successful, return $\perp$.
5. Look up the hash function H', a MAC $= (\mathcal{M}, \mathcal{V})$ algorithm and the length of MACoutput tag σ from meta. Parse c as $\langle c' \parallel \sigma \rangle$. Derive $K_{mac} = \mathrm{H}'(\text{“mac”} \parallel K)$ and run $\mathcal{V}_{K_{mac}}(c', \sigma)$. On failure, return $\perp$.
6. Derive $K_{enc} \leftarrow \mathrm{H}'(\text{“enc”} \parallel K)$, read the start and the end of the payload from meta (it is written by Layout) to parse c' as $\langle hdr \parallel c_{\mathsf{payload}} \parallel \text{padding} \rangle$.
7. Output $m/\perp \leftarrow \mathsf{Dec}_{K_{enc}}(c_{\mathsf{payload}})$ where Dec is the payload decryption algorithm specified in meta.

Algorithm 1 presents the LAYOUT algorithm a sender uses of MsPURB.Enc. LAYOUT arranges a PURB's components in a continuous byte array.

Additional notation. We denote by $a[i:j] \leftarrow b$, the operation of copying the bits of b at the positions $a[i], a[i+1], \cdots a[j-1]$. If, before an operation $a[i:j] \leftarrow b$, $|a| < j$, we first grow a to length j. We sometimes write $a[i:] \leftarrow b$ instead of $a[i:|b|] \leftarrow b$. We use a "reservation array", which is an array with a method array.isFree(start,end) that returns True if and only if none of the bits array$[i]$,array$[i+1], \cdots$ array$[j-1]$ were previously assigned a value, and False otherwise.

2.3.8 Positions for Public Keys

This section provides an example of possible sets of allowed public-key positions for cipher suites in the PURB encoding. We intentionally make the example cipher suites similar to the required and recommended cipher suites in TLS 1.3 [159].

The PURB concept of "cipher suite" combines "suite" and "group" from TLS. We assume for simplicity that the group also defines the `Hide` algorithm. The examples of possible PURB cipher suites with an assignment of the public-key positions are listed in Table 2.1. For instance, if only suites 1 and 3 are used, the sender will place a public key for suite 1 in $[0,64]$ and set the bytes $[96,160]$ so that the XOR of $[0,64]$ and $[96,160]$ equals the encoded public key for suite 3. Note that a sender must respect the suite order during encoding. We provide a simple python script to design such sets in the code repository.

2.3.9 Practical Considerations

Cryptographic agility (i.e., changing the encryption scheme) for the payload is provided by the metadata embedded in the entry points. For entry points themselves, we recall that the recipient uses trial-decryption and iteratively tests cipher suites from a known, public, ordered list. To introduce support for a new cipher suite, it suffices to add it to this list. With this technique, a PURB does not need version numbers. There is, however, a trade-off between the number of supported cipher suites and the maximum decryption time. It is important that a sender follows the fixed order of the cipher suites during encoding because a varying order might result in a different header length, given the same set of recipients and sender's ephemeral keys, which could be used by an insider adversary.

If a nonce-based authenticated-encryption scheme is used for entry points, a sender needs to include a distinct *random* nonce as a part of entry-point ciphertext (the nonce of each entry point must be unique per PURB). Some schemes, e.g., AES-GCM [25], have been shown to retain their security when the same nonce is reused with different keys. When such a scheme is used, there can be a single *global* nonce to reuse by each entry point. Generalizing this global-nonce approach to any scheme requires, however, further analysis.

Algorithm 1: LAYOUT

```
// τ_i is an encoded public key of a suite S_i
// keys_i = ⟨Z_1,...,Z_r⟩ are entry-point keys
// aux_i = ⟨P_1,...,P_r⟩ are entry-point positions
// SuiteAllowedPositions are public values
Input  : ⟨τ_1,...,τ_n⟩, ⟨keys_1,...,keys_n⟩, ⟨aux_1,...,aux_n⟩,
         ⟨S_1,...,S_n⟩, K, meta, c_payload,
         SuiteAllowedPositions
Output : byte[]
// determine public-key positions for each suite
layout = []; // public-key and entry-point assignments
pubkey_pos = []; // chosen primary position per suite
pubkey_fixed = []; // all positions fixed so far
foreach τ_i in ⟨τ_1,...,τ_n⟩ do
    // decide suite's primary public key position
    for pos ∈ SuiteAllowedPositions(S_i) do
        if pubkey_fixed.isFree(pos.start, pos.end) then
            pubkey_pos.append(⟨τ_i, pos⟩);
            layout[pos.start:pos.end] ← τ_i;
            break;
        end
    end
    // later suites cannot modify these positions
    // without disrupting this suite's XOR
    for pos ∈ SuiteAllowedPositions(S_i) do
        pubkey_fixed[pos.start:pos.end] ← 'F';
    end
end
// reserve entry-point positions in hash tables
entrypoints = [];
foreach aux_i in ⟨aux_1,...,aux_n⟩ do
    while aux_i not empty do
        P ← aux_i.pop();
        ht_len = 1; // length of current hash table
        ht_pos = 0; // position of this hash table
        while True do
            index = P  mod ht_len; // selected entry
            start = ht_pos + index * entrypoint_len;
            end = start + entrypoint_len;
            if layout.isFree(start, end) then
                layout[start:end] ←$ {0,1}^end-start;
                entrypoints.append(⟨start, end, S_i⟩);
                break;
            end
            // if not free, double table size
            ht_pos += ht_len * entrypoint_len;
            ht_len *= 2;
        end
    end
end
// fill empty space in the layout
// with random bits
foreach start, end < layout.end do
    if layout.isFree(start, end) then
        layout[start:end] ←$ {0,1}^end-start
    end
end
// place the payload just past
// the header layout
meta.payload_start = |layout|;
meta.payload_end = |layout| + |c_payload|;
// fill entry-point reservations with ciphertexts
foreach keys_i in ⟨keys_1,...,keys_n⟩ do
    while keys_i not empty do
        Z = keys_i.pop();
        ⟨start, end, S⟩ ← entrypoints.pop();
        // Encrypt an entry point
        e ← E_Z(K ‖ meta);
        layout[start:end] ← e;
    end
end
// compute the padding and
// append it to layout
purb_len ← padding(|layout| + |c_payload| + mac_len);
mac_pos ← purb_len - mac_len;
while not pubkey_fixed.isFree(mac_pos, purb_len) do
    // MAC must not overlap with public-key positions:  if so, we pad to the next permitted size
    purb_len ← padding_function(purb_len + 1);
    mac_pos ← purb_len - mac_len;
end
padding_len ← mac_pos - meta.payload_end;
padding ←$ {0,1}^padding_len; // random padding
layout.append(c_payload ‖ padding);
// XOR suites' public key positions
// into primary
for (τ_i, pos) ∈ pubkey_pos do
    buffer = τ_i;
    for altpos ∈ SuiteAllowedPositions(S_i) do
        buffer = buffer ⊕ layout[altpos.start : altpos.end];
    end
    layout[pos.start:pos.end] ← buffer;
    // now ⊕ SuiteAllowedPositions(S_i) = τ_i
end
return layout
```

Hardening Recipient Privacy. The given instantiation of MsPURB provides recipient privacy only under a *chosen-plaintext* attack. If information about decryption success is leaked, an

#	Cipher suite	pk_len	EP_len	Allowed Positions
1	AES_128_GCM_SHA_256_SECP256R1	64	48	{0}
2	AES_128_GCM_SHA_256_X25519	32	48	{0, 64}
3	AES_256_GCM_SHA_384_SECP256R1	64	80	{0, 96}
4	AES_256_GCM_SHA_384_X25519	32	80	{0, 32, 64, 160}
5	CHACHA20_POLY1305_SHA_256_SECP256R1	64	64	{0, 64, 128, 192}
6	CHACHA20_POLY1305_SHA_256_X25519	32	64	{0, 32, 64, 96, 128, 256}

Table 2.1: An example of possible PURB cipher suites and Allowed Positions assigned to them. pk_len is the byte length of a suite's public key, and EP_len is the byte length of a suite's entry point. The encoding scheme finds any mapping so that each cipher suite can coexist in a PURB. The recipient must XOR the values at all the possible positions of a suite to obtain an encoded public key.

insider adversary could learn identities of other recipients of a PURB by altering the header, recomputing the MAC, and querying candidates. A possible approach to achieving IND$-CCA2 recipient privacy is to sign a complete PURB using a strongly existentially unforgeable signature scheme and to store the verification key in each entry point, as similarly done in the broadcast-encryption scheme by Barth et al. [16]. This approach requires, however, adaptation to the multi-suite settings, and it will result in a significant increase of the header size and decrease in efficiency. We leave this question for future work.

Default Scheme for Payload In addition to PURB suites, a list of suitable candidates for a payload encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$, a MAC algorithm MAC, and a hash function H' must be determined and standardized. This list can be seamlessly updated with time, as an encoder makes the choice and records it in meta on per-PURB basis. The chosen schemes are shared by all the cipher suites included in the PURB, hence these schemes must match the security level of the cipher suite with the highest bit-wise security. An example of suitable candidates, given the suites from Table 2.1, is $(\mathsf{Enc}, \mathsf{Dec})$ = AES256-CBC, MAC = HMAC-SHA384, and H' = SHA3-384.

Limitations. The MsPURB scheme is not secure against quantum computers, as it relies on discrete logarithm hardness. It is theoretically possible to substitute DH-based key encapsulation with a quantum-resistant variant to achieve quantum IND$-CCA2 security. The requirements for substitution are the IND$-CCA2 security and compactness (it must be possible to securely reuse sender's public key to derive shared secrets with multiple recipients). Furthermore, as MsPURB is non-interactive, it does not offer forward secrecy.

Simply by looking at the sizes (of the header for a malicious insider, or the total size for a malicious outsider), an adversary can infer a bound on the total number of recipients. We can partially address this with padding. No reasonable padding scheme can, however, perfectly hide this information. If this is a problem in practice, we suggest adding dummy recipients.

Protecting concrete implementations against timing attacks is a highly challenging task. The two following properties are required for basic hardening. First, the implementations of PURBs should always attempt to decrypt all potential entry points using all the recipient's suites. Second, decryption errors of any source as well as inability to recover the payload should be processed in constant time and always return $\perp$.

2.4 Experimental Evaluation

The main question we answer in the evaluation of the encoding scheme is whether it has a reasonable cost, in terms of both time and space overhead, and whether it scales gracefully with an increasing number of recipients and/or cipher suites. First, we measure the average CPU time required to encode and decode a PURB. Then, we compare the decoding performance with the performance of plain and anonymized OpenPGP schemes described below. Finally, we show how the compactness of the header changes with multiple recipients and suites, as a percentage of useful bits in the header.

Implementation. We implemented a prototype of MsPURB in Go. The implementation follows Construction 2, and it consists of 2 kLOC. Our implementation relies on the open-source Kyber library [116] for cryptographic operations. The code is designed to be easy to integrate with existing applications. The code is still proof-of-concept, however, and has not yet gone through rigorous analysis and hardening, in particular against timing attacks.

Reproducibility. All the datasets, the source code for PURBs and as well as scripts for reproducing all the experiments, are available in the main repository

https://github.com/dedis/purb

2.4.1 Experiments

Methodology. We ran the encoding experiments on a consumer-grade laptop, with a quad-core 2.2 GHz Intel Core i7 processor and 16 GB of RAM, using Go 1.12.5. To compare with an OpenPGP implementation, we use and modify Keybase's fork [106] of the default Golang crypto library [85], as the fork adds support for the ECDH scheme on Curve25519.

We further modify Keybase's implementation to add support for the anonymized OpenPGP scheme. All the encoding experiments use a PURB suite based on the Curve25519 elliptic-curve group, AES128-GCM for entry point encryption and SHA256 for hashing. We also apply the global nonce optimization, as discussed in §2.3.9. For experiments needing more than one suite, we use copies the above suite to ensure homogeneity across timing experiments. The payload size in each experiment is 1 KB. For each data point, we generate a new set of keys, one per recipient. We measure each data point 20 times, using fresh randomness each time, and depict the median value and the standard deviation.

Anonymized PGP. In standard PGP, the identity—more precisely, the public key ID—of the recipient is embedded in the header of the encrypted blob. This plaintext marker speeds up decryption, but enables a third party to enumerate all data recipients. In the so-called anonymized or "hidden" version of PGP [42, Section 5.1], this key ID is substituted with zeros. In this case, the recipient sequentially tries the encrypted entries of the header with her keys. We use the hidden PGP variant as a comparison for PURBs, which also does not indicate key IDs in the header but uses a more efficient structure. The hidden PGP variant still leaks the cipher suites used, the total length, and other plaintext markers (version number, etc.).

Encoding Performance

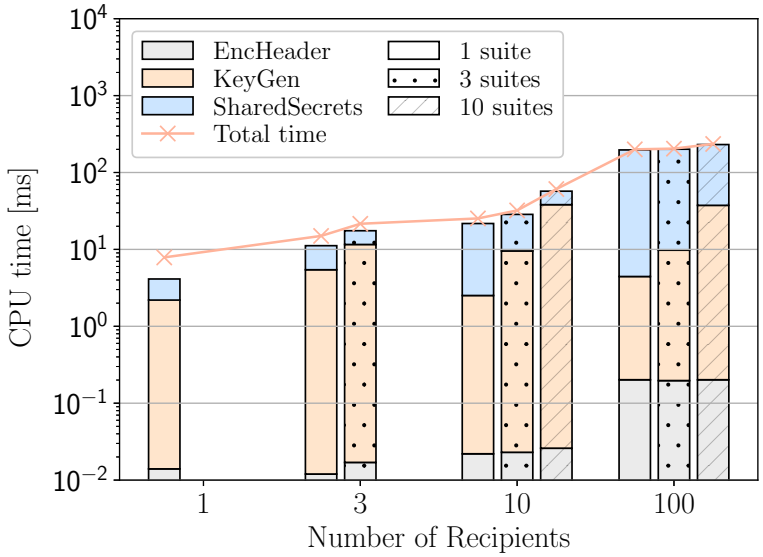

Figure 2.6: The CPU cost of encoding a PURB given the number of recipients and of cipher suites. EncHeader: encryption of entry points; KeyGen: generation and hiding of public keys; SharedSecrets: computation of shared secrets.

In this experiment, we measure how the time required to encode a PURB changes with a growing number of recipients and cipher suites, and analyze how the main computational components contribute to this duration. We divide the total encoding time into three logical components. The first is authenticated encryption of entry points. The second is the generation of sender's public keys (one per suite) and their Elligator encoding. A public key is derived by multiplying a base point with a freshly generated private key (scalar). If the resultant public key is not encodable, which happens in half of the cases, a new key is generated. Point multiplication dominates this component, constituting ≈ 90% of the total time. The third is the derivation of a shared secret with each recipient, essentially a single point-multiplication per recipient. Other significant components of the total encoding duration are payload encryption, MAC computation and layout composition. We consider the scenarios with one, three or ten

cipher suites. When more than one cipher suite is used, the recipients are equally divided among them.

Figure 2.6 shows that in the case of a single recipient, the generation of a public key and the computation of a shared secret dominate the total time and both take ≈ 2 ms. As expected, computing shared secrets starts dominating the total time when the number of recipients grows, whereas the duration of the public-key generation only depends on a number of cipher suites used. The encoding is arguably efficient for most cases of communication, as even with hundred recipients and ten suites, the time for creating a PURB is 235 ms.

Decoding Performance

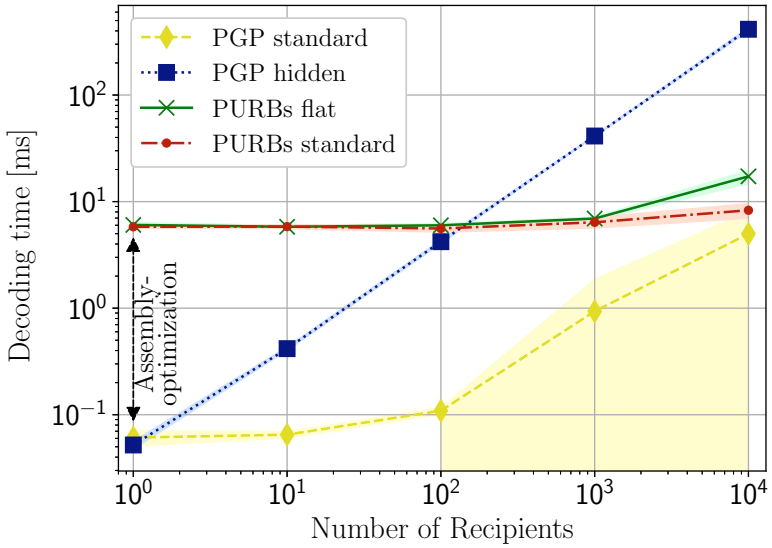

Figure 2.7: The worst-case CPU cost of decoding for PGP, PGP with hidden recipients, PURBs without hash tables (flat), and standard PURBs.

We measure the worst-case CPU time required to decipher a standard PGP message, a PGP message with hidden recipients, a *flat* PURB that has a flat layout of entry points without hash tables, and a standard PURB. We use the Curve25519 suite in all the PGP and PURB schemes.

Figure 2.7 shows the results. The OpenPGP library uses the assembly-optimized Go elliptic library for point multiplication, hence the multiplication takes ≈ 0.05–0.1 ms there, whereas it takes ≈ 2–3 ms in Kyber. This results in a significant difference in absolute values for small numbers of recipients. But our primary interest is the dynamics of total duration. The time increase for anonymous PGP is linear because, in the worst case, a decoder has to derive as many shared secrets as there are recipients. PURBs, in contrast, exhibit almost constant time, requiring only a single point multiplication regardless of the total number of recipients. A decoder still has to perform multiple entry-point trial decryptions, but one such operation

would account for only ≈ 0.3% of the total time in the single-recipient, single-suite scenario. The advantage of using hash tables, and hence logarithmically less symmetric-key operations, is illustrated by the difference between PURBs standard and PURBs *flat*, which is noticeable after 100 recipients and will become more pronounced if point multiplication is optimized.

Header Compactness

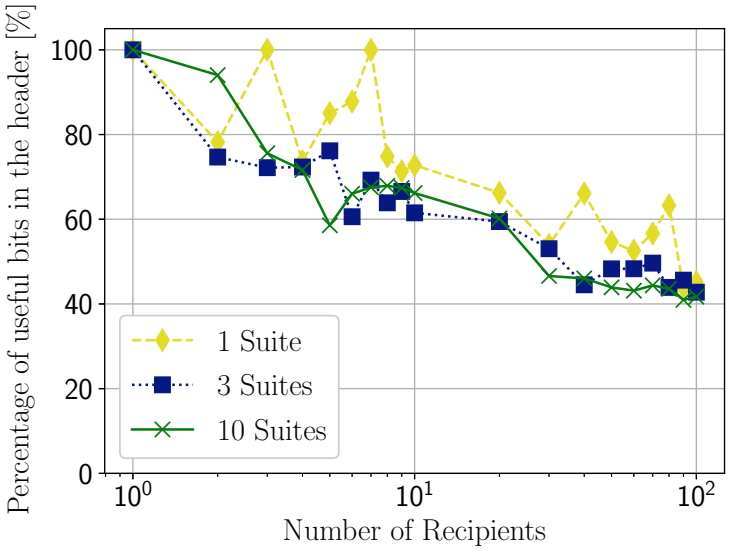

Figure 2.8: Compactness of the PURB header (% of non-random bits).

Our expanding hash table design enables more efficient decoding than the strawman design of linearly placing elements in the header. Our design is also less compact, however, which yields the communication-computation trade-off between the two approaches.

In Figure 2.8, we show the compactness, or the percentage of the PURB header that is filled with actual data, with respect to the number of recipients and cipher suites. Not surprisingly, an increasing number of recipients and/or suites increases the collisions and reduces compactness, although the compactness significantly fluctuates, especially in the single-suite case due to the stronger effect of adding new hash tables. In the most common case, however, of having one recipient and one cipher suite, the header is perfectly compact.

2.5 Related Work

The closest related work to PURBs is on broadcast encryption [16], [34], [65], [77], [83]. This prior work formalizes the security notion behind a ciphertext for multiple recipients and proposes concrete constructions. In particular, Barth et al. [16] were the first to propose a scheme for *private* broadcast encryption and introduced the notion of *recipient privacy*, in

which an adversary cannot tell whether a given public key is valid for decryption of a given ciphertext. In their proposed scheme, a message is encrypted with a random symmetric key, as in MsPURB, but then this key is encapsulated with public-key encryption directly (which leads to ciphertext expansion), and the whole ciphertext is signed using a strongly unforgeable signature scheme. They also propose using *hints* for efficient finding of encapsulated keys. The hints are the hashes of ephemeral Diffie-Hellman secrets which are placed right before the corresponding key encapsulations. Essentially, MsPURB derives the ephemeral secrets in the same way but uses them to encrypt the entry points and to determine entry-point location in the hash tables, whereas Barth et al. explicitly add them to the ciphertext as flags. The most significant improvements over prior work is that PURBs provide cryptographic agility, e.g., support multiple cipher suites in the same ciphertext, and achieve indistinguishability from random bits in the IND$-CCA2 model.

Traffic morphing [193] is a method for hiding the traffic of a specific application by masking it as traffic of another application and imitating the corresponding packet distribution. The tools built upon this method can be standalone [182] or use the concept of Tor pluggable transport [137], [187], [189] that is applied to prevent Tor traffic from being identified and censored [173]. There are two fundamental differences with PURBs. First, PURBs focus on a single unit of data; we do not yet explore the question of the time distribution of multiple PURBs. Second, traffic-morphing systems, in most cases, try to mimic a specific transport and sometimes are designed to hide the traffic of only one given tool, whereas PURBs are universal, arguably adaptable to any underlying application and aim at making encrypted traffic of all applications indistinguishable from one another. Moreover, it has been argued [81], [93], [181] that most traffic-morphing tools do not achieve unobservability in real-world settings due to discrepancies between their implementations and the systems that they try to imitate, because of the uncovered behavior of side protocols, error handling, responses to probing, etc. We believe that for a wide class of applications, using pseudo-random uniform blobs, either alone or in combination with other lower-level tools, is a promising solution in a different direction.

Finally, Sphinx [57] is an encrypted packet format for mix networks with the goal of minimizing the information revealed to the adversary. Sphinx shares similarities with PURBs in its binary format (e.g., the presence of a group element followed by a ciphertext). Unlike PURBs, however, it supports only one cipher suite, and one direct recipient (but several nested ones, due to the nature of mix networks). To the best of our knowledge, PURBs is the first solution that hides all encryption metadata while providing cryptographic agility.

2.6 Conclusion

Conventional encrypted data formats leak information via metadata in their cleartext headers, which might be used by attackers to infer sensitive information about the content or communication parties. We have argued that this metadata exposure is not necessary, and, as

evidence, have presented PURBs, a generic approach for designing encrypted data formats that do not leak anything at all, except for the padded length of the ciphertexts, to anyone without the decryption keys. We have shown that despite having no cleartext header, PURBs can be efficiently encoded and decoded, and can simultaneously support multiple public keys and cryptographic algorithms.

3 Verifiable Single-Server Private Information Retrieval

Data integrity is the property of preserving data in the form that it was originally recorded. This property is crucial for providing security to users who retrieve data: Potentially corrupted data might be not only useless but could even be harmful to a user—if the data have been maliciously tampered with, for example, in order to gain access to the user's machine.

In this chapter, we study data integrity in the context of private information retrieval (PIR) [49], [115]. First, we review standard PIR and demonstrate that naïve application of standard integrity-protection mechanisms to PIR can be unsafe. Then, we define integrity for PIR protocols and design a scheme that satisfies our requirements.

3.1 Private Information Retrieval (PIR)

A PIR protocol [49], [115] takes place between a client and one or more servers. Each server holds a copy of a database that consists of a set of equal-length records. The client wants to fetch a single record from the database, without revealing to the servers *which* record they retrieve.

There are two main types of PIR protocol: multi-server and single-server. In single-server PIR schemes [115], correctness holds if the single server is honest, whereas the traditional notion of PIR privacy holds even when the server is malicious. (We consider a stronger privacy notion in Section 3.3.1.) Single-server PIR schemes require a computationally bounded server and public-key cryptography operations [53], hence this type of PIR schemes is also referred to as *computational*. In multi-server PIR [49], the client communicates with $k > 1$ database replicas; correctness holds if all k servers are honest, and privacy holds if at least one server is honest. Multi-server PIR schemes traditionally offer information-theoretic privacy.

In this dissertation, we focus on single-server PIR. Hereafter, we will use the notions of PIR and single-server PIR interchangeably. Informally, a single-server PIR scheme must satisfy the following properties:

Correctness. A PIR scheme is *correct* if, when an honest client interacts with an honest server, the client always recovers the intended database record.

Privacy. A PIR scheme offers *privacy* if the database server "learns nothing" about which database record the client fetches.

A trivial PIR protocol is simply to have the server send the entire database to the client. Non-trivial PIR schemes, therefore, must also satisfy communication efficiency: the total number of bits that the client and server exchange must be sublinear in the database size.

The computational PIR schemes began with a seminal paper by Kushilevitz and Ostrovsky [115] where they proposed a single-server PIR scheme based on the hardness of the quadratic residuosity assumption [87]. Being the first to do this, their scheme had communication complexity $O(n^{\epsilon})$, for any $\epsilon > 0$, where n represents the database size. Cachin et al. improved the previous result with a scheme based on the Φ-Hiding assumption [41] and achieved a polylogarithmic communication cost. Later efforts [70], [82], [98], [124] retained the polylogarithmic complexity but improved the concrete costs by proposing the schemes that handle large database records (not just bits, as in the initial approaches). In practice, lattice-based PIR schemes are currently the most commonly used in the single-server setting [2], [8], [89].

3.2 PIR and Integrity

The standard definition of PIR gives the client no integrity guarantees. That is to say, if a PIR server deviates from the protocol, the malicious server can—in many PIR protocols, at least—control the output that the client receives. In other words, classic single-server PIR protocols do not give any correctness guarantees if the PIR server is malicious.

This lack of integrity protection is problematic in many applications of PIR:

- *Domain name system*: If a client uses PIR to query a DNS resolver, a malicious PIR server could cause the client to recover the wrong IP address for a hostname thus poison the client's DNS cache.
- *Content library*: If a client uses PIR to fetch a movie [89] or a software update [43], a malicious PIR server could cause the client to recover a malware-infected file instead.
- *Online certificate status protocol (OCSP)*: If a client uses PIR to query a CA's OCSP server to check whether a public-key certificate has been revoked, a malicious PIR server could cause the client to incorrectly accept a revoked certificate.

The *non-private* variants of these applications provide integrity. For example, CONIKS [130] provides integrity of key bindings for public-key directory servers and DNSSEC [11] ensures

integrity of DNS data. Ensuring integrity also in the *private* variants of these applications represents therefore an important security goal.

3.2.1 An Unsafe Approach

One naïve attempt to provide integrity in PIR would be to combine a standard integrity-protection mechanism, such as a digital-signature scheme, with a standard PIR protocol. We first sketch how one such scheme might work and then why it is *insecure*.

In this first-attempt construction, there is a *database owner*, who produces the database, and the *PIR server*, which respond to clients' PIR queries. For example, when using PIR in the context of DNS, the database owner is the entity that maintains the domain-name-system zone file (e.g., Verisign for the `.com` top-level domain), and the PIR server is the infrastructure provider (e.g., Amazon AWS). A client should be able to detect whether Amazon's PIR server is serving authentic DNS records from Verisign.

The naïve construction of verifiable PIR works as follows:

1. The database owner signs each record in the database with a digital-signature scheme. Each record in the authenticated database consists of (a) the original record, and (b) the database owner's signature on the record.

2. Via some out-of-band means, each PIR client obtains the database owner's public key for the digital-signature scheme, e.g., via a consensus or gossip.

3. The client runs a classic PIR protocol with the PIR server over the *authenticated database*. At the end of the protocol run, the client should hold (a) its desired record and (b) a digital signature on this record, with the database owner's secret key.

4. The client accepts the record if the signature, it received in Step 3, verifies against the database owner's public key that it received in Step 2. Otherwise the client rejects.

The practical problem with this approach is that, if the PIR server can learn (e.g., via some protocol-level side-channel) whether the client accepted or rejected the servers' response, a malicious server can violate client privacy. In particular, the single bit of information about whether the client accepted or rejected the data from the servers leaks information about the client's query.

An attack. The attack comes directly from Kushilevitz and Ostrovsky's early work on PIR [115]. Say that the PIR server is malicious and wants to learn whether the client is querying record i of the database. To do this, the server executes the PIR protocol faithfully with respect to a database in which the contents of record i are garbled (e.g., set to random bits).

If the client is fetching record i of the database, the client will *reject*, because the signature on record i will not verify. If the client is *not* fetching record i of the database, the client will *accept*, as the signatures for all records other than i are well-formed. Thus, the client's single accept/reject bit reveals whether the client is reading record i or not.

We find that, in a wide array of PIR applications, a malicious PIR server will be able to learn the client's accept/reject bit. For example, a naïve PIR client would retry its query upon a query-verification failure. In this case, the malicious PIR server would immediately learn the client's accept/reject bit. Alternatively, a client could never query a PIR server once they receive a corrupt response from the server. But this also leaks information to the server: the fact that a client makes a single query and then never queries again indicates a rejection.

In other applications of PIR, there are more direct side-channels: if a client uses PIR for DNS resolution, the pattern of queries after an initial query will indicate success or failure (e.g., if an initial query resolves to an alias/`CNAME` that the client would have to look up on the server).

The failure of a simple composition of a standard integrity-protection mechanism with a standard PIR scheme motivates our search for secure verifiable-PIR protocols.

3.3 Verifiable Single-Server PIR

In this section, we first define verifiable computational-PIR that provides a strong notion of integrity protection and then we give a concrete scheme that achieves the protection requirements.

3.3.1 Defining Verifiable PIR

A challenge of providing integrity in the single-server setting is that the client has no source of information about the database content other than the server itself. A malicious server can answer the client's query with respect to a database of the server's choosing, and hence control the client's output.

We address this by introducing a public database digest that cryptographically binds the server to given database contents and serves as the ground truth in the scheme. Using a digest is a standard integrity-protection technique [120], which we adapt to the PIR setting. The client obtains this digest via some out-of-band means, e.g., via gossip, as in CONIKS [130], or from the database owner if the latter is distinct from the PIR server.

In single-server verifiable-PIR, the client communicates with a single database server, as in standard single-server PIR schemes [115]. The client obtains also a succinct (i.e., constant-size) "digest" of the true database through some out-of-band means (e.g., a gossip protocol, as in CONIKS [130] or in certificate transparency [118]). The client accepts the output from the PIR

protocol only if the record that the client receives is consistent with its database digest. In our definition, the client either obtains "correct" output or aborts. The PIR privacy property holds even if the PIR server learns whether the client has aborted. Therefore, our protocol precludes the attack of Section 3.2.1.

Having outlined our approach intuitively, we now give the formal definition of a single-server verifiable PIR scheme. We assume that each database record consists of a single bit.

Definition 3 (Single-server verifiable PIR)**.** *A* single-server verifiable PIR *scheme, parameterized by a database size $n \in \mathbb{N}$, consists of the following algorithms:*

- $\mathsf{Digest}(1^\lambda, x) \to d$ *takes as input a security parameter λ (in unary) and a database $x \in \{0,1\}^n$ and returns a digest d.*
- $\mathsf{Query}(d, i) \to (\mathsf{st}, q)$ *takes as input a digest d and an index $i \in [n]$ and returns a client state* st *and a query q.*
- $\mathsf{Answer}(d, x, q) \to a$ *applies query q to database $x \in \{0,1\}^n$ with digest d and returns answer a.*
- $\mathsf{Reconstruct}(\mathsf{st}, a) \to \{0, 1, \perp\}$ *takes as input state* st *and answer a and returns a database bit or an error $\perp$.*

The distinction between the syntax of a verifiable PIR scheme and classic PIR is that here, the reconstruction algorithm might output an error symbol $\perp$. The client should output an error only if the server deviates from the protocol in a way that prevents the client from recovering its database bit of interest. The client should detect when a malicious server tries to tamper even with entries that the client did not retrieve.

A single-server verifiable-PIR protocol must satisfy the following properties. We informally state the properties here and give formal cryptographic definitions in Appendix A.2.

Correctness. Informally, a single-server verifiable-PIR scheme is *correct* if, when an honest client interacts with an honest server, the client always recovers its desired database record.

Integrity. A single-server verifiable-PIR scheme preserves *integrity* if, when an honest client interacts with a potentially malicious server that might arbitrarily deviate from the protocol, the client either outputs: its desired database bit or the error symbol $\perp$, except with negligible probability parametrized by λ.

Privacy. A single-server verifiable-PIR scheme satisfies *privacy* if a malicious server "learns nothing" about which database bit the client wants to fetch, even if the servers can also learn whether or not the client outputted the error symbol $\perp$.

If a single-server verifiable-PIR scheme satisfies the integrity and privacy properties, we say that the scheme is secure.

Construction 3 (Single-server verifiable PIR)**.** The construction is parametrized by a database length $n \in \mathbb{N}$, a group $\mathbb{G}$ of prime order $p \approx 2^{2\lambda}$ (on security parameter λ), and a hash function $H\colon [n] \to \mathbb{G}$. We model the group $\mathbb{G}$ as a generic group [170] and H as a random oracle [24]. We represent the database as a bit vector $x \in \{0,1\}^n \subseteq \mathbb{Z}_p^n$.

Digest$(x \in \{0,1\}^n) \to d \in \mathbb{G}$

1. Output $d \leftarrow \prod_{j=1}^{n} H(j)^{x_j} \in \mathbb{G}$.

Query $(d \in \mathbb{G}, i \in [n]) \to (\mathsf{st}, q)$

1. Sample two random values $r, t \xleftarrow{\$} \mathbb{Z}_p$.
2. For $j \in [n] \setminus \{i\}$, compute $h_j \leftarrow H(j)^r \in \mathbb{G}$.
3. For i, compute $h_i \leftarrow H(i)^{r+t} \in \mathbb{G}$.
4. Set $\mathsf{st} \leftarrow (i, d, r, t)$.
5. Set $q \leftarrow (h_1, \ldots, h_n) \in \mathbb{G}^n$.
6. Output (st, q).

Answer $\left(d \in \mathbb{G}, x \in \{0,1\}^n \subseteq \mathbb{Z}_p^n, q\right) \to (a \in \mathbb{G})$

1. Parse the query q as $(h_1, \ldots, h_n) \in \mathbb{G}^n$.
2. Output $a \leftarrow \prod_{j=1}^{n} h_j^{x_j} \in \mathbb{G}$.

Reconstruct $(\mathsf{st}, a) \to \{0, 1, \perp\}$

1. Parse the state st as (i, d, r, t).
2. Set $m \leftarrow d^{-r} \cdot a \in \mathbb{G}$.
3. If $m = 1_{\mathbb{G}}$, output "0." If $m = H(i)^t$, output "1."
   Otherwise, output $\perp$.

3.3.2 A Practical Construction

Now, we present a practical construction for single-server verifiable-PIR. As described in section 3.3.1, we address the lack of the ground truth by introducing a public database digest. The client accepts the protocol's output only if the output is consistent with the database that the digest represents.

Construction overview. The construction (Construction 3) makes use of a group $\mathbb{G}$ of large prime order p. The database is a vector of n bits $x \in \{0,1\}^n$. The public parameters of the scheme include a hash function $H\colon [n] \to \mathbb{G}$. Since we prove security in the generic-group model [170], we model H has an oracle that outputs a random group element [24].

To compute the database digest, we use the hash function H to associate a random group element $g_j \leftarrow H(j) \in \mathbb{G}$ with each bit $x_j \in \{0,1\}^n$ of the database. The digest is then the product $d \leftarrow \prod_{j=1}^{n} g_j^{x_j} \in \mathbb{G}$. Finding two distinct databases that map to the same digest is as hard as solving the discrete-log problem in $\mathbb{G}$ [151].

The protocol operates as follows. First, the client samples two random values $r, t \xleftarrow{\$} \mathbb{Z}_p$. The client then prepares a vector of n group elements. Say the client wants to fetch the ith database bit. For $j \in \{1,\ldots,n\}$, the jth component of this vector is $h_j \leftarrow H(j)^{r+t}$ if $i = j$ and is $h_j \leftarrow H(j)^r$ otherwise.

The client then sends the resulting blinded vector $(h_1,\ldots,h_n)$ to the server as the query. The server exponentiates each element in the vector to the corresponding database bit and computes the product of the resulting elements. The key insight is that, if the server honestly executes the protocol, the client receives back the product of the blinded digest and (a) either the group identity (then, the retrieved bit is zero) or (b) the blinded group element of interest (the retrieved bit is one). If the server returns any answer to the client other than the one prescribed by the protocol, the client will detect this and reject with overwhelming probability.

Rebalancing to reduce communication to $\sqrt{n}$. The PIR scheme of construction 3 has a digest consisting of one group element, and it requires the client to upload n group elements to the server and download a single group element from the server. By rebalancing upload and download, we can reduce the total communication to $O(\sqrt{n})$ group elements.

Essentially, the server splits the database into $\sqrt{n}$ chunks, each of size $\sqrt{n}$. The digest then consists of the hash (with any collision-resistant hash function, e.g., SHA256) of the $\sqrt{n}$ database digests. To query the database for the ith row of the jth chunk, the client issues a single query for row i. The server responds to the query with the $\sqrt{n}$ chunk digests, and the answer computed against each of the chunks. The client checks that (1) the hash of the $\sqrt{n}$ chunk digests match the database digest and (2) all $\sqrt{n}$ chunk queries accept. If these checks pass, the client outputs the value of the jth response as their answer.

We then have, by rebalancing construction 3:

Theorem 4. *In the generic-group model [170], there exists a verifiable single-server PIR scheme for n-bit databases with digest size $O_\lambda(1)$ bits and total communication $O_\lambda(\sqrt{n})$, where $O_\lambda(\cdot)$ suppresses polynomials in the security parameter λ.*

Theorem 4 proves security in the generic-group model [170], which captures security against attackers that make only black-box use of the underlying group. We suspect that it is also possible to prove security of Construction 3 from standard assumptions, such as the decision Diffie-Hellman assumption [33].

Limitations. The primary limitation of the presented scheme is that it cannot be easily extended to efficiently handle large database rows. While there are applications for single-bit

verifiable single-server PIR scheme (§4.6), we still hope that it is possible to construct a more bandwidth- and computation-efficient scheme in the future. We unsuccessfully attempted to combine a (non-verifiable) classic single-server PIR scheme with some sort of algebraic integrity-protection mechanism, but it seems non-trivial to provide our integrity properties while making only black-box use of the underlying single-server PIR scheme. Further investigation along these lines would be an interesting task for future work.

3.4 Experimental Evaluation

In this section, we experimentally evaluate our verifiable PIR scheme by comparing it with a state-of-the-art single-server PIR scheme without integrity protection.

Implementation. We implemented our verifiable-PIR scheme in roughly 800 lines of Go. We rely on the CIRCL library [76] for the group operations in our scheme. In addition, we implemented a single-server non-verifiable PIR scheme in another 300 lines of Go to use it as the baseline in our experiments. The baseline scheme is lattice-based, and it uses Lattigo's implementation [117] of the Brakerski-Fan-Vercauteren (BFV) cryptosystem [39], [75]. The scheme minimizes the size of client's query by sending a single bit per database block. Lattigo supports this minimization via ciphertext packing [8], [138]. Applied to PIR, the packing technique enables a client to place the query bits into a single ciphertext that the server later expands into one ciphertext per packed bit using special rotation keys that the client generates and sends in addition. In both verifiable-PIR and no-integrity PIR schemes, we implemented rebalancing such that the client retrieves a whole database column and locally selects the record of the correct row.

We will make our implementation publicly available under an open-source license.

Parameters. Our verifiable-PIR scheme uses the P256 elliptic curve, hence each database row is an array of 32-byte elements. We use ($N = 8192$, $\log(Q_P) = 218$, $t = 2^{16}$) as the lattice parameters for the non-verifiable scheme; that is, plaintexts and ciphertexts are polynomials of degree $N = 8192$, and $\log t = 16$ bits of data can be encoded per polynomial coefficient, thus one plaintext can hold 16 KiB of data. The lattice expansion factor with these parameters is 24: encrypting 16 KiB of data results in a 384 KiB ciphertext. For the sizes and bandwidth overhead, we use the power of two notation, i.e., 1 KiB = 1024 B, 1 MiB = 1024^2 B, etc.

Experimental methodology. We perform the experiments on a single machine equipped with two Intel Xeon E5-2680 v3 (Haswell) CPUs, each with 12 cores, 24 threads, and operating at 2.5 GHz. The machine has 256 GB of RAM, and runs Ubuntu 20.04 and Go 1.16. We run the experiments with all the available cores and report the sum of the CPU time across all the cores. We run the experiments twenty times and report the median of all executions. We minimize the effect of Go's garbage collector by increasing the garbage-collection target

percentage (`GOGC` = 8000). We have published our experimental code in our source-code repository.

DB [bits]	W/o integrity	Verifiable	Overhead
	CPU time [sec]		
1 M	1.2	16	13×
10 M	7	160	24×
100 M	60	1,561	26×
1 B	668	15,769	24×
	Bandwidth [MiB]		
Setup	37.5	$3 \cdot 10^{-5}\sqrt{DB}$	
1 M	1.5	0.06	0.04×
10 M	3.8	0.2	0.05×
100 M	11	0.6	0.06×
1 B	33	2.0	0.06×

Table 3.1: The cost of retrieving one data bit when using lattice-based classic PIR without integrity protection and when using our verifiable-PIR scheme (Section 3.3.2).

3.4.1 Microbenchmarks

To evaluate our scheme, we measure its computation and communication performance, and compare it with lattice-based classic PIR.

While verifiable-PIR scheme for single-bit databases might not seem terribly useful at first glance, there are at least a few important potential applications of a such a PIR scheme. For example, consider the problem of *private contact discovery* [128]: a client using an encrypted messaging app (e.g., WhatsApp) would like to learn which of her friends use the app as well, without revealing her list of contacts to the app's servers.

As Table 3.1 shows, retrieving one bit from a database of 1 M bits takes 16 core-seconds and 64 KiB of bandwidth. The cost grows linearly with the database size. While this performance is modest, it is comparable with the state-of-the-art contact-discovery protocols that exhibit either seconds-range online time (1.3s) [108] or expensive server setup (7s or 240s depending on the protocol) [102]. Contact discovery with our scheme will take seven core-minutes in Signal (40 M users [54]) and fifty core-minutes in WhatsApp (2 B users [188]).

The advantage of lattice-based classic PIR over our scheme is that the same database is more compactly encoded (e.g., we encode 1 M bits into eight polynomials of 16 KiB each), thus processing it requires fewer public-key operations. As a result, it computationally performs 25× better than our scheme. However, retrieving one bit of information requires downloading the whole lattice plaintext, which leads to higher bandwidth overhead. The setup cost consists of sending upstream the rotation keys for ciphertext expansion [138] in the classic PIR scheme

and of downloading database digests in our verifiable-PIR scheme. Note that this classic PIR baseline is not fully optimized, e.g., SealPIR [8] achieves almost 100× higher throughput. This is primarily due to our conservative choice of lattice parameters (as suggested by the Lattigo library that we use), i.e., the polynomial degree N and $\log(Q_P)$ in our implementation are 4× larger than in SealPIR, which results in costlier operations, and SealPIR embeds 23 data bits per polynomial coefficient, whereas we embed only 16 bits and due to the programming language of the implementation (C++ in SealPIR vs. Go in Lattigo and our scheme implementation).

3.5 Related Work

The closest work to ours is on providing verifiability in PIR on an outsourced database in the multi-server setting [197] and, as a follow-up, in the single-server setting [186]. Building on the concept of signatures of correct computation [149], both papers propose schemes where the data owner generates a proving key and a verification key for a given database, which cloud-provider servers can use to certify that they correctly computed the answers to the client's queries and did it for that database. The limitation of these schemes is their prohibitive computational cost. Even the typically cheaper multi-server scheme [197] has quadratic computational complexity where the computations are primarily bilinear pairings.

While we are not aware of any other work that handles data integrity in the single-server PIR setting, there is a line of work that addresses Byzantine failures in the multi-server setting. This direction was initiated by Beimel and Stahl [18], [19] who showed that one could achieve Byzantine robustness by replacing simple additive secret sharing with threshold Shamir secret sharing [169]. This robustness guarantees that a client can still recover the correct database row even if a subset of servers is malicious or has failed. Follow-up work [66], [86] used list-decodable codes to harden the initial scheme so that it ensured user privacy and protocol correctness with a higher number of colluding servers. The problem that the Byzantine schemes solve slightly differs from ours. Specifically, we aim at the client being able to detect any server misbehavior and abort, where the Byzantine schemes enable the client to recover the database row if even a subset of servers misbehave. Moreover, the idea of having a subset of malicious server is not applicable to the single-server setting.

Practitioners have also recognized and tried to address the problem of integrity in PIR. In DORY [58], an encrypted search system with decentralized trust, Dauterman et al. consider the threat model of potentially malicious servers that respond to keyword search requests via PIR The authors use aggregate MACs to protect integrity in this application. In PIR-Tor [136], Mittal et al. protect against active attacks by requiring the Tor directory authorities to collectively sign each data block (using the BLS signature scheme). Both approaches are designed for multi-server PIR, however, and do not protect against selective manipulation in the single-server setting. Finally, one can achieve integrity in data retrieval by implementing the system

as an enclave-based Oblivious RAM [135], if the trust assumptions of hardware enclaves are acceptable for a given use case.

3.6 Conclusion

We have argued that data integrity in PIR can be crucial for safety and even privacy of users but that it is also difficult to achieve, especially in the single-server setting. As a response, we have introduced verifiable single-server PIR, which enhances the strong privacy properties of classic PIR with strong data-integrity guarantees. Our concrete verifiable-PIR scheme is discrete-log based and takes advantage of a database digest. Although our scheme is limited to single-bit operations, it can find application to the problems where the protocol's output is binary, e.g., in private contact discovery.

4 Securing Retrieval of Software Updates

One characteristic that distinguishes software updates from many other types of content that users retrieve from the Internet is that software updates come typically as binary files that users directly install on their machines. Be it a malware signed with a stolen but valid vendor key or a specific software modification with a backdoor,in most cases, a user is unable to detect the attack at this stage. The power of compromising software-release workflow over compromising directly a user's machine is that, in this setting, an adversary can gain control over computers of the whole user base by breaking into a single system.

In this chapter, we delve into the security of software-updates systems. We begin by describing how a typical software-release workflow operates and by identifying its flaws. Then, we propose our solution that checks all the requirements of a modern secure system.

4.1 Modern Software-Release Workflow and Its Issues

Let us begin our study of the topic by describing a typical software-release workflow. The workflow is depicted in Figure 4.1. Any software product begins with a team of developers. This team can be either independent or part of a larger company. They develop the source code of the product and periodically release software updates. When a new release is ready, someone from the team, e.g., the manager, transfers the release's source code to a dedicated build server where the code is compiled to binary files, possibly one for each supported platform. Either the team's manager or all the developers have access to a master signing key that they use to authenticate the compiled binaries by computing a signature for each of them. Note here the trust assumption of correct compilation at the build server. If someone compromises the build server and substitutes at the last moment the compiled binaries with their maliciously modified versions, the developers will be unlikely to notice the substitution and will still authenticate the release files. This exact discrepancy was at the core of some recent software supply-chain attacks, e.g., the attack on SolarWinds [78] and the NotPetya attack [125]. After authenticating the update release, the developers send it to a software-update center from where end users can download it. Having downloaded the update, the users verify its signature

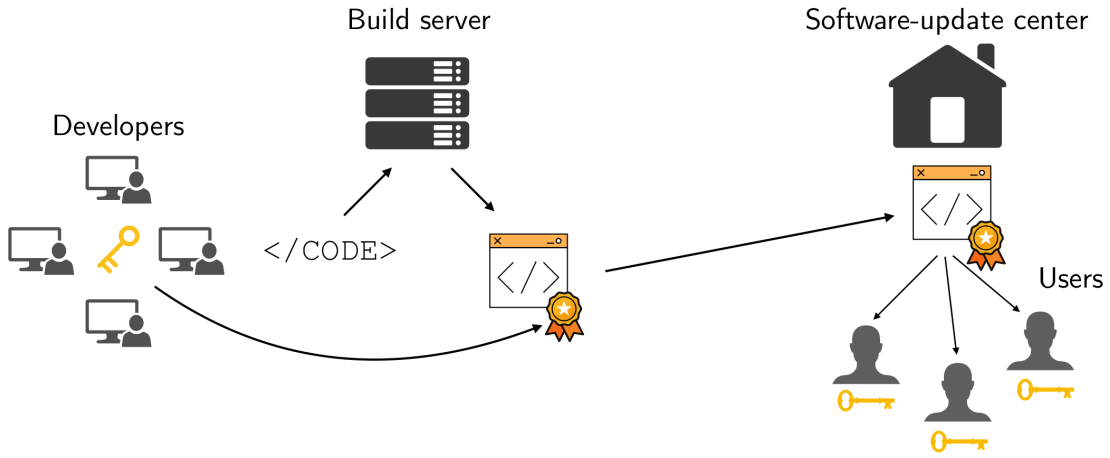

Figure 4.1: Typical workflow of a software-update release. Developers prepare source code for a release and send it to a build server for compilation. Then, the developers sign the compiled binary with a master key and send the update to a software-update center. Users download the update from the center or its mirror, verify the developers' signature, and install the update on their machines.

(the developers' public key is typically either installed with the first version of the software or supported by a public-key certificate) and finally install it.

There are four main problems with the workflow described above. First, the integrity and authenticity of the updates traditionally depends on a single signing key, prone to accidental exposure [100], [133] and theft [80], [157]. Having proper protection for signing keys to defend against such single points of failure is therefore a top priority. Second, the lack of transparency mechanisms in the current infrastructure of software distribution leaves room for equivocation and stealthy backdooring of updates by compromised [44], [119], coerced [27], [79], [171], and malicious [97] software vendors and distributors. Recent works on reproducible software-builds [126], [153] attempt to counteract this deficit by improving on the source-to-binary correspondence. However, in its current form, it is unsuitable for widespread deployment, as rebuilding packages puts a high burden on end users (e.g., building a complex software, such as Tor Browser, can take hours on a modern laptop). Third, attackers might execute a man-in-the-middle attack on the connections between users and update providers (e.g., with DNS cache poisoning [172] or BGP hijacking [13]), thus enabling themselves to mount replay and freeze attacks [44] against their targets. To prevent attackers from exploiting unpatched security vulnerabilities as a consequence of being targeted by one of the above attacks [177], clients must be able to verify timeliness of updates. Finally, revoking and renewing signing keys (e.g., in reaction to a compromise) and informing all their clients about these changes is usually cumbersome. Hence, modern software-update systems should provide efficient and secure means to evolve signing keys and should enable client notification in a timely manner.

To address these problems, we introduce CHAINIAC, a decentralized software-update framework that removes single points of failure, enforces transparency, ensures integrity and authen-

ticity, and retains efficient verifiability of the software-release process. Here, we describe the main rationale behind CHAINIAC' design, and we give the detailed architecture in Section 4.4.

For the first step in CHAINIAC, we replace the single-signing-key approach with a decentralized model where each developer involved in the project has an individual signing key, e.g., a PGP key, and she contributes to vetting of each release. A release is considered approved if a majority of the involved developers have signed off on it to express their approval. The idea is rather simple: compromising multiple keys or coercing multiple developers at once is harder than stealing a single shared key.

Having to verify multiple signatures is, however, a burden for end clients, both in terms of efficiency and because their installed software would have to keep track of all the individual keys that change over time. In addition to these inconveniences, the transparency issue between the source code the developers sign and the binary files that are distributed to users is still present. To resolve these issues, we introduce two third-party server groups, *witness servers* and *build verifiers* that validate the developers' approval of a release and its source-to-binary correspondence and that produce a collective attestation that end users can efficiently verify. The decentralization of trust and the compression of the authentication information are the core principles here: Multiple independent servers check the validity of a release and collectively generate succinct authentication information for it.

CHAINIAC increases transparency and ensures accountability of the update process by implementing a public update-timeline that comprises a release log, freshness proofs, and key records. This mechanism guarantees that all users have a consistent view of the update history, preventing adversaries from stealthily attacking targeted clients with compromised updates. Even if an attacker manages to slip a backdoor into the source code, the corresponding signed binary stays publicly available for scrutiny, thereby preventing secret deployment against targeted users. We achieve tamper evidence of the timeline by employing skipchains: authenticated data structures that enable clients to efficiently navigate arbitrarily long update timelines, both forward (e.g., to validate a new software release) and backward (e.g., to downgrade or verify the validity of older package-dependencies needed for compatibility).

Here, before diving into the details of CHAINIAC, let us provide some necessary background.

4.2 Background

In this section, we give an overview of the concepts and notions CHAINIAC builds on, this includes scalable collective signing, reproducible builds, software-update systems, skipchains, and decentralized consensus.

4.2.1 Collective Signing and Timestamping

CoSi [174] is a protocol for large-scale collective signing. Aggregation techniques and communication trees [64], [179] enable CoSi to efficiently produce compact multi-signatures and to scale to thousands of participants. A complete group of signers, or *witnesses*, is called a collective authority or *cothority*. CoSi assumes that signature verifiers know the public keys of the witnesses, all of which are combined to form an aggregate public key of the cothority. If witnesses are offline during the collective signing process or refuse to sign a statement, the resulting signature includes metadata that documents the event.

In CHAINIAC, we rely on CoSi for efficient collective signing among a large number of witnesses. Furthermore, we use the witness-cosigned timestamp service [174] as a building block in our design for the protection of clients against replay and freeze attacks [44] (where clients are blocked from learning about the availability of new software updates by an adversary). We describe the design of the protection mechanism in Section Section 4.4.6.

4.2.2 Reproducible Builds

Ensuring that source code verifiably compiles to a certain binary is difficult in practice, as there are often non-deterministic properties in the build environment [126], [153], which can influence the compilation process. This issue poses a variety of attack vectors for backdoor insertion and false security-claims [97]. Reproducible builds are software development techniques that enable users to deterministically compile a given source code into one same binary, independent of factors such as system time or build machines. An ongoing collaboration of projects [158] is dedicated to improving these techniques, e.g., as of June, 2021, Debian claims that 95% of its packages in the testing suite are reproducible [59], amounting to ~29,600 packages. To provide a source-to-binary attestation as one of the guarantees, CHAINIAC relies on software projects to adopt the practices of reproducible builds.

4.2.3 Roles in Software-Update Systems

The separation of roles and responsibilities is one of the key concepts in security systems. TUF [166] and its successor, Diplomat [113], are software-update frameworks that make update systems more resilient to key compromise by exploiting this concept. In comparison to classic systems, these frameworks categorize the tasks that are commonly involved in software-update processes and specify a responsible role for every category. Each of these roles is then assigned a specific set of capabilities and receives its own set of signing keys, which enables TUF and Diplomat to realize different trade-offs between security and usability. For example, frequently used keys with low-security risks are kept online, whereas rarely needed keys with a high-security risk are kept offline, making it harder for attackers to subvert them. To achieve, for each role, the sweet-spot between security and usability, we follow a similar delegation model in our multi-layered architecture in Section 4.4.6. However, we decentralize all these roles,

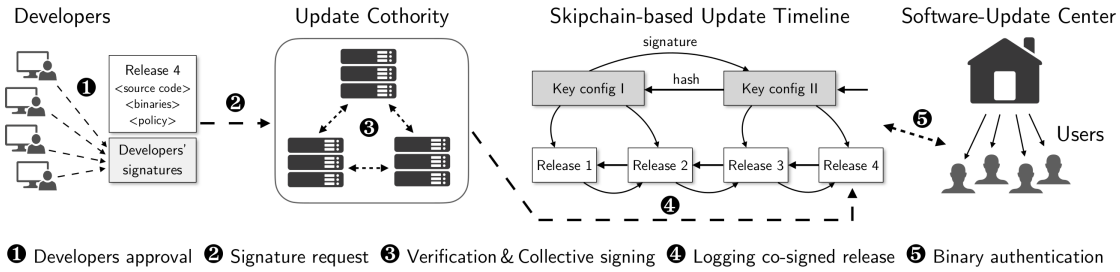

Figure 4.2: Architectural overview of CHAINIAC. The developers sign release data and send it to the update cothority. The update cothority collectively verifies the release against the policy and, if the checks pass, appends the co-signed release information to the update timeline. When users retrieve a software release form an untrusted software-update center, they consult the update timeline to verify the release's validity.

use a larger number of keys, and log their usage and evolution to further enhance security and add transparency.

4.2.4 Skipchains and Consensus

Skipchains [142] are authenticated data structures that combine ideas from blockchains [110], [140] and skiplists [139], [156]. Skipchains enable clients (1) to securely traverse the timeline in both forward and backward directions and (2) to efficiently traverse short or long distances by employing multi-hop links. Backward links are cryptographic hashes of past blocks, as in regular blockchains. Forward links are cryptographic signatures of future blocks, which are added retroactively when the target block appears. We use skipchains in CHAINIAC to implement tamper-resistant release log and a multi-layer role-based architecture. Skipchain's forward links facilitate trust delegation and efficient log traversal.

CHAINIAC uses *BFT-CoSi* [111] as a consensus algorithm to ensure a single consistent timeline, e.g., while rotating signing keys. BFT-CoSi implements PBFT [45] by using collective signing [174] with two CoSi-rounds to realize PBFT's prepare and commit phases.

4.3 System Overview

In this section, we state high-level security goals that a hardened software-update system should achieve, we introduce a system and threat model, and we present an architectural overview of our proposed framework.

4.3.1 Security Goals

To address the challenges listed in Section 4.1, we formulate the following security goals for CHAINIAC:

1. **No single point of failure:** The software-update system should retain its security guarantees in case any single one of its components fails (or gets compromised), whether it is a device or a human.
2. **Source-to-binary affirmation:** The software-update system should provide a high assurance-level to its clients that the deployed binaries have been built from trustworthy and untampered source code.
3. **Efficient release-search and verifiability:** The software-update system should provide means to its clients to find software release (the latest or older ones) and verify its validity in an efficient manner.
4. **Linear immutable public release history:** The software-update system should provide a globally consistent tamper-evident public log where each software release corresponds to a unique log entry that, once created, cannot be modified or deleted.
5. **Evolution of signing keys:** The software-update system should enable the rotation of authoritative keys, even when a (non-majority) subset of the keys is compromised.
6. **Timeliness of updates:** Clients should be able to verify that the software indeed corresponds to the latest one available.

4.3.2 System and Threat Model

In the system model, we introduce terminology and basic assumptions; and, in the threat model, potential attack scenarios against CHAINIAC.

System model. *Developers* write the source code of a software project and are responsible for approving and announcing new project *releases*. Each release includes source code, binaries (potentially, for multiple target architectures), and metadata such as release description. A *snapshot* refers to a set of releases of different software projects at a certain point in time. Projects can have single or multiple packages. *Witnesses* are servers that can validate and attest statements. They are chosen by the developers and should be operated ideally by both developers and independent trusted third parties. Witnesses are trusted as a group but not individually. *Build verifiers* are a subset of the witnesses who execute, in addition to their regular witness tasks, reproducible building of new software releases and compare them to the release binaries. Witnesses and build verifiers jointly form an *update cothority* (collective authority). The *update timeline* refers to a public log that keeps track of the authoritative signing keys, as well as the software releases. *Users* are clients of the system; they receive software releases through an (untrusted) *software-update center*.

Threat model. We assume that a threshold t_d of n_d developers are honest, meaning that less than t_d are compromised and want to tamper with the update process. We further assume that a threshold t_w of n_w witness servers is required for signing, whereas at most $f_w = n_w - t_w$ witnesses can potentially be faulty or compromised. To ensure consistency and resistance to fork attacks, CHAINIAC requires $n_w \geq 3f_w + 1$, hence, $t_w >= 2f_w + 1$. If this property is violated, CHAINIAC does not guarantee single history of the update timeline, however, even

then, each history will individually be valid and satisfy the other correctness and validation properties, provided fewer than t_w witnesses are compromised. Furthermore, a threshold t_v of n_v build verifiers is honest and uses a trustworthy compiler [175] such that malicious and legitimate versions of a given source-code release are compiled into different binaries. Software-update centers and mirrors might be partially or fully compromised. Moreover, a powerful (e.g., state-level) adversary might try to target a specific group of users by coercing developers or an update center to present to his targets a malicious version of a release. Finally, we assume that users of CHAINIAC are able to securely bootstrap, i.e., receive the first version of a software package with a hard-coded initial public key of the system via some secure means, e.g., pre-installed on a hard drive, on a read-only media, or via a secure connection.

An attack on the system is successful if an attacker manages to accomplish at least one of the following:

- Make developers sign the source code that they do not approve.
- Substitute a release binary with its tampered version such that the update cothority signs it.
- Trick the update cothority into signing a release that is not approved by the developers.
- Create a valid fork of the public release history or modify/revoke its entries; or present different users with different views of the history.
- Trick an outdated client into accepting a bogus public key as a new signing key of the update cothority.
- Get a client to load and run a release binary that is not approved by the developers or validated by the update cothority.

4.3.3 Architecture Overview

An illustration of CHAINIAC, showing how its various components interact with each other, is given in Figure 4.2. To introduce CHAINIAC, we begin with a simple strawman design that most of today's software-update systems use, and we present a roadmap for evolving this design into our target layout. Initially, we assume that only a single, static, uncompromisable cryptographic key pair is used to sign/verify software releases. The private key might be shared among a group of developers, and the public key is installed on client devices, e.g., during a bootstrap. To distribute software, one of the developers builds the source code and pushes the binary to a trusted software-update center from where users can download and install it. This strawman system guarantees that users receive authenticated releases with a minimal verification overhead.

This design, though common, is rife with precarious assumptions. Expecting the signing key to be uncompromisable is unrealistic, especially if shared among multiple parties, as attackers need to subvert only a single developer's machine to retrieve the secret key or to coerce only one of the key owners. For similar reasons, it is Utopian to assume that the software-update center is trustworthy. Moreover, without special measures, it is hard to verify that the binaries

were built from the given (unmodified) source code, as the compilation process is often influenced by variations in the building-environment, hence non-deterministic. If an attacker manages to replace a compiled binary with its backdoored version, before it is signed, the developers might not detect the substitution and unknowingly sign the subverted software.

Eliminating these assumptions creates the need to track a potentially large number of dynamically changing signing keys; furthermore, checking a multitude of signatures would incur large overheads to end users who rarely update their software. To address these restrictions, we transform the strawman design into CHAINIAC in six steps:

1. To protect against a single compromised developer, CHAINIAC requires that developers have individual signing keys and that a threshold of the developers sign each release, see step ① in Figure 4.2.
2. To be able to distribute verified binaries to end users, we introduce developer-signed reproducible builds. Although users still need to verify multiple signatures, they no longer need to build the source code.
3. To further unburden users and developers, we use a cothority to validate software releases (check developer signatures and reproducible binaries) and collectively sign them, once validated: steps ② and ③ in Figure 4.2.
4. To protect against release-history tampering or stealthy developer-equivocation, we adopt a public log for software releases in the form of collectively signed decentralized hash chains, see step ④ in Figure 4.2.
5. To enable efficient key rotation, we replace hash chains with skipchains, blockchain-like data structures that enable forward linking and decrease verification overhead by multi-hop links.
6. To ensure update timeliness and further harden the system against key compromise, we introduce a multi-layer skipchain-based architecture that, in particular, implements a decentralized timestamp role.

4.4 Design of CHAINIAC

In this section, we present CHAINIAC in detail. For clarity of exposition, we describe CHAINIAC step-by-step starting from a strawman update-system that uses one key to sign release binaries, as introduced in Section 4.3. We begin by introducing a decentralized validation of both source code and corresponding binaries, while alleviating the developer and client overhead. We then improve transparency and address the evolution of update configurations by using skipchains. Finally, we reduce traversal overheads with multi-level skipchains and demonstrate how to adapt CHAINIAC to multi-package projects.

4.4.1 Decentralized Release-Approval

The first step towards CHAINIAC involves enlarging the trust base that approves software releases. Instead of using a single (shared) key to sign updates, each software developer signs using their individual keys. At the beginning of a project, the developers collect all their public keys in a *policy* file, together with a threshold value that specifies the minimal number of valid developer signatures required to make a release valid. Complying with our threat model, we assume that this policy file, as a trust anchor, is obtained securely by users at the initial acquisition of the software, e.g., it can reside on a project's website as often is the case with a single signing key in the current software model.

Upon the announcement of a software release, which can be done by a subset or all developers depending on the project structure, all the developers check the source code and, if they approve, they sign the hash of it with their individual keys, e.g., using PGP [42], and they add the signatures to an append-only list. Signing source code, instead of binaries, ensures that developers can realistically verify (human-readable) code.

The combination of the source code and the signature list is then pushed to the software-update center from where a user can download it. For simplicity, we first assume that the update center is trusted, later relaxing this assumption. When a user receives an update, she verifies that a threshold of the developers' signatures is valid, as specified in the policy file already stored on user's machine. If so, the user builds the binary from the obtained source code and installs it. An attacker trying to forge a valid software-release needs to control the threshold of the developers' keys, which is presumably harder than gaining control over any single signing key.

4.4.2 Build Transparency via Developers

The security benefits of developers signing source-code releases come at the cost of requiring users to build the binaries. This cost is a significant usability disadvantage, as users usually expect to receive fully functional binaries directly from the software center. Therefore, in our second step towards CHAINIAC, we transfer the responsibility of building binaries from users to developers.

When a new software release is announced, it includes not only the source code but also a corresponding binary (or a set of binaries for multiple platforms) that users will obtain via a software center. Each developer now first validates the source code, then compiles it using reproducible build techniques [126], [153]. If the result matches the announced binary, he signs the software release. Assuming a threshold of developers is not compromised, this process ensures that the release binary has been checked by a number of independent verifiers. Upon receiving the update, a user verifies that a threshold of signatures is valid; if so, she can directly install the binary without needing to build it herself.

4.4.3 Release-Validation via Cothority

Although decentralized developer approval and reproducible builds improve software-update security, running reproducible builds for each binary places a high burden on developers (e.g., deterministically building the Bitcoin Core client can take "a long time" due to all the dependencies being built for all architectures and operating systems [12]). The load becomes even worse for developers involved in multiple software projects. Moreover, verifying many developer-signatures in large software projects can be a burden for client devices, especially when upgrading multiple packages. It would naturally be more convenient for an intermediary to take the developers' commitments, run the reproducible builds and produce a result that is easily verifiable by clients. Using a trusted third party is, however, contrary to CHAINIAC's goal of decentralization. Hence to maintain decentralization, we implement the intermediary as a collective authority or *cothority*.

To announce a new software release, the package developers combine the hashes of the associated source-code and binaries in a Merkle tree [131]. Each developer checks the source code and signs the root hash (of this tree), that summarizes all data associated with the release. The developers then send the release data and the list of their individual signatures to the cothority that validates and collectively signs the release. Clients can download and validate the release's source and/or any associated binary by verifying only a single collective signature and Merkle inclusion proofs for the components of interest.

To validate a release, each cothority server checks the developer signatures against the public keys and the threshold defined in the policy file. Remembering the policy for each software project is a challenge for the cothority that is supposed to be stateless. For now, we assume that each cothority member stores a project-to-policy list for all the projects it serves for. We relax this assumption in Section 4.4.5. The build verifiers then compile the source code and compare the result against the binaries of the release. The latter verification enables the transition from reproducible builds to *verified builds*: a deployment improvement over reproducible builds, which we introduce. The verified builds enable clients to obtain the guarantee of source-to-binary correspondence without the need to accomplish the resource-consuming building work, due to the broad independent validation.

4.4.4 Anti-equivocation Measures

Many software projects are maintained by a small group of (often under-funded or volunteer) developers. Hence, it is not unreasonable to assume that a powerful (state-level) attacker could coerce a threshold of group members to create a secret backdoored release used for targeted attacks. In our next step towards CHAINIAC, we tackle the problem of such stealthy developer-equivocation, as well as the threat of an (untrusted) software-update center that accidentally or intentionally omits parts of the software release history.

We introduce cothority-controlled hash chains that create a public history of the releases for each software project. When a new release is announced, the developers include and sign the summary (Merkle Root) of the software's last version. The cothority then checks the developers' signatures, the collective signature on the parent hash-block, and that there is no fork in the hash-chain (i.e., that the parent hash-block is the last one publicly logged and that there is no other hash-block with the same parent). If everything is valid, it builds the summary for the current release, then runs BFT-CoSi [111] to create a new collective signature. Because the hash chain is cothority controlled, we can distribute the witnessing of its consistency across a larger group: for example, not just across a few servers chosen by the developers of a particular package, but rather across all the servers chosen by numerous developers who contribute to a large software distribution, such as Debian. Even if an attacker controls a threshold of developer keys for a package and creates a seemingly valid release, the only way to convince any client to accept this malicious update is to submit it to the cothority for approval and public logging. As a result, it is not possible for the group to sign the compromised release and keep it "off the public record".

This approach prevents attackers from secretly creating malicious updates targeted at specific users without being detected. It also prevents software-update centers from "forgetting" old software releases, as everything is stored in a decentralized hash chain. CHAINIAC's transparency provisions not only protect users from compromised developers, but can also protect *developers* from attempts of coercion, as real-world attackers prefer secrecy and would be less likely to attack if they perceive a strong risk of the attack being publicly revealed.

4.4.5 Evolution of Authoritative Keys

So far, we have assumed that developer and cothority keys are static, hence clients who verify (individual or collective) signatures do not need to rely on centralized intermediaries, such as Certificate Authorities, to retrieve those public keys. This assumption is unrealistic, however, as it makes a compromise of a key only a matter of time. Collective signing exacerbates this problem, because for both maximum independence and administrative manageability, witnesses' keys might need to rotate on different schedules. To lift this assumption without relying on centralized Certificate Authorities, we construct a decentralized mechanism for trust delegation that enables the evolution of the keys. As a result, developers and cothorities can change, when necessary, their signing keys and create a moving target for an attacker, and the cothority becomes more robust to churn.

To implement this trust delegation mechanism, we employ skipchains [142] (see Section 4.2.4 for our overview). For the cothority keys, each cothority configuration becomes a block in a skipchain. When a new cothority configuration needs to be introduced, the current cothority witnesses run BFT on it. If completed successfully, they add the configuration to the skipchain, along with the produced signature as a forward link. For the developer keys, the trust is rooted in the policy file. To enable a rotation of developer keys, a policy file needs to be a part of

the Merkle tree of the release, hence examined by the developers. Thus, the consistency of key evolution becomes protected by the hash chain. To update their keys, the developers first specify a new policy file that includes an updated set of keys, then, as usual during a new release, they sign it with a threshold of their current keys, thus delegating trust from the old to the new policy. Once the cothority has appended the new release to the chain, the new keys become active and supersede their older counterparts. Anyone following the chain can be certain that a threshold of the developers has approved the new set of keys. With this approach, developers can rotate their keys regularly and, if needed, securely revoke a sub-threshold number of compromised keys.

4.4.6 Role Separation and Timeliness

In addition to verifying and authenticating updates, a software-update system must ensure update timeliness, so that a client cannot unknowingly become a victim of freeze or replay attacks (see Section 4.2.1). To retain decentralization in CHAINIAC, we rely on the update cothority to provide a timestamp service. Using one set of keys for signing new releases and for timestamping introduces tradeoffs between security and usability, as online keys are easier compromisable than offline keys, whereas the latter cannot be used frequently. To address the described challenges, we introduce a multi-layer skipchain-based architecture with different trust roles, each having different responsibilities and rights. We distinguish the four roles ROOT, CONFIG, RELEASE, and TIME. The first three are based on skipchains and interconnected with each other through *upward* and *downward* links represented as cryptographic hashes and signatures, respectively. Figure 4.3 shows an overview of this multi-layer architecture.

The ROOT role represents CHAINIAC's root of trust; its signing keys are the most security-critical. These keys are kept offline, possibly as secrets shared among a set of developer-administrators. They are used to delegate trust to the update cothority and revoke it in case of misbehavior. The ROOT skipchain changes slowly (e.g., once per year), and old keys are deleted immediately. As a result, the ROOT skipchain has a height of one, with only single-step forward and backward links.

The CONFIG role represents the online keys of the update cothority and models CHAINIAC's control plane. These keys are kept online for access to them quicker than to the ROOT keys. Their purpose is to attest to the validity of new release-blocks. The CONFIG skipchain can have higher-level skips, as it can be updated more frequently. If a threshold of CONFIG keys is compromised, the ROOT role signs a new set of CONFIG keys, enabling secure recovery. This is equivalent to a downward link from the ROOT skipchain to the CONFIG skipchain.

The RELEASE role wraps the functionality of the release log, as specified previously, and adds upward links to ROOT and CONFIG skipchains, enabling clients to efficiently look up the latest trusted ROOT and CONFIG configurations required for verifying software releases.

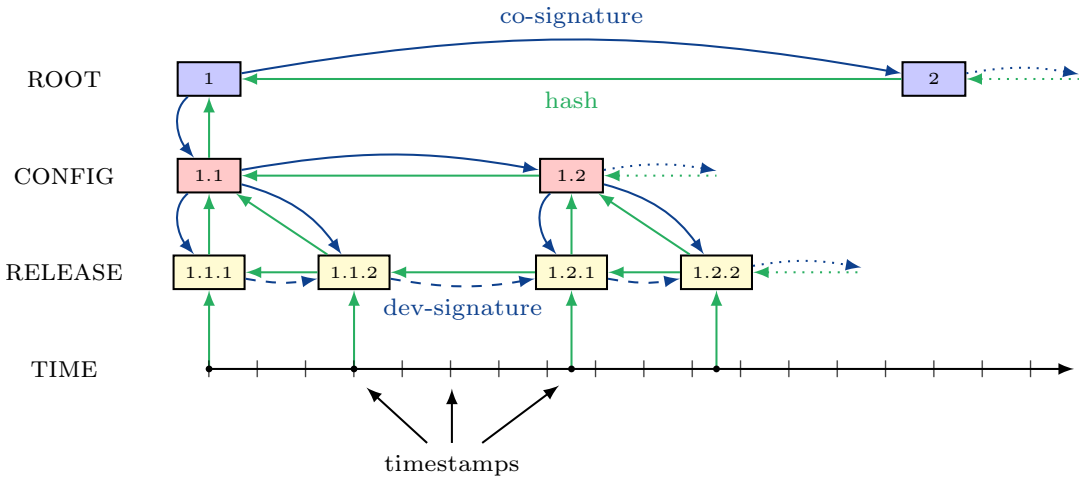

Figure 4.3: Trust delegation in CHAINIAC. The TIME level indicates the latest release at a given timestamp. The RELEASE level records developer-signed release information. The CONFIG level defines the online cothority keys that are used for validating releases. The ROOT level records the offline signing keys that authorize CONFIG keys. Inter-level links indicate inclusion of hashes and signature. For example, the blue arrow from CONFIG 1.1 to RELEASE 1.1.1 indicates that the release data in the block 1.1.1 has been co-signed using the cothority keys from the block 1.1. The green arrow indicates that a hash of CONFIG 1.1 is included in the RELEASE 1.1.1.

Finally, the TIME role provides a timestamp service that informs clients of the latest version of a package, within a coarse-grained time interval. Every TIME block contains a wall-clock timestamp and a hash of the latest release. The CONFIG leader creates this block when a new RELEASE skipblock is co-signed, or every hour if nothing happens. Before signing it off, the rest of the independent servers check that the hash inside the timestamp is correct and that the time indicated is sufficiently close to their clocks (e.g., within five minutes). From an absence of fresh TIME updates and provided that clients has an approximately accurate notion of the current time[1], the clients can then detect freeze attacks.

4.4.7 Multiple-Package Projects

To keep track of software packages, users often rely on large software projects, such as Debian or Ubuntu, and their community repositories. Each of these packages can be maintained by a separate group of developers, hence might deploy its own release log. To stay updated with new releases of installed packages, a user would have to frequently contact all the respective release logs and follow their configuration skipchains. This is not only bandwidth- and time-consuming for the user but also requires the maintainers of each package to run a freshness

[1]Protecting the client's notion of time is an important but orthogonal problem [127], solvable using a timestamping service with collectively-signed proofs-of-freshness, as in CoSi [174, Section V.A.].

service. To alleviate this burden, we further enhance CHAINIAC to support multi-package projects.

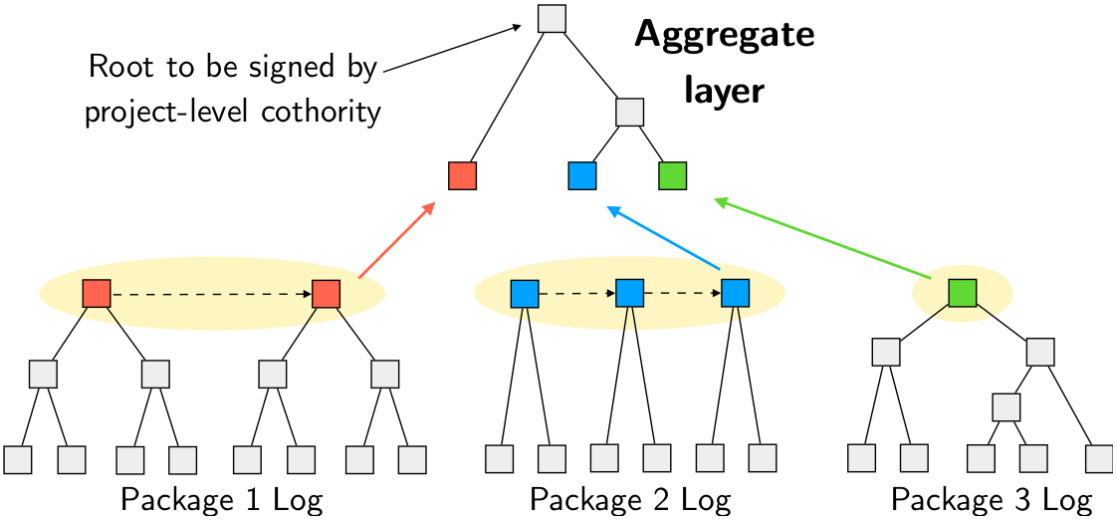

Figure 4.4: Constructing an aggregate layer in CHAINIAC. The latest data from individual-project skipchains are included in each aggregate-level snapshot.

We introduce an *aggregate* layer into CHAINIAC: this layer is responsible for collecting, validating and providing to clients information about all the packages included in the project. A project-level update cothority implements a project log where each entry is a *snapshot* of a project state (Figure 4.4). To publish a new snapshot, the cothority retrieves the latest data from the individual package skipchains, including freshness proofs and signatures on the heads. The witnesses then verify the correctness and freshness of all packages in this snapshot against the corresponding per-package logs. Finally, the cothority forms a Merkle tree that summarizes all package versions in the snapshot, then collectively signs it.

This architecture facilitates the gradual upgrade of large open-source projects, as packages that do not yet have their own skipchains can still be included in the aggregate layer as hash values of the latest release files. The project-level cothority runs an aggregate timestamp service, ensuring that clients are provided with the latest status of all individual packages and a consistent repository state. A client can request the latest signed project-snapshot from the update cothority and check outdated packages on her system using Merkle proofs. If there are such packages, the client accesses their individual release logs, knowing the hash values of the latest blocks.

A multi-package project can potentially have several aggregate layers, each representing a certain distribution , e.g., based on the development phase of packages, such as *stable*, *testing*, and *unstable* in Debian. Individual packages would still maintain a single-view linear skipchain-log but the project developers would additionally tag each release with its distribution affiliation. For example, the stable distribution would then notify clients only when correspondingly tagged releases appear, and would point to the precise block in the package

skipchain by providing its hash value, whereas the developers might move ahead and publish experimental versions of the package to its release log. The timeliness is then ensured by maintaining a separate timestamp service for each distribution.

4.5 Security Analysis

In this section, we informally analyze the security of CHAINIAC against the threat model defined in Section 4.3.2. We thereby assume that an adversary is computationally bound and unable to compromise the employed cryptosystems (e.g., create hash collisions or forge signatures), except with negligible probability.

Developers. The first point of attack in CHAINIAC is the software-release proposal created by developers. An attacker might try to sneak a vulnerability into the source code, to compromise the developers' signing keys, or to intercept a release proposal that the developers send to the update cothority, and to replace it with a backdoored version. If developers carefully review source-code changes and releases, and fewer than the threshold t_d of developers or their keys are compromised, the attacker alone cannot forge a release proposal that the update cothority would accept.[2] As developer-signed release proposals are cryptographically bound to particular sources and binaries, the update cothority will similarly refuse to sign a release proposal whose sources differ from the signed versions, or whose binaries differ from those reproduced by the build verifiers. If a sub-threshold number of developer keys are compromised without detection, a regular signing key rotation (Section 4.4.5) can eventually re-establish full security of the developer keys.

Update cothority. The next point an adversary might attack is the update-cothority's witness servers. The witnesses and build verifiers should be chosen carefully by the software project or repository maintainers, should reside in different physical locations, and be controlled by trustworthy, independent parties. For a successful attack, the adversary must compromise at least t_w witnesses to violate the correctness or transparency of the release timeline, and must compromise t_v build verifiers to break the source-to-binary release correspondence. As with developer keys, the regular rotation of cothority keys further impedes a gradual compromise.

If a threshold of online cothority keys are compromised, then, once this compromise is detected, the developers can use the offline ROOT keys to establish a new cothority configuration (see Section 4.4.6). Non-compromised clients (e.g., those that did not update critical software during the period of compromise) can then "roll forward" securely to the new configuration. An unavoidable limitation of this (or any) recovery mechanism using offline keys, however, is an inability to ensure timeliness of configuration changes. Old clients, whose network

[2] Of course there is no guarantee that even honest, competent developers will detect all bugs, let alone sophisticated backdoors masquerading as bugs. CHAINIAC's transparency provisions ensure that even compromised releases are logged and open to scrutiny, and the freshness mechanisms ensure that a compromised release does not remain usable in rollback or freeze attacks after being fixed and superseded.

connectivity is attacker controlled, could be denied the knowledge of the new configuration, hence remain reliant on the old, compromised cothority configuration. “Fixing” this weakness would require bringing the offline ROOT keys online, defeating their purpose.

Update timeline. An attacker might attempt to tamper with the skipchain-based update timeline containing the authoritative signing keys and the software releases, e.g., by attempting to fork either of the logs, to modify entries, or to present different views to users. The skipchain structure relies on the security of the underlying hash and digital signature schemes. Backward links are hashes ensuring the immutability of the past with respect to any valid release. An attacker can propose a release record with incorrect back-links, but cannot produce a valid collective signature on such a record without compromising a threshold of witnesses, as honest witnesses verify the consistency of new records against their view of history before cosigning. An attacker can attempt to create two distinct successors to the same prior release (a fork), but any honest witness will cosign at most one of these branches. If the cothority is configured with a two-thirds supermajority witness-threshold ($t_w \geq 2n_w + 1$), forks are prevented by the BFT-CoSi consensus mechanism.

Forward links are signatures that can be created only once the (future) target blocks have been appended to the skipchain. This requires that witnesses store the signing keys associated with a given block, until all forward links from that block onwards are generated. This longer key-storage, gives the attacker more time to compromise a threshold of keys. To mitigate this threat, we impose an expiration date on signing keys (e.g., one year), after which honest witnesses delete outdated keys unconditionally, thereby imposing an effective distance limit on forward links. Note that the key expiration-time should be sufficiently long so that the direct forward links are always created to ensure secure trust delegation.

In summary, to manipulate the update timeline managed by the update cothority, an attacker needs to compromise at least a threshold of t_w witness servers. Note that one purpose of the update timeline in CHAINIAC is to ensure accountability so that even if the attacker manages to slip a backdoor into a release, the corresponding source code stays irrevocably available, enabling public scrutiny.

Update center. An adversary might also compromise the software-update center to disseminate malicious binaries, to mount freeze attacks that prevent clients from updating, or to replay old packages with known security vulnerabilities and force clients to downgrade.

Clients can detect that they have received a tampered binary by verifying the associated signature using the public key of the update cothority; the key can be retrieved securely through CHAINIAC’s update timeline. The clients will also never downgrade, as they only install packages that are cryptographically linked to the currently installed version through the release skipchain. Finally, assuming the clients have a correct internal clock, they can detect freeze and replay attacks by verifying timestamps and package signatures, because an

attacker cannot forge collective signatures of the update cothority to create valid-looking TIME blocks (see Section 4.4.6).

4.6 Experimental Evaluation

In this section, we experimentally evaluate our CHAINIAC prototype. The main question we answer is whether CHAINIAC is usable in practice without incurring large overheads. We begin by measuring the cost of reproducible builds using Debian packages as an example, and we continue with the cost of witnesses who maintain an update-timeline skipchain and the overhead of securing multi-package projects.

4.6.1 Implementation

We implemented CHAINIAC in Go and made it publicly available[3], along with the instructions on how to reproduce the evaluation experiments. We built on existing open-source code implementing CoSi [174] and BFT-CoSi [111]. The new code implementing the CHAINIAC prototype was about 1.8kLOC, whereas skipchains, network communication, and BFT-CoSi were 1.2k, 1.5k, and 1.8k lines of code (LOC), respectively. Although the implementation is not yet production quality, it is practical and usable for experimental purposes.

We rely on Git for source-code control and use Git-notes [84], tweaked with server hooks to be append-only, for collecting developer approvals in the form of PGP signatures. For the build verifiers, we use Python to extract the information about the building environment of the packages, and Docker [69] to reproduce it.

4.6.2 Experimental Methodology

In the experiments of Section 4.6.3 and Section 4.6.4, we used 24-core Intel Xeons at 2.5 GHz with 256 GB of RAM and, where applicable, ran up to 128 nodes on one server with the network-delay set between any two nodes to 100 ms with the help of Mininet [134]. Because we had not yet implemented a graceful handling of failing docker-builds, we measured building time in a small grid of 4 nodes and extrapolated this time to the bigger grids in Figure 4.6. In Section 4.6.5, we simulated four collectively signing servers on a computer with a 3.1 GHz Intel Core i7 processor and 16 GB of RAM and did not include any network-latencies, as we measured only CPU-time and bandwidth.

To evaluate the witness cost of the long-term maintenance of an update timeline, we used data from the Debian reproducible builds project [59] and the Debian snapshot archive [63]. The former provides checksums and dependency information for reproducible packages. Unfortunately, the information was not available for older package versions, therefore we

[3] https://github.com/dedis/paper_chainiac

always verified each package against its newest version. We used the latter as an update history to estimate average cost over time for maintaining an individual update timeline and the overhead of running an aggregate multi-package service.

4.6.3 Reproducing Debian Packages

To explore the feasibility of build transparency and to estimate the cost of it for witnesses, we ran an experiment on automatic build reproducing. Using Docker containers, we generated a reproducible build environment for each package, measured the CPU time required to build a binary and verified the obtained hash against a pre-calculated hash from Debian.

We tested three sets of packages: (1) *required* is the set of 27 Debian required packages [62]; (2) *popular* contains the 50 most installed Debian packages [60] that are reproducible and do not appear in *required*; (3) *random* is a set of 50 packages randomly chosen from the full reproducible testing set [59]. Figure 4.5 demonstrates a CDF of the build time for each set.

10 packages from the random set, 8 from required and 2 from popular produced a hash value different from the corresponding advertised hash. 90% of packages from both the random and required sets were built in less than three minutes, whereas the packages in the required-set have a higher deviation. This is expected as, to ensure Debian's correct functioning, the required packages tend to be more security critical and complex.

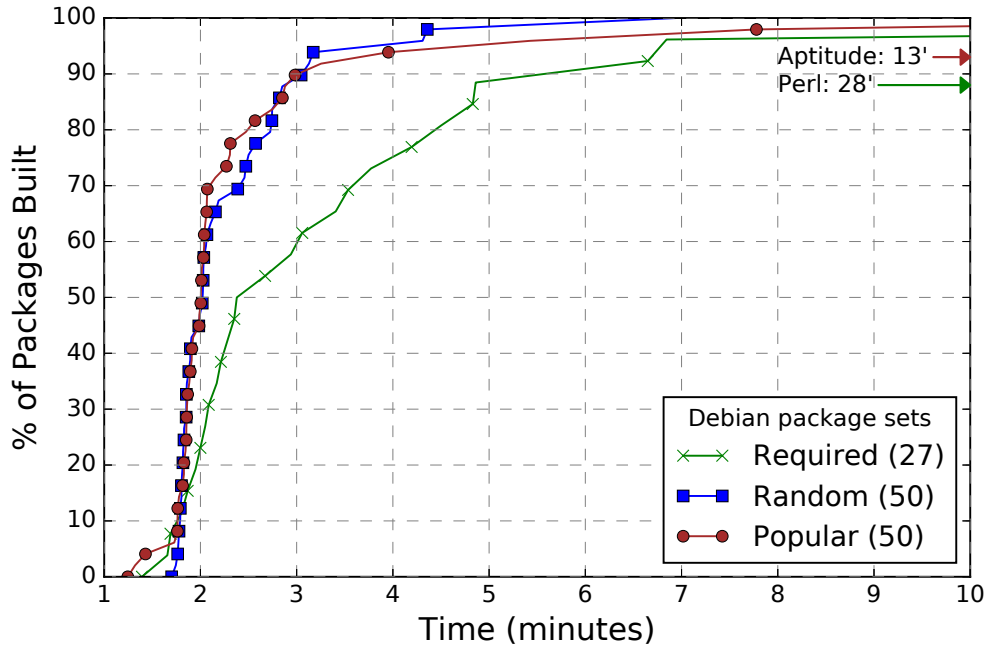

Figure 4.5: Reproducible build latency for Debian packages.

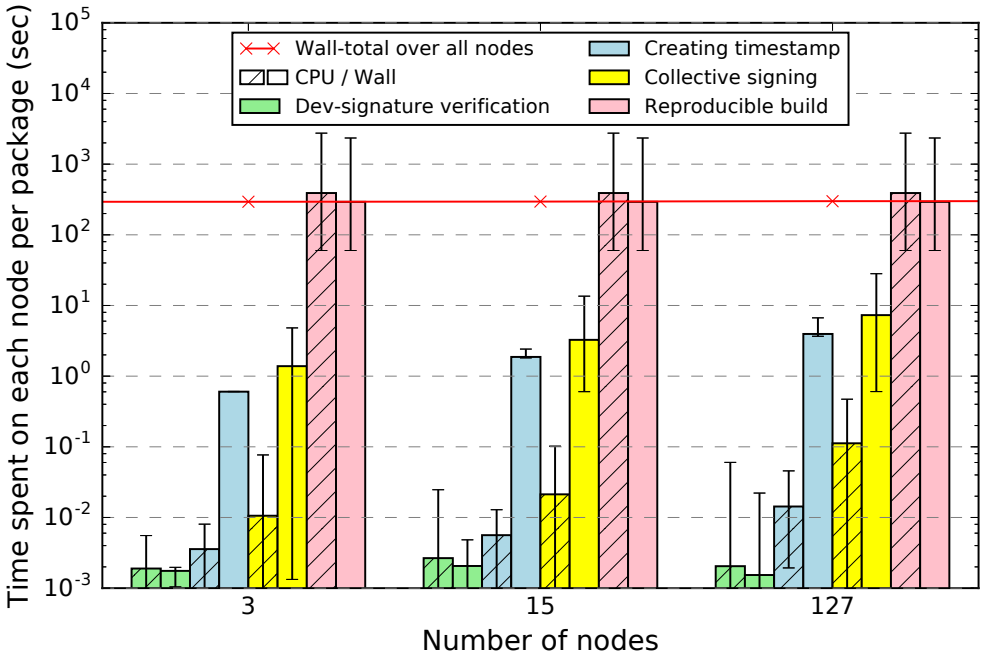

Figure 4.6: CPU cost of adding a new block to a timeline.

4.6.4 End-to-End Witness Cost

In this experiment, we measured the cost for a witness of adding a new release to an update timeline. We took a set of six packages, measured the cost for each one individually and then calculated the average values over all the packages. The build time was measured once and copied to the other runs of the experiment, which enabled us to test different configurations quickly and to break out results for each operation. The operations included verifying developers' signatures, reproducible builds, signing off on the new release and generating a timestamp. The witness cost was measured for an update cothorities composed of 7, 31, and 127 nodes.

Figure 4.6 plots the costs in both CPU time and wall-clock time used. The CPU time is higher than wall-clock time for some metrics, due to the use of a multi-core processor. The verification and build times are constant per node, whereas the time to sign and to generate the timestamp increases with the number of nodes, mostly due to higher communication latency in a larger cothority tree. As expected, the build time dominates the creation of a new skipblock. Every witness spends between 5 and 30 CPU-minutes for each package. Current hosting schemes offer simple servers for 10-US$ per month, enough to run a node doing reproducible builds for the Debian-security repository (about eight packages per day).

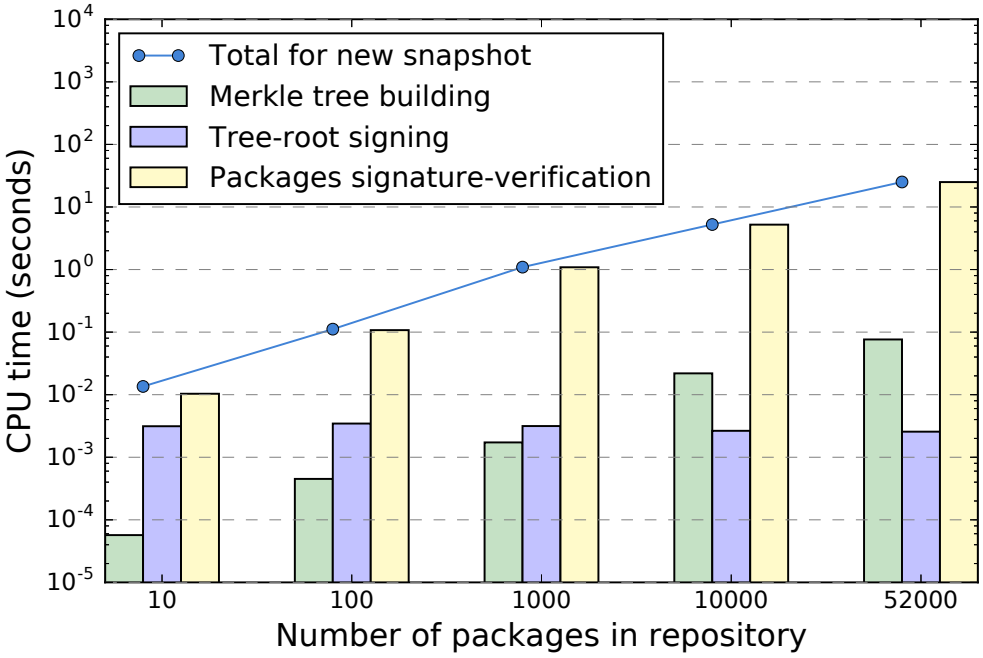

Figure 4.7: CPU time on server for repository-update.

4.6.5 Cost of Securing Debian Distribution

In our final experiment, we measured the cost of a witness server that deploys an aggregate-layer skipchain in a multi-package project (Section 4.4.7) and a client who uses it. We took the list of all the packages from the snapshot archive of the Debian-testing repository and created one skipchain per package over 1.5-year history, such that each skipblock is one snapshot every five days. We then formed the aggregate Debian-testing skipchain over the same period.

In the first experiment, a witness server receives a new repository-state to validate, verifies the signature for all the packages, builds a Merkle tree from the heads of the individual skipchains and signs its root, thus creating a new aggregate skipblock. Figure 4.7 depicts the average costs of the operations, over the whole history, against the size of the repository. For a full repository of 52k packages, which corresponds to the actual Debian-testing system, the overall CPU-cost is about 20 seconds per release. This signifies that Chainiac generates negligible overhead on the servers that update a skipchain.

The second experiment evaluates the overhead that Chainiac introduces to the client-side cost of downloading the latest update of all packages. In order to maintain the security guarantees of Chainiac, the client downloads all package hashes and builds a full Merkle tree to verify them, thereby not revealing packages of interest and preserving her privacy. Figure 4.8 illustrates that Chainiac introduces a constant overhead of 16% to the APT manager. This modest overhead suggests Chainiac's good scalability and applicability.

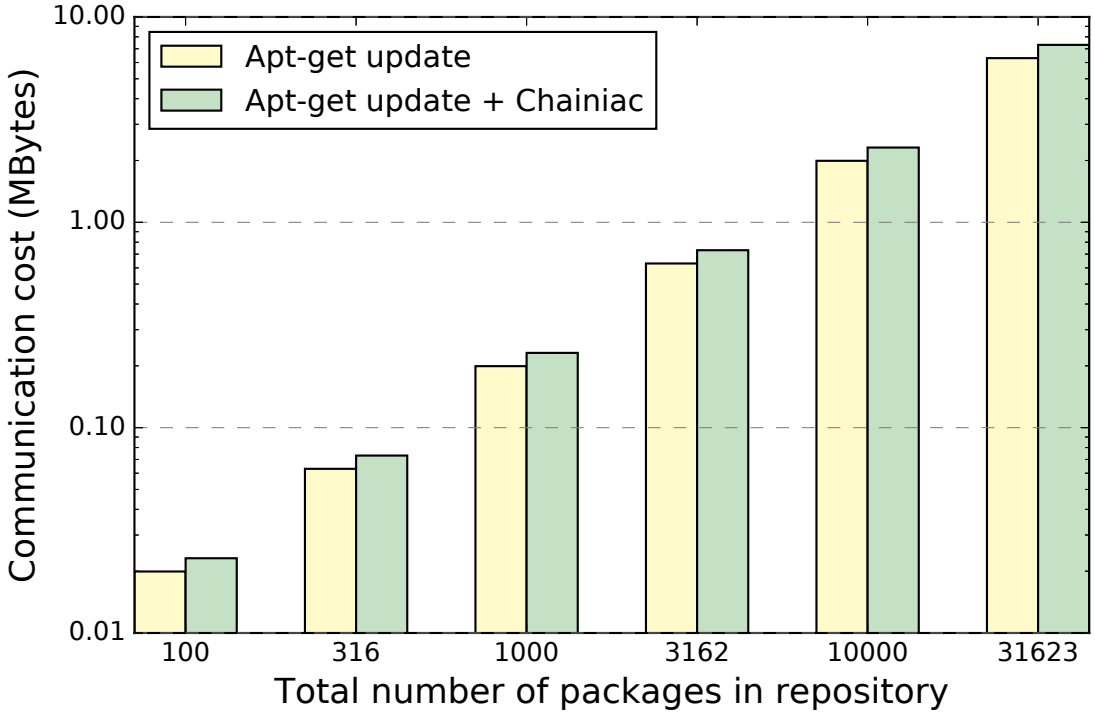

Figure 4.8: Communication cost to get new repository state.

4.7 Related Work

We organize the discussion topically and avoid redundancy with the commentary in Section 4.2.

Software-update protection. The automatic detection and installation of software updates is a common operation in computer and mobile systems, and there are many tools for this task, such as package- and library-managers [61], [73], [94], [194], and various app stores. There are several security studies [26], [44], [143] that reveal weaknesses in the design of software-update systems, and different solutions are proposed to address these weaknesses. Solutions that reduce the trust that end users must have in developers by involving independent intermediaries in testing [4], [5] are shown to be beneficial in open-source projects and content repositories. Several systems, such as Meteor [14], DroidRanger [198] and ThinAV [99], focus on protecting the infrastructure for mobile applications and on detecting malware in mobile markets. Other systems [101], [121], [146] use overlay and peer-to-peer networks for efficient dissemination of security patches, whereas Updaticator [7] enables efficient update distribution over untrusted cache-enabled networks.

Certificate, key, and software transparency. Bringing transparency to different security-critical domains has been actively studied. Solutions for public-key validation infrastructure are proposed in AKI [107], ARPKI [17] and Certificate Transparency (CT) [118] in which all issued

public-key certificates are publicly logged and validated by auditors. Public logs are also used in Keybase [105], which enables users to manage their online accounts and provides checking of name-to-key bindings by verifying ownership of third-party accounts. This is achieved via creating a public log of identity information that third-parties can audit. EthIKS [35] provides stronger auditability to CONIKS [130], an end-user key verification service based on a verifiable transparency log, by creating a Smart Ethereum Contract [190] that guarantees that a hash chain is not forked, as long as the ethereum system is stable and correct. Application Transparency (AT) [74] employs the idea of submitting information about mobile applications to a verifiable public log. Thus, users can verify that a provided app is publicly available to everyone or that a given version existed in the market, but was removed. However, AT can protect only against targeted attacks but leaves attacks against all the users outside of its scope. Finally, Baton [15] addresses the problem of renewing signing keys in Android by chaining them but this solution does not help in the case of stolen signing keys.

Blockchains. The creation of Bitcoin [140] was first perceived as an evolution in the domain of financial technology. Recently, however, there has been an increasing interest in the data structure that enables the properties of bitcoin, namely, the blockchain. There is active work with blockchain in cryptocurrencies [36], [168], DNS alternatives [71] and even general-purpose decentralized computing [190]. All of these systems secure clients in a distributed manner and with a timeline that can be tracked by the clients. However, these systems force the clients to track the full timeline, even if the clients are interested in a very small subset of it, or to forfeit the security of decentralization by trusting a full node.

4.8 Conclusion

We have presented Chainiac, a novel software-update framework that decentralizes each step of the software-update process to increase trustworthiness and to eliminate single points of failure. The distinct layers of skipchains in Chainiac provide (1) tamper-evident and equivocation-resistant logging of the new updates and (2) the secure evolution of signing keys for both developers and the set of online witnesses, while limiting overhead for the client. Verified builds further unburden clients by delegating the actual reproducible building process to a decentralized set of build verifiers. The evaluation of our prototype on real-world data from Debian has demonstrated that Chainiac is practical, both for the clients and for the decentralized group of witnesses.

5 Conclusion

Protecting a complex process, such as data retrieval, requires comprehensive solutions. As we have shown with the examples from prior work, neglecting seemingly irrelevant security properties can lead to subtle vulnerabilities and design flaws in deployed systems. On the bright side, we have also demonstrated that it is possible to design and build mechanisms that provide broad protection and provable security guarantees. An encryption scheme that protects both content and metadata, a private-information-retrieval scheme that atomically ensures user privacy and data integrity, and a software-release architecture that enhances transparency and verifiability of the update process are examples of the comprehensive approach we have presented in this dissertation. We believe that hybrid mechanisms that provide multiple security properties in an atomic way is the right direction forward, and we will see more practical solutions that follow it.

5.1 Future Work

This dissertation leaves room for interesting future work:

Communication Protocols with Metadata Protection. The protocols for secure communication, such as TLS [159], tend to leak information via encryption metadata, record length, and inter-packet timings. Although PURBs are a potential solution to limiting the exposure of encryption metadata and to hiding the true data length (e.g., using the Padmé padding function [141]), the questions about how to apply these techniques in the context of a dynamic protocol and, furthermore, how to protect information revealed via traffic patterns remain to be explored. A possible direction is to define a set of techniques that any protocol for secure communication can directly apply to achieve quantifiable guarantees of metadata privacy and to demonstrate how these techniques work on a concrete real-world example. We have completed a preliminary study of metadata protection in TLS and can confirm that this direction is promising.

More Efficient Verifiable Single-Server PIR Schemes. The state-of-the-art classic single-server PIR schemes achieve polylogarithmic communication costs and are able to efficiently operate over large database blocks, whereas our verifiable PIR scheme achieves only $O(\sqrt{n})$ communication for database size n and operates only with single bits. Therefore, two immediate open questions follow. Are there practical single-server verifiable-PIR schemes (e.g., based on discrete log) with $\mathsf{polylog}(n)$ communication? And do there exist single-server verifiable-PIR schemes that are competitive with state-of-the-art lattice-based PIR [8]?

Transparency and Verifiability for the Web. CHAINIAC uses decentralization and collective validation for ensuring the security of software updates. Our techniques from CHAINIAC, however, are not limited to software-release processes. One immediate follow-up is to translate these techniques to other applications where source code is distributed to users, e.g., in browser extensions or Web pages. Moreover, it is possible to imagine using collective validation, transparency and verifiability in completely different contexts, such as content moderation on social media. The direction of increasing transparency on the Internet is, in general, worth pursuing.

A Security Analysis

A.1 Analysis of MsPURB

This section contains the proofs of the security properties provided by MsPURB.

A.1.1 Preliminaries

Before diving into proving the security of our scheme, we define what it means to be IND-CCA2- and IND$-CCA2-secure for the primitives that MsPURB builds upon.

Key-Encapsulation Mechanism (KEM). Following the definition from Katz & Lindell [104], we begin by defining KEM as a tuple of PPT algorithms.

SYNTAX KEM.
$\mathsf{KEM.Setup}(1^\lambda) \to S$: Given a security parameter λ, initialize a cipher suite S.

$\mathsf{KEM.KeyGen}(S) \to (sk, pk)$: Given a cipher suite S, generate a (private, public) key pair.

$\mathsf{KEM.Encap}(pk) \to (c, k)$: Given a public key pk, output a ciphertext c and a key k.

$\mathsf{KEM.Decap}(sk, c) \to k/\perp$: Given a private key sk and a ciphertext c, output a key k or $\perp$ in case of failure.

Consider an IND-CCA2 security game against an adaptive adversary $\mathscr{A}$:

GAME KEM.
The KEM IND-CCA2 game for a security parameter λ is between a challenger and an adaptive adversary $\mathscr{A}$. It proceeds along the following phases.

Init: The challenger and adversary take λ as input. The adversary outputs a cipher suite S it wants to attack. The challenger verifies that S is a valid cipher suite, i.e., that it a valid output of $\mathsf{KEM.Setup}(1^\lambda)$. The challenger aborts, and sets $b^\star \xleftarrow{\$} \{0,1\}$ if S is not valid.

Setup: The challenger runs $(sk, pk) \leftarrow \mathsf{KEM.KeyGen}(S)$ and gives pk to $\mathscr{A}$.

Phase 1: $\mathscr{A}$ can make decapsulation queries $\mathsf{qDecap}(c)$ with ciphertexts c of its choice, to the challenger who responds with $\mathsf{KEM.Decap}(sk, c)$.

Challenge: The challenger runs $(c^\star, k_0) \leftarrow \mathsf{KEM.Encap}(pk)$ and generates $k_1 \xleftarrow{\$} \{0,1\}^{|k_0|}$. The challenger picks $b \xleftarrow{\$} \{0,1\}$ and sends $\langle c^\star, k_b \rangle$ to $\mathscr{A}$.

Phase 2: $\mathscr{A}$ continues querying $\mathsf{qDecap}(c)$ with the restriction that $c \neq c^\star$.

Guess: $\mathscr{A}$ outputs its guess $b^\star$ for b and wins if $b^\star = b$.

We define $\mathscr{A}$'s advantage in this game as:

$$\mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathscr{A}}(1^\lambda) = 2\left|\Pr[b = b^\star] - \tfrac{1}{2}\right|.$$

We say that a KEM is IND-CCA2-secure if $\mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathscr{A}}(1^\lambda)$ is negligible in the security parameter.

Definition 5. *We say that a KEM is* perfectly correct *if for all* $(sk, pk) \leftarrow \mathsf{KEM.KeyGen}(S)$ *and for all* $(c, k) \leftarrow \mathsf{KEM.Encap}(pk)$ *we have* $k = \mathsf{KEM.Decap}(sk, c)$.

Instantiation IES-KEM.
We instantiate a KEM based on the Integrated Encryption Scheme [1].

$\mathsf{IES.Setup}(1^\lambda)$: Initialize a cipher suite $S = \langle \mathbb{G}, p, g, \mathrm{H} \rangle$, where $\mathbb{G}$ is a cyclic group of order p and generated by g, and $\mathrm{H} : \mathbb{G} \to \{0,1\}^{2\lambda}$ is a hash function.

$\mathsf{IES.KeyGen}(S)$: Pick $x \in \mathbb{Z}_p$, compute $X = g^x$, and output $(sk = x, pk = X)$.

$\mathsf{IES.Encap}(pk)$: Given $pk = Y$, pick $x \in \mathbb{Z}_p$, compute $X = g^x$, and output $\langle c = X, k = \mathrm{H}(Y^x) \rangle$.

$\mathsf{IES.Decap}(sk, c)$: Given $sk = y$ and $c = X$, output a key $k = \mathrm{H}(X^y)$.

Theorem 6 (Theorem 11.22 [104], Section 7 [1])**.** *If the gap-CDH problem is hard relative to* $\mathbb{G}$*, and* H *is modeled as a random oracle, then IES-KEM is an* IND-CCA2*-secure KEM.*

Multi-Suite Broadcast Encryption. We consider MsPURB as a multi-suite broadcast encryption (MSBE) scheme extending the single-suite setting by Barth et al. [16].

Syntax MSBE.
$\mathsf{MSBE.Setup}(1^\lambda) \to S$: Given a security parameter λ, initialize a cipher suite S.

$\mathsf{MSBE.KeyGen}(S) \to (sk, pk)$: Given a cipher suite S, generate a (private, public) key pair.

$\mathsf{MSBE.Enc}(R, m) \to c$: Given a set of public keys $R = \{pk_1, \ldots, pk_r\}$ with corresponding cipher suites $S_1, \ldots, S_r$ and a message m, generate a ciphertext c.

$\mathsf{MSBE.Dec}(sk, c) \rightarrow m/\perp$: Given a private key sk and the ciphertext c, return a message m or $\perp$ if c does not decrypt correctly.

Note that MsPURB as described in §2.3.7 satisfies the syntax of a multi-suite broadcast encryption scheme.

Barth et al. [16] define the security of broadcast encryption schemes under adaptive chosen-ciphertext attack for single-suite schemes. Here, we adjust this definition to the multi-suite setting, and instead require that the ciphertext is indistinguishable from a random string (IND$-CCA2).

Game MSBE.
The MSBE IND$-CCA2 game for a security parameter λ is between a challenger and an adversary $\mathscr{A}$. It proceeds along the following phases.

Init: The challenger and adversary take λ as input. The adversary outputs a number of recipients r and corresponding cipher suites $S_1, \ldots, S_r$ it wants to attack. Let s be the number of unique cipher suites. The challenger verifies, for each $i \in \{1, \ldots, r\}$, that S_i is a valid cipher suite, i.e., that it is a valid output of $\mathsf{MSBE.Setup}(1^\lambda)$. The challenger aborts, and sets $b^\star \xleftarrow{\$} \{0,1\}$ if the suites are not all valid.

Setup: The challenger generates private-public key pairs for each recipient i given by $\mathscr{A}$ by running $(sk_i, pk_i) \leftarrow \mathsf{MSBE.KeyGen}(S_i)$ and gives $R = \{pk_1, \ldots, pk_r\}$ to $\mathscr{A}$.

Phase 1: $\mathscr{A}$ can make decryption queries $\mathsf{qDec}(pk_i, c)$ to the challenger for any $pk_i \in R$ and any ciphertext c of its choice. The challenger replies with $\mathsf{MSBE.Dec}(sk_i, c)$.

Challenge: $\mathscr{A}$ outputs $m^\star$. The challenger generates $c_0 = \mathsf{MSBE.Enc}(R, m^\star)$ and $c_1 \xleftarrow{\$} \{0,1\}^{|c_0|}$. The challenger picks $b \xleftarrow{\$} \{0,1\}$ and sends $c^\star = c_b$ to $\mathscr{A}$.

Phase 2: $\mathscr{A}$ continues making decryption queries $\mathsf{qDec}(pk_i, c)$ with a restriction that $c \neq c^\star$.

Guess: $\mathscr{A}$ outputs its guess $b^\star$ for b and wins if $b^\star = b$.

We define $\mathscr{A}$'s advantage in this game as:

$$\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda) = 2\left|\Pr[b = b^\star] - \tfrac{1}{2}\right|.$$

We say that a MSBE scheme is IND$-CCA2-secure if $\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda)$ is negligible in the security parameter.

Finally, we require that the MAC scheme is strongly unforgeable under an adaptive chosen-message attack *and* outputs tags that are indistinguishable from random. A MAC scheme is given by the algorithms $\mathsf{MAC.KeyGen}, \mathscr{M}$, and $\mathscr{V}$, where $\mathsf{MAC.KeyGen}(1^\lambda)$ outputs a key K_{mac}. To compute a tag on the message m, run $\sigma = \mathscr{M}_{K_{mac}}(m)$. The verification algorithm

$\mathcal{V}_{K_{mac}}(m,\sigma)$ outputs $\top$ if σ is a valid tag on the message m and $\perp$ otherwise. We formalize the strong unforgeability and indistinguishability properties using the following simple games.

GAME MAC-SFORGE.
The MAC-sforge game for a security parameter λ is between a challenger and an adversary $\mathcal{A}$.

Setup: The challenger and adversary take λ as input. The challenger generates a MAC key $K_{mac} \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$.

Challenge: The adversary $\mathcal{A}$ is given oracle access to the oracles $\mathcal{M}(\cdot)$ and $\mathcal{V}(\cdot)$. On a query $\mathcal{M}(m)$ the challenger returns $\sigma = \mathcal{M}_{K_{mac}}(m)$. On a query $\mathcal{V}(m,\sigma)$ the challenger returns $\mathcal{V}_{K_{mac}}(m,\sigma)$.

Output: $\mathcal{A}$ eventually outputs a message-tag pair (m,σ). $\mathcal{A}$ wins if $\mathcal{V}_{K_{mac}}(m,\sigma) = 1$ *and* $\mathcal{A}$ has not made a query $\mathcal{M}(m)$ that returned σ.

We define $\mathcal{A}$'s advantage in this game as:

$$\mathsf{Adv}^{\mathsf{suf}}_{\mathsf{MAC},\mathcal{A}}(1^\lambda) = \Pr[\mathcal{A} \text{ wins}].$$

We say that a MAC scheme is strongly unforgeable under adaptive chosen-message attacks if $\mathsf{Adv}^{\mathsf{suf}}_{\mathsf{MAC},\mathcal{A}}(1^\lambda)$ is negligible in the security parameter.

GAME MAC-IND$.
The MAC-IND$ game is between a challenger and an adversary $\mathcal{A}$.

Setup: The challenger and adversary take λ as input. The challenger generates a MAC key $K_{mac} \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$ and picks a bit $b \xleftarrow{\$} \{0,1\}$.

Challenge: The adversary outputs a message m. The challenger computes $\sigma_0 = \mathcal{M}_{K_{mac}}(m)$ and $\sigma_1 \xleftarrow{\$} \{0,1\}^{|\sigma_0|}$ and returns σ_b.

Output: The adversary outputs its guess $b^\star$ of b, and wins if $b^\star = b$.

We define $\mathcal{A}$'s advantage in this game as:

$$\mathsf{Adv}^{\mathsf{ind\$}}_{\mathsf{MAC},\mathcal{A}}(1^\lambda) = 2\left|\Pr[b = b^\star] - \tfrac{1}{2}\right|.$$

We say that the tags of a MAC scheme are indistinguishable from random if $\mathsf{Adv}^{\mathsf{ind\$}}_{\mathsf{MAC},\mathcal{A}}(1^\lambda)$ is negligible in the security parameter.

A.1.2 Proof of Theorem 1

We prove the IND$-CCA2 security of MsPURB as an MSBE scheme. More precisely, we will show that there exists adversaries $\mathbb{B}_1, \ldots, \mathbb{B}_5$ such that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda) \leq r\left(\mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathbb{B}_1}(1^\lambda) + \mathsf{Adv}^{\mathsf{ind\$\text{-}cca2}}_{\Pi,\mathbb{B}_2}(1^\lambda)\right) + \\ \mathsf{Adv}^{\mathsf{suf}}_{\mathsf{MAC},\mathbb{B}_3}(1^\lambda) + \mathsf{Adv}^{\mathsf{ind\$}}_{\mathsf{MAC},\mathbb{B}_4}(1^\lambda) + \\ \mathsf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{(\mathsf{Enc},\mathsf{Dec}),\mathbb{B}_5}(1^\lambda).\end{aligned}$$

Thus, given our assumptions, $\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda)$ is indeed negligible in λ. To do so we use a sequence of games. This sequence of games step by step transforms from the situation where $b = 0$ in the IND$-CCA2 game of MSBE, i.e., the adversary receives the real ciphertext, to $b = 1$, i.e., the adversary receives a random string.

Game G_0.
This game is as the original MSBE IND$-CCA2 game where $b = 0$.

Game G_1.
As in G_0, but the challenger will no longer call HdrPURB.Decap to derive the keys k_i on ciphertexts derived from the challenge ciphertext $c^\star$. In particular, for every recipient pk_i using a suite S_j, we store $(X_j^\star, k_i^\star)$ when constructing the PURB headers for the challenge ciphertext. Then, when receiving a decryption query for a recipient $\mathsf{qDec}(pk_i(S_j), c)$, we proceed by following MsPURB.Dec. If the encoded public key τ recovered in step (1) of MsPURB.Dec is such that $\mathtt{Unhide}(\tau) = X_j^\star$, then we use $k_i = k_i^\star$ (as stored when creating the challenge ciphertext) directly, rather than computing $k_i = \mathsf{HdrPURB.Decap}(y_i, \tau)$ in step (2) of MsPURB.Dec. If the encoded public key τ does not match $X_j^\star$, then the challenger proceeds as before.

Game G_2.
As in G_1, but we change how the keys $k_1^\star, \ldots, k_r^\star$ for the *challenge ciphertext* are computed in HdrPURB.Encap. Rather than computing $k_i^\star = \mathrm{H}(Y_i^x)$ as in step (2) of HdrPURB.Encap, we set $k_i^\star \xleftarrow{\$} \{0,1\}^{\lambda_H}$ for all the keys, where λ_H is the bit-length of the corresponding hash function H. Recall that as per the changes in G_1, the challenger will store $k_i^\star$ generated in this way, and use them directly (without calling HdrPURB.Decap) when asked to decrypt variants of the challenge ciphertext.

Game G_3.
Let e_i be the encrypted entry point under key Z_i (derived from k_i) for recipient i computed in line 47 of Layout (step (7) of MsPURB.Enc). The game goes as in G_2, but for the challenge ciphertext, the challenger saves the mapping of the challenge entry points and the encapsulated key $K^\star$ with metadata $\mathsf{meta}^\star$: $(e_i^\star, k_i^\star, K^\star \parallel \mathsf{meta}^\star)$. If the challenger receives a decryption query $\mathsf{qDec}(pk_i(S_i), c)$ it proceeds as before, except when it should decrypt $e_i^\star$ using key $k_i^\star$ in step (4) of MsPURB.Dec. In that case, it acts as if the decryption returned $K^\star \parallel \mathsf{meta}^\star$.

GAME G_4.
As in G_3, but the challenger replaces $e_1^\star, \ldots, e_r^\star$ in the challenge ciphertext with random strings of the appropriate length. Note that per the change in G_3, the challenger will not try to decrypt these $e_i^\star$, but will recover $K^\star$ and $\mathsf{meta}^\star$ directly instead.

GAME G_5.
As in G_4, but the challenger replies differently to the queries $\mathsf{qDec}(pk_i(S_i), c)$ where c is not equal the challenge ciphertext $c^\star$ but the encoded public key τ recovered in step (1) of MsPURB.Dec is such that $\mathtt{Unhide}(\tau) = X_j^\star$ *and* $e_i = e_i^\star$. In this case, the challenger replies with $\perp$ directly, without running $\mathcal{V}_{K_{mac}}(\cdot)$ (step (5) of MsPURB.Dec).

GAME G_6.
As in G_5, but the challenger replaces the integrity tag in the challenge ciphertext in step (9) of MsPURB.Enc with a random string of the same length.

GAME G_7.
As in G_6, but the challenger replaces the encrypted payload c_{payload} in the challenge ciphertext in step (6) of MsPURB.Enc with a random string of the same length.

Conclusion. As of G_7, all ciphertexts in the PURBs header, the payload encryption and the MAC have been replaced by random strings. The open slots in the hash tables are always filled with random bits. Finally, the encoded keys $\tau = \mathtt{Hide}(X)$ are indistinguishable from random strings as well, since the keys X are random. Therefore, the PURB ciphertexts c are indeed indistinguishable from random strings, as in the MSBE game with $b = 1$.

Proof. Let W_i be the event that $\mathscr{A}$ outputs $b^\star = 1$ in game G_i. We aim to show that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda) &= \left|\Pr[b^\star = 1 \mid b = 0] - \Pr[b^\star = 1 \mid b = 1]\right| \\ &= \left|\Pr[W_0] - \Pr[W_7]\right|\end{aligned}$$

is negligible. To do so, we show that each of the steps in the sequence of games is negligible, i.e., that $\left|\Pr[W_i] - \Pr[W_{i+1}]\right|$ is negligible. The result then follows from the triangle inequality.

G_0 <–> G_1.
As long as the KEMs are perfectly correct, the games G_0 and G_1 are identical. Therefore:

$$\left|\Pr[W_0] - \Pr[W_1]\right| = 0.$$

G_1 <–> G_2.
We show that the games G_1 and G_2 are indistinguishable using a hybrid argument on the number of recipients r. Consider the hybrid games H_i where the first i recipients use random keys $k_1, \ldots, k_i$ as in G_2, whereas the remaining $r - i$ recipients use the real keys $k_{i+1}, \ldots, k_r$ as in G_1. Then $G_1 = H_0$ and $G_2 = H_r$.

We prove that $\mathscr{A}$ cannot distinguish H_{j-1} from H_j. Let $S_j = \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$, be the suite corresponding to recipient j. Suppose $\mathscr{A}$ can distinguish H_{j-1} from H_j, then we can build a distinguisher $\mathbb{B}$ against the IND$-CCA2 security of the IES KEM for the suite $S'_j = \langle \mathbb{G}, p, g, \mathrm{H} \rangle$. Recall that $\mathbb{B}$ receives, from its IND$-CCA2-KEM challenger,

- a public key Y;
- a challenge $\langle X^\star, k^\star \rangle$, where depending on bit $b \xleftarrow{\$} \{0,1\}$, we have $k^\star = \mathrm{H}(Y^{x^\star})$ if $b = 0$ or $k^\star \xleftarrow{\$} \{0,1\}^{\lambda_H}$ if $b = 1$ (where λ_H is the bit-length of H);
- access to a $\mathsf{Decap}(\cdot)$ oracle for all but $X^\star$.

At the start of the game, $\mathbb{B}$ will set $pk_j = Y$, so that the public key of recipient j matches that of its IES KEM challenger. Note that $\mathbb{B}$ does not know the corresponding private key y_j. For all other recipients i, $\mathbb{B}$ sets $(sk_i = y_i, pk_i = Y_i) = \mathsf{MsPURB.KeyGen}(S_i)$.

The distinguisher $\mathbb{B}$ will use its challenge $(X^\star, k^\star)$ to construct the challenge ciphertext for $\mathscr{A}$. In particular, when running $\mathsf{HdrPURB.Encap}$ for a suite S_j, it sets $X = X^\star$ in step (1) of $\mathsf{HdrPURB.Encap}$. Moreover, for recipient j it will use $k_j = k^\star$. For all other recipients i with corresponding suites S_i it proceeds as follows when computing k_i in $\mathsf{HdrPURB.Encap}$.

- If $i < j$, then it sets $k_i \xleftarrow{\$} \{0,1\}^{\lambda_H}$ for appropriate λ_H;
- If $i > j$ and the suite S_i for user i is the same as suite S_j for user j, then it sets $k_i = \mathrm{H}(X^{\star\, y_i})$; and
- If $i > j$, but $S_j \neq S_i$, then it computes k_i as per steps (1) and (2) of $\mathsf{HdrPURB.Encap}$.

Thereafter, $\mathbb{B}$ continues running $\mathsf{MsPURB.Enc}$ as before.

Whenever $\mathbb{B}$ receives a decryption query for a user pk_i, it proceeds as before. When it receives a decryption query for user pk_j, it uses its IES-KEM Decap oracle in step (2) of $\mathsf{HdrPURB.Decap}$. Note that $\mathbb{B}$ is not allowed to call $\mathsf{Decap}(\cdot)$ on $X^\star$, but as per the changes in G_1, it will directly use $k^\star$ for user pk_j if $\mathsf{HdrPURB.Decap}$ recovers $X^\star$ in step (1).

If $b = 0$ in $\mathbb{B}$'s IES KEM challenge, then recipient j's key $k_j = \mathrm{H}(Y^{x^\star})$, and hence $\mathbb{B}$ perfectly simulates H_{j-1}. If $b = 1$ in $\mathbb{B}$'s IES KEM challenge, then j's key $k_j \xleftarrow{\$} \{0,1\}^{\lambda_H}$ and, hence, $\mathbb{B}$ perfectly simulates H_j. If $\mathscr{A}$ distinguishes H_{j-1} from H_j then $\mathbb{B}$ breaks the IND$-CCA2-KEM security of IES. Hence, H_{j-1} and H_j are indistinguishable. Repeating this argument r times shows that G_1 and G_2 are indistinguishable. More precisely:

$$\left|\Pr[W_1] - \Pr[W_2]\right| \leq r \cdot \mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathscr{A}}(1^\lambda).$$

$\underline{G_2 \text{ <-> } G_3}$.
By perfect correctness of the authentication encryption scheme, we have that for all keys k and messages m that $\mathscr{D}_k(\mathscr{E}_k(m)) = m$, thus, games G_2 and G_3 are identical. Therefore:

$$\left|\Pr[W_2] - \Pr[W_3]\right| = 0.$$

G_3 <–> G_4.
Similarly to the proof above, consider the hybrid games H_i where the first i entry points are substituted with random strings $e_1, \ldots, e_i$ as in G_4, whereas the remaining $r - i$ are the actual encryptions as in G_3. Then $G_3 = H_0$ and $G_4 = H_r$. We show that $\mathscr{A}$ cannot distinguish H_{j-1} from H_j. Let $S_j = \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$, be the suite corresponding to recipient j. We show that if $\mathscr{A}$ distinguishes H_{j-1} from H_j then we can build a distinguisher $\mathbb{B}$ against the IND$-CCA2 security of Π. $\mathbb{B}$ receives from its IND$-CCA2 challenger:

- a challenge ciphertext $e^\star$, in response to an encryption call with a message m such that, depending on the bit $b \in \{0,1\}$, we have that $e^\star = \mathscr{E}_Z(m)$ if $b = 0$ or $e^\star$ is a random string if $b = 1$;
- a decryption oracle $\mathscr{D}_Z(\cdot)$.

When constructing the challenge ciphertext, $\mathbb{B}$ calls its challenge oracle with $K \parallel \mathsf{meta}$ to obtain $e^\star$, and then sets $e_j^\star = e^\star$ for user j's entry point (in line 47 of Layout). We note that in the random oracle the real encryption key $Z_j = \hat{\mathrm{H}}(\text{“key”} \parallel k_j)$ is independent from adversary $\mathscr{A}$'s view, so we can replace it with the random key of the IND$-CCA2 challenger. For other users i it proceeds as follows:

- If $i < j$, it sets $e_i^\star$ to a random string of appropriate length.
- If $i > j$, it computes $e_i^\star$ as per line 47 of Layout.

Thereafter, $\mathbb{B}$ answers decryption queries as before. Except that whenever,$\mathbb{B}$ derives key k_j for user j, it will use its decryption oracle $\mathscr{D}_Z(\cdot)$. Note that in particular, because of the changes in G_3, $\mathbb{B}$ will not make $\mathscr{D}_Z(\cdot)$ queries on $e_i^\star$ from the challenge ciphertext $c^\star$.

If $b = 0$, $\mathbb{B}$ simulates H_{j-1}, and if $b = 1$, it simulates H_j. Therefore, if $\mathscr{A}$ distinguishes between H_{j-1} and H_j, then $\mathbb{B}$ breaks the IND$-CCA2 security of Π. To show that G_3 is indistinguishable from G_4, repeat this argument r times. More precisely:

$$\left|\Pr[W_3] - \Pr[W_4]\right| \leq r \cdot \mathsf{Adv}^{\mathsf{ind\$\text{-}cca2}}_{\Pi,\mathscr{A}}(1^\lambda).$$

G_4 <–> G_5.
The challenger's actions in G_4 and G_5 only differ if $\mathscr{A}$ could create a decryption request $\mathsf{qDec}(pk_i(S_i), c)$ where $\texttt{Unhide}(\tau) = X_i^\star$, $e_i = e_i^\star$, and the integrity tag σ is valid but c is different from $c^\star$ (recall $\mathscr{A}$ is not allowed to query $c^\star$ itself). We show that if $\mathscr{A}$ can cause the challenger to output $\perp$ incorrectly, then we can build a simulator $\mathbb{B}$ that breaks the strong unforgeability of MAC.

Assume a simulator $\mathbb{B}$ that tries to win an unforgeability game. Simulator $\mathbb{B}$ receives access to the oracles $\mathscr{M}(\cdot)$ and $\mathscr{V}(\cdot)$, and needs to output a pair (c, σ), such that $\mathscr{V}_{K_{mac}}(c, \sigma)$ returns true.

Simulator $\mathbb{B}$ now proceeds as follows. When creating the challenge ciphertext $c^\star$, it does not compute σ in step (8) of MsPURB.Enc using $K^\star$, but instead uses its oracle $\mathscr{M}$ and sets

$\sigma = \mathscr{M}(c')$. Note that because of the random oracle model for H′ and the fact that $\mathscr{A}$'s view is independent of $K^\star$, this change of K_{mac} remains undetected.

Whenever $\mathscr{A}$ makes a decryption query $\mathsf{qDec}(pk_i(S_i), c)$ $\mathbb{B}$ proceeds as before, except when it derives the key K^*. In that case it runs $\mathscr{V}(c', \sigma)$ to use its oracle to verify the MAC in step (5) of MsPURB.Dec. If $\mathscr{V}(c', \sigma)$ returns $\top$ then $\mathbb{B}$ outputs (c', σ) as its forgery (by construction, c' was not queried to the MAC oracle $\mathscr{M}(\cdot)$).

Therefore, $\mathscr{A}$ cannot make queries that cause the challenger to incorrectly output $\perp$, and therefore the two games are indistinguishable, provided MACis strongly unforgeable. More precisely:

$$\left|\Pr[W_4] - \Pr[W_5]\right| \leq \mathsf{Adv}^{\mathsf{suf}}_{\mathsf{MAC},\mathscr{A}}(1^\lambda).$$

G_5 <–> G_6.
If $\mathscr{A}$ can distinguish between G_5 and G_6, then we can build a distinguisher $\mathbb{B}$ that breaks the indistinguishability from random bits (MAC-IND$) of MAC.

Distinguisher $\mathbb{B}$ proceeds as follows to compute the challenge ciphertext $c^\star$. It proceeds as before, except that in step (9) of MsPURB.Enc, it submits c' to its challenge oracle to receive a tag $\tau^\star$. It then sets $\tau = \tau^\star$ and proceeds to construct the PURB ciphertext.

Note that as per the changes before, $\mathbb{B}$ never needs to verify a MAC under the key that was used to create $\tau^\star$ for the challenge ciphertext. Moreover, as before, $\mathscr{A}$'s view is independent of the $K^\star$, so also this change of K_{mac} remains undetected.

If $b = 0$, $\mathbb{B}$ simulates G_5, and if $b = 1$, $\mathbb{B}$ simulates G_6. Hence, if $\mathscr{A}$ can distinguish between these two games, $\mathbb{B}$ breaks the MAC-IND$ game. More precisely:

$$\left|\Pr[W_5] - \Pr[W_6]\right| \leq \mathsf{Adv}^{\mathsf{ind\$}}_{\mathsf{MAC},\mathscr{A}}(1^\lambda).$$

G_6 <–> G_7.
If $\mathscr{A}$ can distinguish between G_6 and G_7, then we can build a distinguisher $\mathbb{B}$ that breaks the IND$-CPA property of (Enc, Dec). In the IND$-CPA game [165], $\mathbb{B}$ receives:

- a challenge ciphertext $c_{\mathsf{payload}} = c_b$, s.t. $c_0 = \mathsf{Enc}_{K_{enc}}(m)$ on a chosen-by-$\mathbb{B}$ m, $c_1 \xleftarrow{\$} \{0,1\}^{|c_0|}$, and $b \xleftarrow{\$} \{0,1\}$.

$\mathbb{B}$ runs MsPURB.Dec as before to create a challenge for $\mathscr{A}$, except that $\mathbb{B}$ uses the IND$-CPA challenge ciphertext c_{payload} in step (6), instead of encrypting, as $\mathbb{B}$ does not know K_{enc}. As before, $\mathscr{A}$'s view is independent of $K^\star$, so also this change of K_{enc} remains undetected.

$\mathbb{B}$ answers decryption queries $\mathsf{qDec}(pk_i(S_i), c)$ from $\mathscr{A}$ as before. In particular

- if $\mathtt{Unhide}(\tau) = X_i^\star$ and $e_i = e_i^\star$, $\mathbb{B}$ returns $\perp$ as per the changes in G_5;
- Otherwise, $\mathbb{B}$ runs $\mathsf{MsPURB.Dec}(\cdot)$.

If $b = 0$, $\mathbb{B}$ simulates G_6, and, if $b = 1$, $\mathbb{B}$ simulates G_7. Hence, if $\mathscr{A}$ can distinguish between these two games, $\mathbb{B}$ can break the the IND$-CPA property of $(\mathsf{Enc}, \mathsf{Dec})$. More precisely:

$$\left|\Pr[W_6] - \Pr[W_7]\right| \leq \mathsf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{(\mathsf{Enc},\mathsf{Dec}),\mathscr{A}}(1^\lambda).$$

Combining the individual inequalities we find that there exists adversaries $\mathbb{B}_1, \ldots, \mathbb{B}_5$ such that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda) \leq r\left(\mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathbb{B}_1}(1^\lambda) + \mathsf{Adv}^{\mathsf{ind\$\text{-}cca2}}_{\Pi,\mathbb{B}_2}(1^\lambda)\right) + \\ \mathsf{Adv}^{\mathsf{suf}}_{\mathsf{MAC},\mathbb{B}_3}(1^\lambda) + \mathsf{Adv}^{\mathsf{ind\$}}_{\mathsf{MAC},\mathbb{B}_4}(1^\lambda) + \\ \mathsf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{(\mathsf{Enc},\mathsf{Dec}),\mathbb{B}_5}(1^\lambda),\end{aligned}$$

completing the proof. □

A.1.3 Proof of Theorem 2

For our MsPURB IND$-CPA recipient-privacy game, we take inspiration from the single-suite recipient-privacy game defined by Barth et al. [16], but we restate it in the IND$-CPA setting.

Game Recipient-Privacy.
The game is between a challenger and an adversary $\mathscr{A}$, and proceeds along the following phases:

Init: The challenger and adversary take λ as input. The adversary outputs a number of recipients r and corresponding cipher suites $S_1, \ldots, S_r$ it wants to attack. Let s be the number of unique cipher suites. The challenger verifies, for each $i \in \{1, \ldots, r\}$, that S_i is a valid cipher suite, i.e., that it a valid output of $\mathsf{MSBE.Setup}(1^\lambda)$. The challenger aborts, and sets $b^\star \xleftarrow{\$} \{0,1\}$ if the suites are not all valid. Adversary $\mathscr{A}$ then outputs two sets of recipients $N_0, N_1 \subseteq \{1, \ldots, n\}$ such that $|N_0| = |N_1| = r$, and the number of users in N_0 and N_1 using suite S_j is the same.

Setup: For each $i \in 1, \ldots, n$ given by $\mathscr{A}$, the challenger runs $(sk_i, pk_i) \leftarrow \mathsf{MsPURB.KeyGen}(S_i)$, where S_i is previously chosen by $\mathscr{A}$. The challenger gives two sets $R_0 = \{pk_1^0, \ldots, pk_r^0\}$ and $R_1 = \{pk_1^1, \ldots, pk_r^1\}$ to $\mathscr{A}$, where R_0, R_1 are the generated public keys of the recipients N_0, N_1 respectively. The challenger also gives to $\mathscr{A}$ all sk_i that correspond to $i \in N_0 \cap N_1$.

Challenge: $\mathscr{A}$ outputs $m^\star$. The challenger generates $c_0 = \mathsf{MsPURB.Enc}(R_0, m^\star)$ and $c_1 = \mathsf{MsPURB.Enc}(R_1, m^\star)$. The challenger flips a coin $b \xleftarrow{\$} \{0,1\}$ and sends $c^\star = c_b$ to $\mathscr{A}$.

Guess: $\mathscr{A}$ outputs its guess $b^\star$ for b and wins if $b^\star = b$.

We define $\mathscr{A}$'s advantage in this game as:

$$\mathsf{Adv}^{\mathsf{cpa\text{-}in}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda) = 2\left|\Pr[b = b^\star] - \tfrac{1}{2}\right|.$$

We say that a MSBE scheme is cpa-secure against insiders if $\mathsf{Adv}^{\mathsf{cpa\text{-}in}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda)$ is negligible in the security parameter.

The conditions on N_0 and N_1 in the game ensure that $\mathscr{A}$ cannot trivially win by looking at the size of the ciphertext. PURBs allows for suites with different groups (resulting in different size encodings of the corresponding IES public key) and for suites to use different authenticated encryption schemes (that could result in different sizes of encrypted entry points). Since PURBs must encode groups and entry points into the header, we mandate that for each suite the number of recipients is the same in N_0 and N_1. This assumption is similar to requiring equal-size sets of recipients in a challenge game for single-suite broadcast encryption [16]. As in broadcast encryption, if this requirement is an issue, a sender can add dummy recipients to avoid structural leakage to an insider adversary.

We will show that

$$\mathsf{Adv}^{\mathsf{cpa\text{-}in}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda) \le 2d \cdot \mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathbb{B}}(1^\lambda),$$

where d is the number of recipients in which N_0 and N_1 differ.

Proof. Similarly to Barth et al. [16], we prove recipient privacy when the sets R_0 and R_1 differ only by one public key in one suite. The general case follows by a hybrid argument. Consider the following games:

Game G_0.
This game is as the original recipient-privacy IND$-CPA game where $b = 0$ and $pk_i = R_0 \setminus R_1$, $pk_j = R_1 \setminus R_0$, where the public keys pk_i and pk_j are of the same suite S.

Game G_1.
As in G_0, but we change how a key $k_i^\star$ corresponding to the recipient i is computed in HdrPURB.Encap for the *challenge ciphertext*. Instead of computing $k_i^\star = \mathrm{H}(Y_i^x)$ (where $Y_i = pk_i$) as in step (2) of HdrPURB.Encap, we set $k_i^\star \xleftarrow{\$} \{0,1\}^{\lambda_H}$. As the challenger generates fresh public keys for each encryption query and thus a fresh key k_i, and does not have to answer decryption queries, it does not need to memorize $k_i^\star$.

Game G_2.
As in G_1, but we change the random sampling $k_i^\star$ in HdrPURB.Encap for the challenge ciphertext with $k_i^\star = \mathrm{H}(Y_j^x) = k_j^\star$ where $Y_j = pk_j$. The game now is the original recipient-privacy IND$-CPA game where $b = 1$.

Conclusion. G_0 represents the recipient-privacy game with $b = 0$ and G_2 recipient-privacy game with $b = 1$. If $\mathscr{A}$ cannot distinguish between G_0 and G_2, $\mathscr{A}$ does not have an advantage in winning the recipient-privacy game.

Let W_i be the event that $\mathscr{A}$ outputs $b^\star = 1$ in game G_i.

G_0 <–> G_1.
If $\mathscr{A}$ can distinguish between G_0 and G_1, we can build a distinguisher $\mathbb{B}$ against the IND$-CCA2 security of the IES KEM. Recall that $\mathbb{B}$ receives, from its IND$-CCA2-KEM challenger,

- a public key Y;
- a challenge $\langle X^\star, k^\star \rangle$, where depending on bit $b \xleftarrow{\$} \{0,1\}$, we have $k^\star = \mathrm{H}(Y^{x^\star})$ if $b = 0$ or $k^\star \xleftarrow{\$} \{0,1\}^{l(\lambda)}$ if $b = 1$;
- access to a $\mathsf{Decap}(\cdot)$ oracle for all but $X^\star$.

At the start of the game, $\mathbb{B}$ will set $pk_i = Y$, so that the public key of recipient i matches that of its IES KEM challenger. Note that $\mathbb{B}$ does not know the corresponding private key y_i. For all other recipients h, $\mathbb{B}$ sets $(sk_h = y_h, pk_h = Y_h) = \mathsf{MsPURB.KeyGen}(S_h)$. As $\mathscr{A}$ plays an IND$-CPA game, $\mathbb{B}$ does not need to use the $\mathsf{Decap}(\cdot)$ oracle (in fact, for IND$-CPA recipient privacy IND$-CPA security of the IES KEM suffices).

If $b = 0$ in the IES-KEM challenge, then $\mathbb{B}$ simulates G_0, and, If $b = 1$, $\mathbb{B}$ simulates G_1. Hence, if $\mathscr{A}$ distinguishes between G_0 and G_1, $\mathbb{B}$ wins in the IND$-CCA2 IES-KEM game. Therefore:

$$|\Pr[W_0] - \Pr[W_1]| \leq \mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathbb{B}}(1^\lambda)$$

G_1 <–> G_2.
The proof follows the same steps as the proof of G_0 <–> G_1. Therefore:

$$|\Pr[W_0] - \Pr[W_1]| \leq \mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathbb{B}}(1^\lambda).$$

Let d be the number of recipients that differ in N_0 and N_1. Then by repeating the above two steps d times in a hybrid argument, we find that:

$$\mathsf{Adv}^{\mathsf{cpa\text{-}in}}_{\mathsf{msbe},\mathscr{A}}(1^\lambda) \leq 2d \cdot \mathsf{Adv}^{\mathsf{cca2}}_{\mathsf{KEM},\mathbb{B}}(1^\lambda),$$

as desired. □

A.2 Analysis of Verifiable Single-Server PIR

For simplicity, we assume that each database record is only a single bit long. All of our definitions generalize naturally to the case in which database records are long bit strings.

Additional notation. In the following, we use $\mathsf{SD}(\cdot,\cdot)$ to denote the statistical distance between two distributions. The empty string is denoted with ϵ.

A.2.1 Our Definitions

Definition 7 (Verifiable single-server PIR correctness)**.** *A verifiable single-server PIR scheme* (Digest, Query, Answer, Reconstruct) *parametrized by a database length $n \in \mathbb{N}$, satisfies* correctness *if for every database $x \in \{0,1\}^n$, $i \in [n]$, and $\lambda \in \mathbb{N}$, the following holds:*

$$\Pr\left[x_i' = x_i : \begin{array}{c} d \leftarrow \mathsf{Digest}(1^\lambda, x) \\ (\mathsf{st}, q) \leftarrow \mathsf{Query}(d, i) \\ a \leftarrow \mathsf{Answer}(d, x, q) \\ x_i' \leftarrow \mathsf{Reconstruct}(\mathsf{st}, a) \end{array}\right] = 1,$$

.

Definition 8 (Verifiable single-server PIR integrity)**.** *A verifiable single-server PIR scheme* (Digest, Query, Answer, Reconstruct) *parametrized by a database length $n \in \mathbb{N}$ provides* integrity *if for every efficient adversary $\mathscr{A}$, for every database $x \in \{0,1\}^n$ and $i \in [n]$,*

$$\Pr\left[x_i' \notin \{x_i, \perp\} : \begin{array}{c} d \leftarrow \mathsf{Digest}(1^\lambda, x) \\ (\mathsf{st}, q) \leftarrow \mathsf{Query}(d, i) \\ a^* \leftarrow \mathscr{A}(d, x, q) \\ x_i' \leftarrow \mathsf{Reconstruct}(\mathsf{st}, a^*) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Definition 9 (Verifiable single-server PIR security)**.** *Let* (Digest, Query, Answer, Reconstruct) *be a verifiable single-server PIR scheme parametrized by a database length $n \in \mathbb{N}$, For $x \in \{0,1\}^n$, $i \in [n]$, $\lambda \in \mathbb{N}$, and an adversary $\mathscr{A} = (\mathscr{A}_0, \mathscr{A}_1)$, define the distribution*

$$\mathit{REAL}_{\mathscr{A},x,i,\lambda} := \left\{ \hat{\beta} : \begin{array}{l} d \leftarrow \mathsf{Digest}(1^\lambda, x) \\ (\mathsf{st}, q) \leftarrow \mathsf{Query}(d, i) \\ (\mathsf{st}_{\mathscr{A}}, a^*) \leftarrow \mathscr{A}_0(d, x, q) \\ x_i' \leftarrow \mathsf{Reconstruct}(\mathsf{st}, a^*) \\ b \leftarrow \mathbb{1}\{x_i' \neq \perp\} \\ \hat{\beta} \leftarrow \mathscr{A}_1(\mathsf{st}_A, b) \end{array} \right\}.$$

Similarly, for $n \in \mathbb{N}$, $x \in \{0,1\}^n$, $\lambda \in \mathbb{N}$, and a simulator $\mathscr{S} = (\mathscr{S}_0, \mathscr{S}_1)$, let the distribution

$$\mathit{IDEAL}_{\mathscr{A},\mathscr{S},x,\lambda} := \left\{ \beta : \begin{array}{l} d \leftarrow \mathsf{Digest}(1^\lambda, x) \\ (\mathsf{st}_{\mathscr{S}}, q) \leftarrow \mathscr{S}_0(d, x) \\ (\mathsf{st}_{\mathscr{A}}, a^*) \leftarrow \mathscr{A}_0(d, x, q) \\ b \leftarrow \mathscr{S}_1(\mathsf{st}_{\mathscr{S}}, a^*) \\ \beta \leftarrow \mathscr{A}_1(\mathsf{st}_{\mathscr{A}}, b) \end{array} \right\}.$$

A verifiable PIR scheme (Digest, Query, Answer, Reconstruct) *parametrized by a database length $n \in \mathbb{N}$, is* secure *if for every adversary $\mathscr{A} = (\mathscr{A}_0, \mathscr{A}_1)$ there exists a simulator $\mathscr{S} = (\mathscr{S}_0, \mathscr{S}_1)$ such that for every $x \in \{0,1\}^n$, $i \in [n]$, the following holds:*

$$\mathsf{SD}\left(\mathit{REAL}_{\mathscr{A},X,i,\lambda}, \mathit{IDEAL}_{\mathscr{A},\mathscr{S},X,\lambda}\right) \leq \mathsf{negl}(\lambda).$$

A.2.2 The Generic-Group Model

We analyze the security of our verifiable single-server PIR scheme in the generic-group model [170]. The generic-group model captures security against adversaries that make "black-box" use of the group operation in the cyclic group $\mathbb{G}$ over which the PIR scheme operates.

An algorithm in this model has access to:

- a "generator" oracle that outputs a bit string representing the canonical generator g of the group $\mathbb{G} = \langle g \rangle$,
- a "group-operation" oracle that takes as input bit strings σ_x and σ_y representing two group elements $g^x \in \mathbb{G}$ and $g^y \in \mathbb{G}$ and outputs the bit string σ_{x+y} representing their product $g^{x+y} \in \mathbb{G}$ under the group operation in $\mathbb{G}$, and
- a "hash" oracle that takes as input a bit string and outputs the bit string representing a random group element in $\mathbb{G}$.

While we believe that it is possible to prove security of our verifiable-PIR scheme under a standard assumption (e.g., the decision Diffie Hellman assumption [33]), we use the generic-group model here for simplicity.

A.2.3 A Useful Lemma

Lemma 10 will be useful for proving our main security results for the single-server setting. Let $\Pi_{\mathbb{G}}$ be the verifiable single-server PIR scheme $(\mathsf{Digest}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Reconstruct})$ introduced in construction 3, parametrized by a number of database rows $n \in \mathbb{N}$ and a group $\mathbb{G}$ of prime order $q \approx 2^\lambda$. Then, for a generic-group adversary $\mathscr{A}$ making at most Q group-operation-oracle queries, database $x \in \{0,1\}^n$, and integer $i \in [n]$, define the function $\epsilon_{\mathscr{A},\Pi_{\mathbb{G}}}(Q)$ as

$$\epsilon_{\mathscr{A},\Pi_{\mathbb{G}}}(Q) := \Pr\left[\begin{array}{c} a \neq a^* \\ x'_i \neq \bot \end{array} : \begin{array}{r} d \leftarrow \mathsf{Digest}(1^\lambda, x) \\ (\mathsf{st}, q) \leftarrow \mathsf{Query}(d, i) \\ a \leftarrow \mathsf{Answer}(d, x, q) \\ a^* \leftarrow \mathscr{A}(d, x, q) \\ x'_i \leftarrow \mathsf{Reconstruct}(\mathsf{st}, a^*) \end{array}\right].$$

Informally, the quantity $\epsilon_{\mathscr{A},\Pi_{\mathbb{G}}}(\cdot)$ indicates the probability that a malicious database owner deviates from the prescribed protocol in a way that causes the client to recover an incorrect output *and* the reconstruction algorithm accepts.

Lemma 10. *Modeling $\mathbb{G}$ as a generic group, for all adversaries $\mathscr{A}$ making at most Q queries to the group-operation oracle, it holds that $\epsilon_{\mathscr{A},\Pi_{\mathbb{G}}}(Q) \leq \mathcal{O}\left(Q^2\right)/p$.*

Proof. We prove the lemma using a sequence of games. For $i \in \{0,1,\ldots\}$, let W_i be the event that adversary $\mathscr{A}$ wins in Game i.

Game 0. Game 0 proceeds exactly as in the definition of $\epsilon_{\Pi,\mathbb{G}}$. The adversary $\mathscr{A}$ wins in Game 0 if $a \neq a^*$ and $x'_i \neq \perp$. In the game, the adversary interacts with a group-operation oracle that the experiment faithfully simulates. By definition, we have: $\Pr[W_0] = \epsilon_{\mathscr{A},\Pi_{\mathbb{G}}}(Q)$.

Game 1. Game 1 proceeds exactly the same as Game 0, except that we change the simulation of the generic-group-oracle, following Shoup [170]. To each group-element output $H(j)$ by the hash function $H(\cdot)$, we associate a formal variable H_j. At the *end* of the interaction, the oracle samples independent random values for each of these formal variables $(H_1, H_2, \ldots)$, to map the generic-group symbols to discrete logs in $\mathbb{Z}_p$. In this way, in Game 1, the adversary has no information—in an information-theoretic sense—on the discrete logs of any of the group elements that $H(\cdot)$ outputs.

A standard argument [170] shows that the only deviation between Game 0 and Game 1 occurs when the random choice of discrete logs at the end of the game induces a collision (two different group elements who have the same discrete log). The probability of a single such collision is $1/p$ and taking a union bound over all $\binom{Q}{2}$ pairs of elements yields: $|\Pr[W_0] - \Pr[W_1]| \leq O(Q^2/p)$.

Game 2. We again change the experiment, now replacing r and t with formal variables R and T. Now, the probability of any pair of group elements colliding is $2/p$, by the Schwartz-Zippel Lemma, so using an argument as in Game 1, we have: $|\Pr[W_1] - \Pr[W_2]| \leq O(Q^2/p)$.

Game 3. The adversary's view is now independent of the values r, t, and of all discrete logs of elements outputted by $H(\cdot)$. Assume that the adversary received the group element $a^* \in \mathbb{G}$ as the result of a group-operation-oracle query. (If not, the adversary has a negligible chance of winning.) Then, the discrete log of the value $a^* \in \mathbb{G}$ that the adversary outputs is a linear combination of the discrete log of the group elements it received.

In particular, the adversary only can compute group elements whose discrete logs are (linear combinations of):

$$(H_1, \ldots, H_n), \quad (H_1 R, \ldots, H_{i-1} R, H_{i+1} R, \ldots H_n R), \quad H_i R + H_i T.$$

For Reconstruct to accept, it must be that $a^* \in \{d^r, d^r \cdot H(i)^t\} \in \mathbb{G}$. The group elements on the right-hand side of this relation has a discrete log of the form

$$(\sum_{j=1}^{n} x_j H_j) R \quad \in \mathbb{Z}_p[R, T, H_1, \ldots, H_n] \tag{A.1}$$

$$\underbrace{(\sum_{j=1}^{n} x_j H_j) R}_{\text{digest } d} + H_i T \tag{A.2}$$

The answer a that the honest server sends has discrete log $(\sum_{j=1}^{n} x_j H_j)R + x_i H_i T$. We claim that if $a^* \neq a$, then the polynomial representing the discrete log of adversary's answer must differ from the two accepting polynomials (A.1) and (A.2). There are two cases:

- **Case** $x_i = 0$: The honest server's reply is $(\sum_{j=1}^{n} x_j H_j)R$. The adversarial server's reply is thus not equal to (A.1) as a formal polynomial. In addition, the adversary's reply cannot be equal to (A.2) as a formal polynomial either. This is because the monomial $H_i T$ appears in (A.2) but the monomial $H_i R$ does not. No linear combination of the adversary's group elements amounts to the relation in (A.2).
- **Case** $x_i = 1$: As Case $x_i = 0$; swap roles of of (A.1) and (A.2).

If the adversarial server sends any other group element ($a^* \neq a$), then the corresponding discrete logs will be unequal as formal polynomials. The probability that the client accepts is at most $2/|G| = 2/p$, by the Schwartz-Zippel Lemma. A union bound over the two possible relations, (A.1) and (A.2), that the adversary can satisfy gives $\Pr[W_3] \leq O(1/p)$. This completes the proof of the lemma. □

A.2.4 Security Proofs

The correctness of our scheme follows by construction. Therefore, we focus here on integrity and security.

Theorem 11. *In the generic-group model, our verifiable single-server PIR construction (Construction 3) satisfies integrity.*

Proof sketch. The result follows almost immediately from Lemma 10. In the integrity game, the adversarial server returns the correct answer ($a = a^*$, using the notation of Lemma 10) or not ($a \neq a^*$). In the first case, $x_i' = x_i$ and there is no integrity problem. In the second case, Lemma 10 implies that $x_i' = \perp$, except with negligible probability in the security parameter. □

Theorem 12. *In the generic-group model, our verifiable single-server PIR construction (Construction 3) satisfies security (Definition 9), provided that the underlying single-server PIR scheme is secure.*

Proof sketch. Again, the result follows almost immediately from Lemma 10. The simulator $\mathscr{S}$ simulates the client's query q by appealing to the DDH assumption (which holds unconditionally in the generic-group model). Then, the simulator inspects the adversary's response. If the adversary answered the client's query according to the protocol (i.e., the adversary's answer is $\mathsf{Answer}(d, x, q)$ on digest d and database x), the simulator outputs $b = 1$ and the simulator outputs $b = 0$ otherwise. Lemma 10 implies the correctness of this simulation. □

Bibliography

[1] M. Abdalla, M. Bellare, and P. Rogaway, “The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES”, in *Cryptographers’ Track at the RSA Conference*, 2001.

[2] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, “XPIR: private information retrieval for everyone”, *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, 2016.

[3] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt, “On the Security of RC4 in TLS”, in *USENIX Security Symposium*, 2013.

[4] K. Alhamed, M. C. Silaghi, I. Hussien, R. Stansifer, and Y. Yang, “"Stacking the Deck" Attack on Software Updates: Solution by Distributed Recommendation of Testers”, in *IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, 2013.

[5] K. Alhamed, M. C. Silaghi, I. Hussien, and Y. Yang, “Security by Decentralized Certification of Automatic-Updates for Open Source Software controlled by Volunteers”, in *Workshop on Decentralized Coordination*, 2013.

[6] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo, “Communication–computation trade-offs in PIR”, in *USENIX Security Symposium*, 2021.

[7] M. Ambrosin, C. Busold, M. Conti, A.-R. Sadeghi, and M. Schunter, “Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution over Untrusted Cache-enabled Networks”, in *European Symposium on Research in Computer Security*, 2014.

[8] S. Angel, H. Chen, K. Laine, and S. Setty, “PIR with compressed queries and amortized query processing”, in *IEEE Symposium on Security and Privacy*, 2018.

[9] S. Angel and S. Setty, “Unobservable Communication over Fully Untrusted Infrastructure”, in *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[10] D. F. Aranha, P.-A. Fouque, C. Qian, M. Tibouchi, and J.-C. Zapalowicz, “Binary Elligator Squared”, in *Selected Areas in Cryptography*, 2014.

[11] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, “DNS Security Introduction and Requirements”, RFC 4003, 2005.

[12] J. Atack, *Gitian building*, https://github.com/jonatack/bitcoin-development/blob/master/gitian-building.md, Accessed June 12, 2021, Apr. 29, 2021.

[13] H. Ballani, P. Francis, and X. Zhang, "A Study of Prefix Hijacking and Interception in the Internet", in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 37, 2007.

[14] D. Barrera, W. Enck, and P. C. van Oorschot, "Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems", in *IEEE Mobile Security Technologies*, 2012.

[15] D. Barrera, D. McCarney, J. Clark, and P. C. van Oorschot, "Baton: Certificate Agility for Android's Decentralized Signing Infrastructure", in *ACM Conference on Security and Privacy in Wireless & Mobile Networks*, 2014.

[16] A. Barth, D. Boneh, and B. Waters, "Privacy in Encrypted Content Distribution Using Private Broadcast Encryption", in *Financial Cryptography and Data Security*, 2006.

[17] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: Attack Resilient Public-Key Infrastructure", in *ACM Conference on Computer and Communications Security*, 2014.

[18] A. Beimel and Y. Stahl, "Robust Information-Theoretic Private Information Retrieval", in *Security in Communication Networks*, 2002.

[19] ——, "Robust Information-Theoretic Private Information Retrieval", *J. Cryptology*, vol. 20, no. 3, 2007.

[20] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval, "Key-Privacy in Public-Key Encryption", in *ASIACRYPT*, 2001.

[21] M. Bellare, A. Boldyreva, K. Kurosawa, and J. Staddon, "Multi-Recipient Encryption Schemes: Efficient Constructions and Their Security", *IEEE Transactions on Information Theory*, vol. 53, no. 11, 2007.

[22] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication", in *CRYPTO*, 1996.

[23] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm", *J. Cryptology*, vol. 21, no. 4, 2008.

[24] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols", in *ACM Conference on Computer and Communications Security*, 1993.

[25] M. Bellare and B. Tackmann, "The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3", in *CRYPTO*, 2016.

[26] A. Bellissimo, J. Burgess, and K. Fu, "Secure Software Updates: Disappointments and New Challenges", in *USENIX Workshop on Hot Topics in Security*, Jul. 2006.

[27] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi, "Duqu: Analysis, Detection, and Lessons Learned", in *ACM European Workshop on System Security (EuroSec)*, 2012.

[28] L. Bernaille and R. Teixeira, "Early recognition of encrypted applications", in *Passive and Active Network Measurement*, Apr. 2007.

[29] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings", in *ACM Conference on Computer and Communications Security*, Nov. 2013.

[30] S. Bhat, D. Lu, A. Kwon, and S. Devadas, "Var-CNN: A data-efficient website fingerprinting attack based on deep learning", *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 4, 2019.

[31] A. Biryukov, D. Dinu, and D. Khovratovich, *Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications*, https://www.password-hashing.net/argon2-specs.pdf, Dec. 26, 2015.

[32] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine, "Privacy vulnerabilities in encrypted HTTP streams", in *Privacy Enhancing Technologies*, May 2005.

[33] D. Boneh, "The decision Diffie-Hellman problem", in *Algorithmic Number Theory*, 1998.

[34] D. Boneh, C. Gentry, and B. Waters, "Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys", in *CRYPTO*, 2005.

[35] J. Bonneau, "EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log", in *Financial Cryptography and Data Security*, 2016.

[36] M. Borge, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, and B. Ford, "Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies", in *IEEE Security and Privacy on the Blockchain*, Apr. 2017.

[37] N. Borisov, G. Danezis, and I. Goldberg, "DP5: a private presence service", *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, 2015.

[38] E. Boyle, Y. Ishai, R. Pass, and M. Wootters, "Can we access a database both locally and privately?", in *Theory of Cryptography*, 2017.

[39] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP", in *CRYPTO*, 2012.

[40] E. Brumaghin, R. Gibb, W. Mercer, M. Molyett, and C. Williams, *CCleanup: a vast number of machines at risk*, https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html, Sep. 18, 2017.

[41] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication", in *EUROCRYPT*, 1999.

[42] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format", RFC 4880, Nov. 2007.

[43] J. Cappos, "Avoiding theoretical optimality to efficiently and privately retrieve security updates", in *Financial Cryptography and Data Security*, Apr. 2013.

[44] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A Look In the Mirror: Attacks on Package Managers", in *ACM Conference on Computer and Communications Security*, Oct. 2008.

[45] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", in *USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[46] Z. Chai, A. Ghafari, and A. Houmansadr, "On the Importance of Encrypted-SNI (ESNI) to Censorship Circumvention", in *Free and Open Communications on the Internet*, 2019.

[47] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: a reality today, a challenge tomorrow", in *IEEE Symposium on Security and Privacy*, 2010.

[48] H. Cheng and R. Avnur, *Traffic analysis of SSL encrypted web browsing*, https://people.eecs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps, UC Berkeley, 1998.

[49] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private Information Retrieval", in *Foundations of Computer Science*, 1995.

[50] Click Studios, *PASSWORDSTATE: incident management advisory #01*, https://www.clickstudios.com.au/advisories/Incident_Management_Advisory-01-20210424.pdf, Apr. 24, 2021.

[51] S. Colombo, K. Nikitin, B. Ford, and H. Corrigan-Gibbs, "Verifiable private information retrieval", *Under submission*, 2021.

[52] H. Corrigan-Gibbs and D. Kogan, "Private Information Retrieval with Sublinear Online Time", in *EUROCRYPT*, 2020.

[53] G. D. Crescenzo, T. Malkin, and R. Ostrovsky, "Single Database Private Information Retrieval Implies Oblivious Transfer", in *EUROCRYPT*, 2000.

[54] D. Curry, *Signal revenue & usage statistics*, https://www.businessofapps.com/data/signal-statistics/, Accessed 15 April 2021, Mar. 10, 2021.

[55] G. Danezis, *Traffic analysis of the HTTP protocol over TLS*, http://wwwo.cs.ucl.ac.uk/staff/G.Danezis/papers/TLSanon.pdf, 2009.

[56] G. Danezis and R. Clayton, "Introducing Traffic Analysis", *Digital Privacy: Theory, Technologies, and Practices*, Dec. 2007.

[57] G. Danezis and I. Goldberg, "Sphinx: A Compact and Provably Secure Mix Format", in *IEEE Symposium on Security and Privacy*, May 2009.

[58] E. Dauterman, E. Feng, E. Luo, R. A. Popa, and I. Stoica, "DORY: An Encrypted Search System with Distributed Trust", in *USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[59] Debian, *Overview of reproducible builds for packages in bullseye for amd64*, https://tests.reproducible-builds.org/debian/testing/index_suite_amd64_stats.html, Jun. 11, 2021.

[60] ——, *Debian popularity contest*, http://popcon.debian.org/by_inst, Accessed September, 2016.

[61] ——, *Package Management*, https://wiki.debian.org/PackageManagement.

[62] ——, *Reproducible builds for required packages (amd64)*, https://tests.reproducible-builds.org/debian/testing/amd64/pkg_set_required.html, Accessed September, 2016.

[63] ——, *snapshot.debian.org*, Accessed: September, 2016.

[64] S. E. Deering and D. R. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs", *ACM Transactions on Computer Systems*, vol. 8, no. 2, May 1990.

[65] C. Delerablée, "Identity-Based Broadcast Encryption with Constant Size Ciphertexts and Private Keys", in *Theory and Application of Cryptology and Information Security*, 2007.

[66] C. Devet, I. Goldberg, and N. Heninger, "Optimally Robust Private Information Retrieval", in *USENIX Security Symposium*, 2012.

[67] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, Aug. 2008.

[68] W. Diffie and M. E. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory (IT)*, 1976.

[69] Docker, *What is Docker?*https://www.docker.com/what-docker, Sep. 2016.

[70] N. Döttling, S. Garg, Y. Ishai, G. Malavolta, T. Mour, and R. Ostrovsky, "Trapdoor Hash Functions and Their Applications", in *CRYPTO*, 2019.

[71] V. Durham, *Namecoin*, https://namecoin.info, 2011.

[72] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson, "The security impact of HTTPS interception", in *Network and Distributed System Security Symposium*, 2017.

[73] *EasyInstall Module*, http://peak.telecommunity.com/DevCenter/EasyInstall.

[74] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith, "Hey, NSA: Stay Away from My Market! Future Proofing App Markets Against Powerful Attackers", in *ACM Conference on Computer and Communications Security*, 2014.

[75] J. Fan and F. Vercauteren, *Somewhat Practical Fully Homomorphic Encryption*, Cryptology ePrint Archive, Report 2012/144, 2012.

[76] A. Faz-Hernández and K. Kwiatkowski, *Introducing CIRCL: an advanced cryptographic library*, https://github.com/cloudflare/circl, Cloudflare, 2019.

[77] N. Fazio and I. M. Perera, "Outsider-Anonymous Broadcast Encryption with Sublinear Ciphertexts", in *Public Key Cryptography*, 2012.

[78] FireEye, *Highly evasive attacker leverages SolarWinds supply chain to compromise multiple global victims with SUNBURST backdoor*, https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html, Dec. 13, 2020.

[79] B. Ford, *Apple, FBI, and Software Transparency*, Freedom to Tinker, https://freedom-to-tinker.com/blog/bford/apple-fbi-and-software-transparency/, Mar. 2016.

[80] Frields, Paul W., *Infrastructure Report*, https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, Aug. 22, 2008.

[81] S. Frolov and E. Wustrow, "The use of TLS in Censorship Circumvention", in *Network and Distributed System Security Symposium*, 2019.

[82] C. Gentry and Z. Ramzan, "Single-database private information retrieval with constant communication rate", in *International Colloquium on Automata, Languages, and Programming*, 2005.

[83] C. Gentry and B. Waters, "Adaptive Security in Broadcast Encryption Systems (with Short Ciphertexts)", in *Theory and Applications of Cryptographic Techniques*, A. Joux, Ed., 2009.

[84] *Git-notes Documentation*, Sep. 2016.

[85] *Go cryptography*, https://pkg.go.dev/golang.org/x/crypto.

[86] I. Goldberg, "Improving the robustness of private information retrieval", in *IEEE Symposium on Security and Privacy*, 2007.

[87] S. Goldwasser and S. Micali, "Probabilistic encryption", *Journal of Computer and System Sciences*, vol. 28, no. 2, 1984.

[88] B. Greschbach, G. Kreitz, and S. Buchegger, "The devil is in the metadata — New privacy challenges in Decentralised Online Social Networks", in *Pervasive Computing and Communications Workshops*, Mar. 2012.

[89] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish, "Scalable and private media consumption with Popcorn", in *USENIX Conference on Networked Systems Design and Implementation*, 2016.

[90] B. Harsha, R. Morton, J. Blocki, J. Springer, and M. Dark, "Bicycle Attacks Considered Harmful: Quantifying the Damage of Widespread Password Length Leakage", *Computers & Security*, vol. 100, 2021.

[91] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique", in *USENIX Security Symposium*, Aug. 2016.

[92] P. Hoffman and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, Aug. 2012.

[93] A. Houmansadr, C. Brubaker, and V. Shmatikov, "The parrot is dead: Observing unobservable network communications", in *IEEE Symposium on Security and Privacy*, May 2013.

[94] M. Howell, *Homebrew – The Missing Packet Manager for macOS*, https://brew.sh/.

[95] M. Husák, M. Čermák, T. Jirsík, and P. Čeleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", *EURASIP Journal on Information Security*, vol. 2016, no. 1, 2016.

[96] IDRIX, *Veracrypt*, https://www.veracrypt.fr/en/Home.html.

[97] T. Intercept, *Strawhorse: Attacking the macOS and iOS Software Development Kit*, https://theintercept.com/document/2015/03/10/strawhorse-attacking-macos-ios-software-development-kit, Mar. 10, 2015.

[98] Y. Ishai and A. Paskin, "Evaluating branching programs on encrypted data", in *Theory of Cryptography*, 2007.

[99] C. Jarabek, D. Barrera, and J. Aycock, "ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware", in *ACM Annual Computer Security Applications Conference*, 2012.

[100] jfinnigan, *HCSEC-2021-12 - Codecov security event and HashiCorp GPG key exposure*, https://discuss.hashicorp.com/t/hcsec-2021-12-codecov-security-event-and-hashicorp-gpg-key-exposure/23512, Apr. 22, 2021.

[101] H. Johansen, D. Johansen, and R. van Renesse, "FirePatch: Secure and Time-Critical Dissemination of Software Patches", in *Information Security Conference*, May 2007.

[102] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale", in *USENIX Security Symposium*, 2019.

[103] Kaspersky Lab, *Operation ShadowHammer*, https://securelist.com/operation-shadowhammer/89992/, Mar. 25, 2019.

[104] J. Katz and Y. Lindell, *Introduction to modern cryptography*, 2nd ed. CRC press, 2014.

[105] *Keybase – Public Key Crypto for Everyone, Publicly Auditable Proofs of Identity*, 2016.

[106] *Keybase go-crypto*, https://github.com/keybase/go-crypto.

[107] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure", in *International Word Wide Web Conference (WWW)*, 2014.

[108] Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, "Private set intersection for unequal set sizes with mobile applications", *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 4, 2017.

[109] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems", in *CRYPTO*, 1996.

[110] E. Kokoris-Kogias, L. Gasser, I. Khoffi, P. Jovanovic, N. Gailly, and B. Ford, "Managing Identities Using Blockchains and CoSi", Workshop on Hot Topics in Privacy Enhancing Technologies, Tech. Rep., 2016.

[111] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing", in *USENIX Security Symposium*, 2016.

[112] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme", in *CRYPTO*, 2010.

[113] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using Delegations to Protect Community Repositories", in *USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2016.

[114] K. Kurosawa, "Multi-recipient public-key encryption with shortened ciphertext", in *Public Key Cryptography*, 2002.

[115] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: single database, computationally-private information retrieval", in *Foundations of Computer Science*, 1997.

[116] *DEDIS advanced crypto library for Go*, https://github.com/dedis/kyber, EPFL-DEDIS, 2010 – 2021.

[117] *Lattigo v2.1.1*, http://github.com/ldsec/lattigo, EPFL-LDS, Dec. 2020.

[118] B. Laurie, "Certificate transparency", *Communications of the ACM*, vol. 57, no. 10, 2014.

[119] E. Levy, "Poisoning the software supply chain", *IEEE Security & Privacy*, vol. 1, no. 3, May 2003.

[120] J. Li, M. N. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)", in *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[121] J. Li, P. L. Reiher, and G. J. Popek, "Resilient Self-organizing Overlay Networks for Security Update Delivery", *IEEE J.Sel. A. Commun.*, vol. 22, no. 1, Sep. 2006.

[122] L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart, "Protocols for checking compromised credentials", in *ACM Conference on Computer and Communications Security*, 2019.

[123] X. Li, B. A. Azad, A. Rahmati, and N. Nikiforakis, "Good bot, bad bot: characterizing automated browsing activity", in *IEEE Symposium on Security and Privacy*, 2021.

[124] H. Lipmaa, "An oblivious transfer protocol with log-squared communication", in *Conference on Information Security*, 2005.

[125] LogRhythm Labs, *NotPetya technical analysis*, https://gallery.logrhythm.com/threat-intelligence-reports/notpetya-technical-analysis-logrhythm-labs-threat-intelligence-report.pdf, Jul. 2017.

[126] Lunar, "How to make your software build reproducibly", Chaos Communication Camp, Aug. 2015.

[127] A. Malhotra, I. E. Cohen, E. Brakke, and S. Goldberg, "Attacking the Network Time Protocol", in *Network and Distributed System Security Symposium*, Feb. 2016.

[128] M. Marlinspike, *Private contact discovery*, https://signal.org/blog/private-contact-discovery/, Accessed 11 April 2021, Sep. 26, 2017.

[129] J. Mayer, P. Mutchler, and J. C. Mitchell, "Evaluating the privacy properties of telephone metadata", *Proceedings of the National Academy of Sciences*, vol. 113, no. 20, 2016.

[130] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing Key Transparency to End Users", in *USENIX Security Symposium*, 2015, pp. 383–398.

[131] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function", in *Advances in Cryptology (CRYPTO)*, 1988.

[132] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", in *Privacy Enhancing Technologies Symposium*, Jul. 2014.

[133] M. Mimoso, *D-Link Accidentally Leaks Private Code-Signing Keys*, https://threatpost.com/d-link-accidentally-leaks-private-code-signing-keys/114727, Sep. 18, 2015.

[134] *Mininet – An Instant Virtual Network on your Laptop (or other PC)*, Feb. 2018.

[135] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: an efficient oblivious search index", in *IEEE Symposium on Security and Privacy*, 2018.

[136] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, "PIR-Tor: scalable anonymous communication using private information retrieval", in *USENIX Security Symposium*, 2011.

[137] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg, "SkypeMorph: Protocol Obfuscation for Tor Bridges", in *ACM Conference on Computer and Communications Security*, 2012.

[138] C. Mouchet, J. Troncoso-Pastoriza, J.-P. Bossuat, and J.-P. Hubaux, *Multiparty Homomorphic Encryption from Ring-Learning-with-Errors*, Cryptology ePrint Archive, Report 2020304, 2020.

[139] J. I. Munro, T. Papadakis, and R. Sedgewick, "Deterministic Skip Lists", in *ACM-SIAM Symposium on Discrete Algorithms*, 1992.

[140] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, https://bitcoin.org/bitcoin.pdf, 2008.

[141] K. Nikitin, L. Barman, W. Lueks, M. Underwood, J.-P. Hubaux, and B. Ford, "Reducing metadata leakage from encrypted files and communication with PURBs", *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 4, Jul. 2019.

[142] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds", in *USENIX Security Symposium*, 2017.

[143] Null Byte, *Hack Like a Pro: How to Hijack Software Updates to Install a Rootkit for Backdoor Access*, WonderHowTo, 2014.

[144] OpenVault, *OVBI: median broadband usage on pace to surpass 250 GB per month in 2020*, https://openvault.com/ovbi-median-broadband-usage-on-pace-to-surpass-250-gb-per-month-in-2020, Accessed: May 19, 2021, Feb. 11, 2020.

[145] P. Paillier, "Public-Key cryptosystems based on composite degree residuosity classes", in *Advances in Cryptology (EUROCRYPT)*, May 1999.

[146] E. Palomar, J. Estevez-Tapiador, J. Hernandez-Castro, and A. Ribagorda, "A Protocol for Secure Content Distribution in Pure P2P Networks", in *Conference on Database and Expert Systems Applications*, 2006.

[147] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website Fingerprinting in Onion Routing Based Anonymization Networks", in *Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2011.

[148] J. Pang, B. Greenstein, R. Gummadi, S. Seshan, and D. Wetherall, "802.11 User Fingerprinting", in *Mobile Computing and Networking*, 2007.

[149] C. Papamanthou, E. Shi, and R. Tamassia, "Signatures of correct computation", in *Theory of Cryptography*, 2013.

[150] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval", in *ACM Conference on Computer and Communications Security*, 2018.

[151] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing", in *CRYPTO*, 1991.

[152] C. Percival, *Stronger key derivation via sequential memory-hard functions*, http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf, 2009.

[153] M. Perry, S. Schoen, and H. Steiner, *Reproducible Builds. Moving Beyond Single Points of Failure for Software Distribution*, Chaos Communication Congress, Dec. 2014.

[154] A. Pironti, P.-Y. Strub, and K. Bhargavan, "Identifying Website Users by TLS Traffic Analysis: New Attacks and Effective Countermeasures", INRIA, Tech. Rep., Sep. 2012.

[155] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk, "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels", in *USENIX Security Symposium*, 2018.

[156] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Communications of the ACM*, vol. 33, no. 6, Jun. 1990.

[157] Red Hat, *[RHSA-2008:0855-01] Critical: openssh security update*, https://listman.redhat.com/archives/rhsa-announce/2008-August/msg00017.html, Aug. 22, 2008.

[158] *Reproducible builds*, https://reproducible-builds.org/, Accessed June 12, 2021.

[159] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, Aug. 2018.

[160] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning", in *Network and Distributed System Security Symposium*, Feb. 2018.

[161] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds", in *ACM Conference on Computer and Communications Security*, 2009.

[162] I. Ristić, *HTTP client fingerprinting using SSL handshake analysis*, https://blog.ivanristic.com/2009/06/http-client-fingerprinting-using-ssl-handshake-analysis.html, Jun. 2009.

[163] T. Ritter and D. K. Gillmor, *Protecting the TLS Handshake*, https://datatracker.ietf.org/meeting/interim-2014-tls-01/materials/slides-interim-2014-tls-1-3, Presentation, IETF Interim, May 2014.

[164] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, 1978.

[165] P. Rogaway, "Nonce-based symmetric encryption", in *Fast Software Encryption*, 2004.

[166] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable Key Compromise in Software Update Systems", in *ACM Conference on Computer and Communications security*, Oct. 2010.

[167] R. Schuster, V. Shmatikov, and E. Tromer, "Beauty and the Burst: Remote Identification of Encrypted Video Streams", in *USENIX Security Symposium*, Aug. 2017.

[168] D. Schwartz, N. Youngs, and A. Britto, "The Ripple protocol consensus algorithm", *Ripple Labs Inc White Paper*, 2014.

[169] A. Shamir, "How to Share a Secret", *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[170] V. Shoup, "Lower bounds for discrete logarithms and related problems", in *EUROCRYPT*, 1997.

[171] C. Soghoian and S. Stamm, "Certified lies: detecting and defeating government interception attacks against SSL", in *Financial Cryptography and Data Security*, 2012.

[172] S. Son and V. Shmatikov, "The Hitchhiker's Guide to DNS Cache Poisoning", in *Security and Privacy in Communication Networks*, 2010.

[173] ssteele, *Tor at the heart: bridges and pluggable transports*, https://blog.torproject.org/tor-heart-bridges-and-pluggable-transports, Tor Blog, Dec. 11, 2016.

[174] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning", in *IEEE Symposium on Security and Privacy*, May 2016.

[175] K. Thompson, "Reflections on Trusting Trust", *Communications of the ACM*, vol. 27, no. 8, Aug. 1984.

[176] M. Tibouchi, "Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings", in *Financial Cryptography and Data Security*, 2014.

[177] S. Torres-Arias, A. K. Ammula, R. Curtmola, and J. Cappos, "On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities", in *USENIX Security Symposium*, Aug. 2016.

[178] B. VanderSloot, A. McDonald, W. Scott, J. A. Halderman, and R. Ensafi, "Quack: Scalable remote measurement of application-layer censorship", in *USENIX Security Symposium*, Aug. 2018.

[179] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast", in *Conference on Network Protocols*.

[180] G. Vranken, *HTTPS bicycle attack*, https://guidovranken.com/2015/12/30/https-bicycle-attack/, Dec. 30, 2015.

[181] L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton, "Seeing through network-protocol obfuscation", in *ACM Conference on Computer and Communications Security*, Oct. 2015.

[182] Q. Wang, X. Gong, G. T. K. Nguyen, A. Houmansadr, and N. Borisov, "Censorspoofer: asymmetric communication using IP spoofing for censorship-resistant web browsing", in *ACM Conference on Computer and Communications Security*, Oct. 2012.

[183] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective Attacks and Provable Defenses for Website Fingerprinting", in *USENIX Security Symposium*, Aug. 2014.

[184] T. Wang and I. Goldberg, "Improved Website Fingerprinting on Tor", in *ACM Workshop on Privacy in the Electronic Society*, 2013.

[185] ——, "On realistically attacking Tor with website fingerprinting", *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, Jul. 2016.

[186] X. Wang and L. Zhao, "Verifiable single-server private information retrieval", in *Information and Communications Security*, 2018.

[187] 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 *ACM Conference on Computer and Communications Security*, Oct. 2012.

[188] *Two billion users – connecting the world privately*, https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately, Accessed 15 April 2021, Feb. 12, 2020.

[189] P. Winter, T. Pulls, and J. Fuss, "ScrambleSuit: A polymorphic network protocol to circumvent censorship", in *ACM Workshop on Privacy in the Electronic Society*, 2013.

[190] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger", *Ethereum Project Yellow Paper*, 2014.

[191] World Wide Web Technology Surveys, *Usage statistics of default protocol https for websites*, https://w3techs.com/technologies/details/ce-httpsdefault, Accessed: May 19, 2021.

[192] C. V. Wright, F. Monrose, and G. M. Masson, "On inferring application protocol behaviors in encrypted network traffic", *Journal of Machine Learning Research*, vol. 7, no. 12, 2006.

[193] C. V. Wright, S. E. Coull, and F. Monrose, "Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis", in *Network and Distributed System Security Symposium*, 2009.

[194] *Yum Package Manager*, http://yum.baseurl.org/index.html.

[195] K. Zetter, *Hackers hijacked ASUS software updates to install backdoors on thousands of computers*, https://www.vice.com/en/article/pan9wn/hackers-hijacked-asus-software-updates-to-install-backdoors-on-thousands-of-computers, Mar. 25, 2019.

[196] F. Zhang, W. He, X. Liu, and P. G. B. Bridges, "Inferring Users' Online Activities Through Traffic Analysis", in *ACM Conference on Wireless Network Security*, Jun. 2011.

[197] L. F. Zhang and R. Safavi-Naini, "Verifiable multi-server private information retrieval", in *Applied Cryptography and Network Security*, 2014.

[198] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Hey, You, Get Off of My Market: Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces", in *ACM Conference on Data and Application Security and Privacy*, 2012.

Graphics Credit

Some of the icons (the server, the program, the software-update center, and the signature) in Figure 4.1 and Figure 4.2 are from vecteezy.com.

Kirill Nikitin

Updated: 16/08/2021

Curriculum Vitae
EPFL IC IINFCOM DEDIS, BC 209,
Station 14, CH-1015 Lausanne, Switzerland

mob. +47 77 9866320
kirill.nikitin@protonmail.com
https://nikirill.com

Education

Ph.D. Computer and Communication Sciences Sep 2015–Aug 2021

École polytechnique fédérale de Lausanne, Switzerland

Thesis: "Integrity and Metadata Protection in Data Retrieval"
Advisor: Bryan Ford

M.S. Communication Systems Sep 2013–Jun 2015

KTH Royal Institute of Technology, Sweden

Thesis: "DTLS Adaptation for Efficient Secure Group Communication" @ RISE SICS
Advisors: Marco Tiloca, Shahid Raza (both RISE SICS), Markus Hidell (KTH)

Diploma Information Security (with honors) Sep 2008–Jun 2013

Kazan (Volga Region) Federal University, Russia

Thesis: "Cryptographic Key Distribution via Randomness from Multipath Propagation of Radio Waves"
Advisor: Arkady Karpov

Exchange Student Computer Science Jan–May 2012

University of Helsinki, Finland

Refereed Publications [Google Scholar]

4. J. Lee, K. Nikitin, S. Setty. "Replicated state machines without replicated execution". In *IEEE Symposium on Security and Privacy*, 2020.

3. K. Nikitin, L. Barman, W. Lueks, M. Underwood, J.-P. Hubaux, B. Ford. "Reducing Metadata Leakage from Encrypted Files and Communication with PURBs". *Proceedings on Privacy Enhancing Technologies, 2019(4)*, 2019.

2. K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. "CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds". In *USENIX Security Symposium*, 2017.

1. M. Tiloca, K. Nikitin, S. Raza. "Axiom: DTLS-Based Secure IoT Group Communication". In *ACM Transactions on Embedded Computing Systems (TECS), Special Issue on Embedded Computing for IoT, 16(3), 66*, 2017.

Reports and manuscripts under submission:

2. S. Colombo, K. Nikitin, B. Ford, H. Corrigan-Gibbs. "Verifiable private information retrieval". *Under submission*, 2021.

1. C. Basescu, M. Nowlan, K. Nikitin, J. Faleiro, and B. Ford, "Crux: Locality-Preserving Distributed Services". Technical report, in *CoRR*, 1405.0637, arXiv, 2018.

Professional Experience

Doctoral Researcher Sep 2015–Aug 2021
Decentralized and Distributed Systems laboratory, **EPFL**, Lausanne, Switzerland
The lead of the research projects on:

- Exploiting and protecting metadata in encrypted files and communications;
- Security and transparency of software-distribution systems.

Research Intern Aug–Oct 2019
Confidential Computing Group, **Microsoft Research**, Cambridge, UK

- Information-flow control for confidentiality in smart contracts.

Research Intern Aug–Nov 2018
Systems Security and Privacy Group, **Microsoft Research**, Redmond, US

- Improving scalability of smart contracts via off-chain execution and verifiable computation.

External Master's Thesis Jan–Jun 2015
Security Lab, **RISE Swedish Institute of Computer Science**, Stockholm, Sweden

- Designing a protocol for secure group communication for the Internet-of-Things.

Research Intern Jun–Aug 2014
Laboratory for Cryptologic Algorithms, **EPFL**, Lausanne, Switzerland

- Integer factorization and analysis of public-key ecosystem weaknesses.

Academic Service and Extracurricular Activities

- A member of the program committee or the student editorial board for
  - JSys 2021: Journal of Systems Research
  - ACM CCS 2021 Posters & Demos
  - CryBlock 2019, 2020: Workshop on Cryptocurrencies and Blockchains for Distributed Systems
  - BlockSys 2019: Workshop on Blockchain-enabled Networked Sensor Systems
  - ICBC 2019: IEEE International Conference on Blockchain and Cryptocurrency
- An external reviewer for IEEE Transactions on Industrial Informatics 2019, IEEE Transactions on Parallel and Distributed Systems 2020, and ACM CCS 2017, 2021.
- I was a president of a graduate student association at IC EPFL. Organized invited talks, activities for current students, and helped with the organization of Open Houses for newcomers.
- Back in the past, I played in a student theater, performing team comedy stand-ups.

Teaching and Supervision

- **CS-438 Decentralized Systems Engineering** (Fall 17, 18, 20)
  As a part of a team, designed, implemented and graded homework assignments, gave guest lectures, supervised semester group projects, and evaluated student progress throughout semester. *Student reviews:*
  "The course is very demanding as time is concerned but the topics and the assigments are very fascinating." "The TAs are awesome. I can't imagine the problem they went through preparing these testcases."

- **ICC Information, Computation and Communication** (Spring 20)
  Guided students during exercise solving.
- **CS-234 Technologies of societal self-organization** (Fall 19)
  Participated in the design of the brand-new course: guiding projects and creating assignments, quizzes, and the exam.
- **COM-402 Information Security and Privacy** (Spring 17, 18)
  As a part of a team, designed and implemented CTF-style exercise labs from scratch, contributed to creating lectures, and guided students. *Student reviews:*
  "Best homeworks I've ever had at EPFL so far. They are neither too guided or to free just perfect..." "Exercises are really interesting..." "Homeworks are awesome." "Oscillating between "These exercises are insuffurable!" (before you get your token...) and "These exercises are so fun!" (after you get your token!) ..." "The Excercises are not always easy, but they are fun to do and give a good practical insight into security."
- **MATH-101 Analyse I** (Fall 16)
  Guided students during exercise solving.
- **COM-102 Advanced information, computation, communication II** (Spring 16)
  Designed exercises on basic cryptography (part of the course) and guided students.

Supervision:

- Fernando Monje Real. "Traffic analysis of real-time collaborative editors". *Master's thesis* (Spring 20).
- Carlos Villa Sánchez. "Secure management of browser extensions and their dependencies". *Master's thesis* (Spring 20).
- Charles Parzy-Turlat. "Tree-based Group Key Agreement". *Master's project* (Spring 19).
- Simone Colombo. "DecenArch: a decentralized system for privacy-conscious Web archiving against censorship". *Master's thesis* (Spring 18).
- Nicolas Plancherel. "Decentralized Internet Archive". *Master's thesis* (Fall 17).
- Nicolas Ritter. "Access Control In Real-Time Peer-to-Peer Collaboration". *Master's project* (Fall 17).
- Damien Aymon. "Implementation of an Algorithm for Peer-to-Peer Collaborative Editing". *Bachelor's project* (Spring 17).
- Rehan Mulakhel. "Web Interface for Secure Decentralized Collaboration Platform". *Bachelor's project* (Spring 17).
- Gaspard Zoss. "Enhancing Debian Update Service". *Master's project* (Fall 17).

Awards

- 2020: The Doc.Mobility Fellowship from the Swiss National Science Foundation
- 2015: EPFL EDIC Fellowship for Doctoral Studies
- 2013-2015: The Swedish Institute Scholarship
- 2009, 2011, 2012: Triple scholar of Vladimir Potanin Fellowship Program