sciendo

Proceedings on Privacy Enhancing Technologies ..; .. (..):1–28

Kirill Nikitin*, Ludovic Barman*, Wouter Lueks, Matthew Underwood, Jean-Pierre Hubaux, and Bryan Ford

Reducing Metadata Leakage from Encrypted Files and Communication with PURBs

Abstract: Most encrypted data formats leak metadata via their plaintext headers, such as format version, encryption schemes used, number of recipients who can decrypt the data, and even the recipients' identities. 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.

We propose that future encrypted data formats improve security and privacy hygiene by producing *Padded Uniform Random Blobs* or PURBs: ciphertexts indistinguishable from random bit strings to anyone without a decryption key. A PURB's content leaks *nothing at all*, even the application that created it, and is padded such that even its length leaks as little as possible.

Encoding and decoding ciphertexts with *no* cleartext markers presents efficiency challenges, however. We present cryptographically agile encodings enabling legitimate recipients to decrypt a PURB efficiently, even when encrypted for any number of recipients' public keys and/or passwords, and when these public keys are from different cryptographic suites. PURBs employ Padmé, a novel padding scheme that limits information leakage via ciphertexts of maximum length M to a practical optimum of $O(\log \log M)$ bits, comparable to padding to a power of two, but with lower overhead of at most 12% and decreasing with larger payloads.

Keywords: metadata, leakage, padding, traffic analysis

DOI Editor to enter DOI

Received ..; revised ..; accepted ...

***Corresponding Author: Kirill Nikitin:** EPFL, E-mail: kirill.nikitin@epfl.ch

***Corresponding Author: Ludovic Barman:** EPFL, E-mail: ludovic.barman@epfl.ch

Wouter Lueks: EPFL, E-mail: wouter.lueks@epfl.ch

Matthew Underwood: unaffiliated

Jean-Pierre Hubaux: EPFL, E-mail: jean-pierre.hubaux@epfl.ch

Bryan Ford: EPFL, E-mail: bryan.ford@epfl.ch

1 Introduction

Traditional encryption schemes and protocols aim to protect only their data payload, leaving related metadata exposed. Formats such as PGP [64] reveal in cleartext headers the public keys of the intended recipients, the algorithm used for encryption, and the actual length of the payload. Secure-communication protocols similarly leak information during key and algorithm agreement. The TLS handshake [44], for example, leaks in cleartext the protocol version, chosen cipher suite, and the public keys of the parties. This metadata exposure is traditionally assumed not to be security-sensitive, but important for the recipient's decryption efficiency.

Research has consistently shown, however, that attackers can exploit metadata to infer sensitive information about communication content. In particular, an attacker may be able to fingerprint users [39, 52] and the applications they use [63]. Using traffic analysis [16], an attacker may be able to infer websites a user visited [16, 20, 38, 56, 57] or videos a user watched [42, 43, 50]. On VoIP, metadata can be used to infer the geo-location [34], the spoken language [61], or the voice activity of users [14]. Side-channel leaks from data compression [31] facilitate several attacks on SSL [4, 24, 48]. The lack of proper padding might enable an active attacker to learn the length of the user's password from TLS [53] or QUIC [45] traffic. In social networks, metadata can be used to draw conclusions about users' actions [25], whereas telephone metadata has been shown to be sufficient for user re-identification and for determining home locations [35]. Furthermore, by observing the format of packets, oppressive regimes can infer which technology is used and use this information for the purposes of incrimination or censorship. Most TCP packets that Tor sends, for example, are 586 bytes due to its standard cell size [26].

As a step towards countering these privacy threats, we propose that encrypted data formats should produce *Padded Uniform Random Blobs* or PURBs: ciphertexts 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 while minimizing space overhead. 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. While simple in concept, because PURBs by definition contain *no* cleartext structure or markers, encoding and decoding them efficiently presents practical challenges.

This paper's first key contribution is *Multi-Suite PURB* or MsPURB, a cryptographically agile PURB encoding scheme that supports any number of recipients, who can use either shared passwords or public-private key pairs utilizing multiple cryptographic suites. The main technical challenge is providing *efficient* decryption to recipients without leaving any cleartext markers. If efficiency was of no concern, the sender could simply discard all metadata and expect the recipient to parse and trial-decrypt the payload using every possible format version, structure, and cipher suite. Real-world adoption requires both decryption efficiency and cryptographic agility, however. MsPURB combines a variable-length header containing encrypted metadata with a symmetrically-encrypted payload. The header's structure enables efficient decoding by legitimate recipients via a small number of trial decryptions. MsPURB facilitates the seamless addition and removal of supported cipher suites, while leaking no information to third parties without a decryption key. We construct our scheme starting with the standard construction of the Integrated Encryption Scheme (IES) [1] and use the ideas of multi-recipient public-key encryption [6, 33] as a part of the multi-recipient development.

To reduce information leakage from data lengths, this paper's second main contribution is Padmé, a padding scheme that groups encrypted PURBs into indistinguishability sets whose visible lengths are representable as limited-precision floating-point numbers. Like obvious alternatives such as padding to the next power of two, Padmé reduces maximum information leakage to $O(\log \log M)$ bits, where M is the maximum length of encrypted blob a user or application produces. Padmé greatly reduces constant-factor overhead with respect to obvious alternatives, however, enlarging files by at most +12%, and less as file size increases.

In our evaluation, creating a MsPURB ciphertext takes 235 ms for 100 recipients on consumer-grade hardware using 10 different cipher suites, and takes only 8 ms for the common single-recipient single-suite scenario. Our implementation is in pure Go without assembly optimizations that might speed up public-key operations. Because the MsPURB design limits the number of costly public-key operations, however, decoding performance is comparable to PGP, and is almost independent of the number of recipients (up to 10,000).

Analysis of real-world data sets show that many objects are trivially identifiable by their unique sizes without padding, or even after padding to a fixed block size (*e.g.,* that of a block cipher or a Tor cell). We show that Padmé can significantly reduce the number of objects uniquely identifiable by their sizes: from 83% to 3% for 56k Ubuntu packages, from 87% to 3% for 191k Youtube videos, from 45% to 8% for 848k hard-drive user files, and from 68% to 6% for 2.8k websites from the Alexa top 1M list. This much stronger leakage protection incurs an average space overhead of only 3%.

In summary, our main contributions are as follows:

- We introduce MsPURB, a novel encrypted data format that reveals no metadata information to observers without decryption keys, while efficiently supporting multiple recipients and cipher suites.
- We introduce Padmé, a padding scheme that asymptotically minimizes information leakage from data lengths while also limiting size overheads.
- We implement these encoding and padding schemes, evaluating the former's performance against PGP and the latter's efficiency on real-world data.

2 Motivation and Background

We first offer example scenarios in which PURBs may be useful, and summarize the Integrated Encryption Scheme that we later use as a design starting point.

2.1 Motivation and Applications

Our goal is to define a generic method applicable to most of the common data-encryption scenarios such that the techniques are flexible to the application type, to the cryptographic algorithms used, and to the number of participants involved. We also seek to enhance plausible deniability such that a user can deny that a PURB is created by a given application or that the user owns the key to decrypt it. We envision several immediate applications that could benefit from using PURBs.

E-mail Protection. E-mail systems traditionally use PGP or S/MIME for encryption. Their packet formats [13], 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, which does not provide "size privacy", as we show in §5.3. Using PURBs for encrypted e-mail could minimize this metadata leakage. Furthermore, as e-mail traffic is normally sparse, the moderate overhead PURBs incur can easily be accommodated.

Initiation of Cryptographic Protocols. In most cryptographic protocols, initial cipher suite negotiation, handshaking, and key exchange are normally performed unencrypted. In TLS 1.2 [19], an eavesdropper who monitors a connection from the start can learn many details such as cryptographic schemes used. The unencrypted Server Name Indication (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 [46] or distinguish censorship-circumvention tools that try to mimic TLS traffic [22, 28]. TLS 1.3 [44] takes a few protective measures: *e.g.,* less unencrypted metadata during the handshake, and an experimental extension for encrypted SNI [44, 47]. These measures are only partial, however, and leave other metadata, such as protocol version number, cipher suites, and public keys, still visible. PURBs could facilitate fully-encrypted handshaking from the start, provided a client already knows at least one public key and cipher suite 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 [27].

Encrypted Disk Volumes. VeraCrypt [29] 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 might 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.

2.2 Integrated Encryption Scheme

The Integrated Encryption Scheme (IES) [1] is a hybrid encryption scheme that enables the encryption of arbitrary message strings (unlike ElGamal, which requires the message to be a group element), and offers flexibility in underlying primitives. To send an encrypted message, a sender first generates an ephemeral Diffie-Hellman key pair and uses the public key of the recipient to derive a shared secret. The choice of the Diffie-Hellman group is flexible, *e.g.,* multiplicative groups of integers or elliptic curves. The sender then relies on a cryptographic hash function to derive the shared keys used to encrypt the message with a symmetric-key cipher and to compute a MAC using the encrypt-then-MAC approach. The resulting ciphertext is structured as shown in Figure 1.

pk_s	$\mathrm{enc}(M)$	σ_{mac}

Fig. 1. Ciphertext output of the Integrated Encryption Scheme where pk_s is an ephemeral public key of the sender, and σ_{mac} and $\mathsf{enc}(M)$ are generated using the DH-derived keys.

3 Hiding Encryption Metadata

This section addresses the challenges of encoding and decoding *Padded Uniform Random Blobs* or PURBs 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.

3.1 Preliminaries

Let λ be a standard security parameter. We use \$ to indicate randomness, $\xleftarrow{\$}$ to denote random sampling, $\|$ to denote string concatenation and $|\text{value}|$ to denote the bit-length of "value". We write PPT as an abbreviation for probabilistic polynomial-time. Let $\Pi = (\mathcal{E}, \mathcal{D})$ be an ind\$-cca2-secure authenticated-encryption (AE) scheme [7] where $\mathcal{E}_K(m)$ and $\mathcal{D}_K(c)$ are encryption and decryption algorithms, respectively, given a message m, a ciphertext c, and a key K. Let $\mathsf{MAC} = (\mathcal{M}, \mathcal{V})$

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 $\texttt{Hide}: \mathbb{G}(1^\lambda) \rightarrow \{0,1\}^\lambda$ 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 [9], Elligator Squared [2, 51]). Let $\texttt{Unhide}: \{0,1\}^\lambda \rightarrow \mathbb{G}(1^\lambda)$ be the counterpart to Hide that decodes a binary string into a group element of $\mathbb{G}$.

Let $\mathrm{H} : \mathbb{G} \rightarrow \{0,1\}^{2\lambda}$ and $\hat{\mathrm{H}} : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda}$ be two distinct cryptographic hash functions. Let $\texttt{PBKDF} : \{salt, password\} \rightarrow \{0,1\}^{2\lambda}$ be a secure password-based key-derivation function [10, 32, 40], 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.

3.1.1 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 [19]. Replacing any component of a suite (*e.g.,* the group) results in a different cipher suite.

3.1.2 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 latter 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 [3] 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 [5] where an adversary is unable to determine whether a given public key has been used for a given encryption.

3.1.3 System Goals

We wish to achieve two system goals beyond security:

- PURBs must provide cryptographic agility. They should accommodate either one or multiple recipients, allow encryption for each recipient using a shared password or a public key, and support different cipher suites. Adding 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, and padding must not impose excessive space overhead.

3.2 Encryption to a Single Passphrase

We begin with a simple strawman PURB encoding format allowing a sender to encrypt data using a single long-term passphrase $\hat{S}$ 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. The sender first generates a fresh symmetric session key K and computes the PURB payload as $\mathcal{E}_K(\text{data})$. The sender then generates a random salt and derives the ephemeral secret $S = \texttt{PBKDF}(\text{salt}, \hat{S})$. The sender next creates an *entry point* (EP) containing the session key K, the position of the payload and potentially other metadata. The sender then encrypts the EP using S. Finally, the sender concatenates the three segments to form the PURB as shown in Figure 2.

salt	$\mathcal{E}_S(K \parallel \text{meta})$	$\mathcal{E}_K(\text{data})$
	entry point	payload

Fig. 2. A PURB addressed to a single recipient and encrypted with a passphrase-derived ephemeral secret S.

3.3 Single Public Key, Single Suite

We often prefer to use public-key cryptography, instead of pre-shared secrets, to establish secure communication or encrypt data at rest. Typically the sender or initiator indicates in the file's cleartext metadata which public key this file is encrypted for (*e.g.,* in PGP), or else parties exchange public-key certificates in cleartext during communication setup (*e.g.,* in TLS). Both approaches generally leak the receiver's identity. We address this use case with a second strawman PURB encoding format that builds on the last by enabling the decryption of an entry point EP using a private key.

To expand our scheme to the public-key scenario, we adopt the idea of a hybrid asymmetric-symmetric scheme from the IES (see §2.2). 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 it 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. The resulting PURB structure is shown in Figure 3.

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

Fig. 3. 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.

3.4 Multiple Public Keys, Single Suite

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

As the first step, we adopt the idea of multi-recipient public-key encryption [6, 33] where the sender generates a single key pair and uses it to derive an ephemeral secret with each of the intended recipients. The sender creates one entry point per recipient. These entry points contain the same session key and metadata but are encrypted with different ephemeral secrets.

As a PURB's purpose is to prevent metadata leakage, including the number of 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 primary challenge is to find a space-efficient layout of entry points—with no cleartext markers—such that the recipients are able to find their segments efficiently.

Linear Table. The most space-efficient approach is to place entry points sequentially. In fact, OpenPGP suggests a similar approach for achieving better privacy [13, Section 5.1]. However, in this case, decryption is inefficient: the recipients have to attempt sequentially to decrypt each potential entry point, before finding their own or reaching the end of the PURB.

Fixed Hash Tables. A more computationally-efficient approach is to use a hash table of a fixed size. The sender creates a hash table and places 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 bit strings, 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. This approach also has the downside of imposing an artificial limit on the number of recipients.

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. 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.

Definitions. We now formalize this scheme. Let r be the number of recipients and $(y_1, Y_1), \dots, (y_r, Y_r)$ 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 $\mathcal{E}_{Z_1}(K, \text{meta}), ..., \mathcal{E}_{Z_r}(K, \text{meta})$. The position of an entry in a hash table j is $(P_i \mod 2^j)$. The sender iteratively tries 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 for five recipients is illustrated in Figure 4.

encoded pk	HT0	HT1	HT2	payload
$\mathtt{Hide}(X)$	$\mathcal{E}_{Z_1}(K)$	$\mathcal{E}_{Z_3}(K)$	$\mathcal{E}_{Z_4}(K)$	$\mathcal{E}_K(\text{data})$
		$\mathcal{E}_{Z_2}(K)$	random	
			$\mathcal{E}_{Z_5}(K)$	
			random	

Fig. 4. A PURB with hash tables of increasing sizes (HT0, HT1, HT2). Five and two slots of the hash tables are filled with entry points and random bit strings respectively. The metadata “meta” in the entry points is omitted from the figure. Hash-table entries are put one after another in the byte representation of a PURB.

To decode, a recipient reads the public key; derives the ephemeral secret k_i, the encryption key Z_i and the position key P_i; and iteratively tries matching positions in hash tables until the decryption of the entry point succeeds. Although the recipient does not initially know the number of hash tables in a PURB, the 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. With r recipients, the worst-case compactness is having r hash tables (if each insertion leads to a collision), which happens with exponentially decreasing probability. The expected number of trial decryptions is $\log_2 r$.

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. Further, we must be able to introduce new cipher suites gradually, often requiring larger and differently-structured keys and ciphertexts, while preserving interoperability and compatibility with old cipher suites. We therefore build on the above 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 positions where the suite's public key could be in the PURB. For instance, let us consider a 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 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 Appendix B.

encoded pk$_A$	HT0	HT1	HT2	payload
Hide(X_A)	rnd	Hide(X_B)	$\mathcal{E}_{Z_2}(K)$	$\mathcal{E}_K$(data)
		$\mathcal{E}_{Z_1}(K)$	random	
			$\mathcal{E}_{Z_3}(K)$	
			random	

Fig. 5. Example of a PURB encoded for three public keys in two suites (suite 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 (since 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 challenge is that suites might indicate different lengths for both their public keys and entry points. An encoder can easily accommodate 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, an encoder 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.

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 [41], 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 $K_{enc} = \hat{\text{H}}(\text{“enc”} \parallel K)$ and MAC $K_{mac} = \hat{\text{H}}(\text{“mac”} \parallel K)$ 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. To handle this situation, the sender is responsible for ensuring that the authentication tag does not fall into any of the possible public key positions for the cipher suites in use.

To encode a PURB, a sender prepares entry points, lays out the header, encrypts the payload, adds padding (see §4), 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 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.

3.7 Complete Algorithms

We summarize the encoding scheme by giving detailed algorithms. We begin by defining 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 order p generated by g; Hide is a mapping: $\mathbb{G} \to \{0,1\}^{\lambda}$; $\Pi = (\mathcal{E}, \mathcal{D})$ 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. 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. We then define the full HdrPURB and MsPURB algorithms as follows:

<u>Algorithms HdrPURB</u>.

$\mathsf{HdrPURB.Encap}(R, S) \to (\tau, k_1, \dots, k_r)$: Given a set of public keys $R = \{pk_1 = Y_1, \dots, pk_r = Y_r\}$ of a suite S:
1. Pick a fresh $x \in \mathbb{Z}_p$ and compute $X = g^x$ where p, g are defined in S.
2. Derive $k_1 = \mathrm{H}(Y_1^x), \dots, k_r = \mathrm{H}(Y_r^x)$.
3. Map X to a uniform string $\tau_X = \texttt{Hide}(X)$.
4. Output an encoded public key $\tau = \tau_X$ and $k_1, \dots, k_r$.

$\mathsf{HdrPURB.Decap}(sk(S), \tau) \to k$: Given a private key $sk = y$ of a suite S and an encoded public key τ:
1. Retrieve $X = \texttt{Unhide}(\tau)$.
2. Compute and output $k = \mathrm{H}(X^y)$.

<u>Algorithms MsPURB</u>.

$\mathsf{MsPURB.Setup}(1^\lambda) \to S$: Initialize a cipher suite $S = \langle \mathbb{G}, p, g, \texttt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$.

$\mathsf{MsPURB.KeyGen}(S) \to (sk, pk)$: Given a suite $S = \langle \mathbb{G}, p, g, \dots \rangle$, pick $x \in \mathbb{Z}_p$ and compute $X = g^x$. Output $(sk = x, pk = X)$.

$\mathsf{MsPURB.Enc}(R, m) \to c$: Given a set of public keys of an indicated suite $R = \{pk_1(S_1), \dots, pk_r(S_r)\}$ and a message m:
1. Pick an appropriate symmetric-key encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ with key length λ_K, a MAC algorithm $\mathsf{MAC} = (\mathcal{M}, \mathcal{V})$, and a hash function $\mathrm{H}' : \{0,1\}^* \to \{0,1\}^{\lambda_K}$ such that the key length λ_K matches the security level of the most conservative suite.
2. Group R into $R_1, \dots, R_n$, s.t. all public keys in a group R_i share the same suite S_i. Let $r_i = |R_i|$.
3. For each R_i:
   (a) Run $(\tau_i, k_1, \dots, k_{r_i}) = \mathsf{HdrPURB.Encap}(R_i, S_i)$;
   (b) Compute entry-point keys $\mathsf{keys}_i = (Z_1 = \hat{\mathrm{H}}(\text{“key”} \parallel k_1), \dots, Z_{r_i} = \hat{\mathrm{H}}(\text{“key”} \parallel k_{r_i}))$ and positions $\mathsf{aux}_i = (P_1 = \hat{\mathrm{H}}(\text{“pos”} \parallel k_1), \dots, P_{r_i} = \hat{\mathrm{H}}(\text{“pos”} \parallel k_{r_i}))$.
4. Pick $K \xleftarrow{\$} \{0,1\}^{\lambda_K}$.
5. Record $(\mathsf{Enc}, \mathsf{Dec})$, MAC and H' in meta.
6. Compute a payload key $K_{enc} = \mathrm{H}'(\text{“enc”} \parallel K)$ and a MAC key $K_{mac} = \mathrm{H}'(\text{“mac”} \parallel K)$.
7. Obtain $c_{\mathsf{payload}} = \mathsf{Enc}_{K_{enc}}(m)$.
8. Run $c' \leftarrow \textsc{Layout}(\tau_1, \dots, \tau_n, \mathsf{keys}_1, \dots, \mathsf{keys}_n, \mathsf{aux}_1, \dots, \mathsf{aux}_n, S_1, \dots, S_n, K, \mathsf{meta}, c_{\mathsf{payload}})$ (see Algorithm 2 on page 21).
9. Derive an authentication tag $\sigma = \mathcal{M}_{K_{mac}}(c')$ and output $c = c' \parallel \sigma$.

$\mathsf{MsPURB.Dec}(sk(S), c) \to m/\perp$: Given a private key sk of a suite S and a ciphertext c:

(1) Look up the possible positions of a public key defined by S and XOR bytes at all the positions to obtain the encoded public key τ.
(2) Run $k \leftarrow \mathsf{HdrPURB.Decap}(sk, \tau)$.
(3) Derive $Z = \hat{\mathrm{H}}(\text{“key”} \parallel k)$ and $P = \hat{\mathrm{H}}(\text{“pos”} \parallel k)$.
(4) Parse c as growing 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 $\mathsf{MAC} = (\mathcal{M}, \mathcal{V})$ algorithm and the length of MAC output 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 $K_{enc} = \mathrm{H}'(\text{“enc”} \parallel K)$, read the start and the end of the payload from meta (it is written by Layout) to parse c' as $\langle hdr \parallel c_{\mathsf{payload}} \parallel \text{padding} \rangle$, and return $\mathsf{Dec}_{K_{enc}}(c_{\mathsf{payload}})$ where Dec is the payload decryption algorithm specified in meta.

Theorem 1. If for each cipher suite $S = \langle \mathbb{G}, p, g, \mathtt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$ used in a PURB we have that: the gap-CDH problem is hard relative to $\mathbb{G}$, $\mathtt{Hide}$ maps group elements in $\mathbb{G}$ to uniform random strings, Π is ind\$-cca2-secure, and 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 $(\mathsf{Enc}, \mathsf{Dec})$ is ind\$-cpa-secure, then MsPURB is ind\$-cca2-secure against an outsider adversary.

Proof. See Appendix D.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 $S = \langle \mathbb{G}, p, g, \mathtt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$, used in a PURB we have that: the gap-CDH problem is hard relative to $\mathbb{G}$, $\mathtt{Hide}$ 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 D.3. □

3.8 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 suites from a known, public, ordered list. To add a new 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 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 [8], 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. However, generalizing this approach of a global nonce to any scheme requires further analysis.

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 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. [3]. This approach, however, requires 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.

Limitations. The MsPURB scheme above is not secure against quantum computers, as it relies on discrete logarithm hardness. It is theoretically possible to substitute IES-based key encapsulation with a quantum-resistant variant to achieve quantum ind\$-cca2 security. The requirements for substitution are 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, they do 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 partially address this with padding in §4. However, no reasonable padding scheme can 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 $\perp$.

4 Limiting Leakage via Length

The encoding scheme presented above in §3 produces blobs of data that are indistinguishable from random bit-strings of the same length, thus leaking no information to the adversary directly via their content. The length itself, however, might indirectly reveal information about the content. Such leakage is already used extensively in traffic-analysis attacks, *e.g.,* website fingerprinting [20, 38, 56, 57], video identification [42, 43, 50], and VoIP traffic fingerprinting [14, 61]. Although solutions involving application- or network-level padding are numerous, they are typically designed for a specific problem domain, and the more basic problem of length-leaking ciphertexts remains. In any practical solution, some leakage is unavoidable. We show, however, that typical approaches such as padding to the size of a block cipher are fundamentally insufficient for efficiently hiding the plaintext length effectively, especially for plaintexts that may vary in size by orders of magnitude.

We introduce Padmé, a novel padding scheme designed for, though not restricted to, encoding PURBs. Padmé reduces length leakage for a wide range of encrypted data types, ensuring asymptotically lower leakage of $O(\log\log M)$, rather than $O(\log M)$ for common stream- and block-cipher-encrypted data. Padmé's space overhead is moderate, always less than 12% and decreasing with file size. The intuition behind Padmé is to pad objects to lengths representable as limited-precision floating-point numbers. A Padmé length is constrained in particular to have no more significant bits (*i.e.,* information) in its mantissa than in its exponent. This constraint limits information leakage to at most double that of conservatively padding to the next power of two, while reducing overhead through logarithmically-increasing precision for larger objects.

Many defenses already exist for specific scenarios, *e.g.,* against website fingerprinting [20, 58]. Padmé does not attempt to compete with tailored solutions in their domains. Instead, Padmé aims for a substantial increase in application-independent length leakage protection as a generic measure of security/privacy hygiene.

4.1 Design Criterion

We design Padmé again using intermediate strawman approaches for clarity. To compare these straightforward alternatives with our proposal, we define a game where an adversary guesses the plaintext behind a padded encrypted blob. This game is inspired by related work such as defending against a *perfect attacker* [58].

Padding Game. Let P denote a collection of plaintext objects of maximum length M: *e.g.,* data, documents, or application data units. An honest user chooses a plaintext $p \in P$, then pads and encodes it into a PURB c. The adversary knows almost everything: all possible plaintexts P, the PURB c and the parameters used to generate it, such as schemes and number of recipients. The adversary lacks only the private inputs and decryption keys for c. The adversary's goal is to guess the plaintext p based on the observed PURB c of length $|c|$.

Design Goals. Our goal in designing the padding function is to manage both space overhead from padding and maximum information leaked to the adversary.

4.2 Definitions

Overhead. Let c be a padded ciphertext resulting from PURB-encoding plaintext p. For simplicity we focus here purely on overhead incurred by padding, by assuming an unrealistic, "perfectly-efficient" PURB encoding that (unlike MsPURB) incurs no space overhead for encryption metadata. We define the *additive overhead* of $|c|$ over $|p|$ to be $|c|-|p|$, the number of extra bytes added by padding. The *multiplicative overhead* of padding is $\frac{|c|-|p|}{|p|}$, the relative fraction by which $|c|$ expands $|p|$.

Leakage. Let P be a finite space of plaintexts of maximum length M. Let $f : \mathbb{N} \to \mathbb{N}$ be a padding function that yields the padded size $|c|$ given a plaintext length

$|p|$, for $p \in P$. The image of f is a set R of padded lengths that f can produce from plaintexts $p \in P$.
We quantify the leakage of padding function f in terms of the number of elements in R. More precisely, we define the leakage as the number of bits (amount of information entropy) required to distinguish a unique element of R, which is $\lceil \log_2 |R| \rceil$. Intuitively, a function that pads everything to a constant size larger than all plaintexts (*e.g.,* $f(p) = 1$ Tb) leaks no information to the adversary, because $|R| = 1$ (and observing $|c| = 1$ Tb leaks no information about the plaintext), whereas more fine-grained padding functions leak more bits.

4.3 Strawman Padding Approaches

We first explore two strawman designs, based on different padding functions f. A padding function that offers any useful protection cannot be one-to-one, otherwise the adversary could trivially invert it and recover $|p|$. We also exclude randomized padding schemes for simplicity, and because in practice adversaries can typically cancel out and defeat random padding factors statistically over many observations. Therefore, only padding functions that group many plaintext lengths into fewer padded ciphertexts are of interest in our analysis.

Strawman 1: Fixed-Size Blocks. We first consider a padding function $f(L) = b \cdot \lceil L/b \rceil$, where b is a block size in bytes. This is how objects often get "padded" by default in practice, *e.g.,* in block ciphers or Tor cells. In this case, the PURB's size is a multiple of b, the maximum additive overhead incurred is $b-1$ bytes, and the leakage is $\lceil \log_2 \ M/b \rceil = O(\log M)$, where M is the maximum plaintext size.

In practice, when plaintext sizes differ by orders of magnitude, there is no good value for b that serves all plaintexts well. For instance, consider $b = 1\,\text{MB}$. Padding small files and network messages would incur a large overhead: *e.g.,* padding Tor's 512 B cells to 1 MB would incur overheads of 2000×. In contrast, padding a 700 MB movie with at most 1 MB of chaff would add only a little confusion to the adversary, as this movie may still be readily distinguishable from others by length. To reduce information leakage asymptotically over a vast range of cleartext sizes, therefore, padding must depend on plaintext size.

Strawman 2: Padding to Powers of 2. The next step is to pad to varying-size blocks, which is the basis for our actual scheme. The intuition is that for small plaintexts, the blocks are small too, yielding modest overhead, whereas for larger files, blocks are larger and group more plaintext lengths together, improving leakage asymptotically. A simple approach is to pad plaintexts into buckets b_i of size varying as a power of some base, *e.g.,* two, so $b_i = 2^i$. The padding function is thus $f(L) = 2^{\lceil \log_2 L \rceil}$. We call this strawman NextP2.

Because NextP2 pads plaintexts of maximum length M into at most $\lceil \log_2 M \rceil$ buckets, the image R of f contains only $O(\log M)$ elements. This represents only $O(\log \log M)$ bits of entropy or information leakage, a major asymptotic improvement over fixed-size blocks.

The maximum overhead is substantial, however, almost +100%: *e.g.,* a 17 GB Blu-Ray movie would be padded into 32 GB. Using powers of another base $x > 2$, we reduce leakage further at a cost of more overhead: *e.g.,* padding to the nearest power of 3 incurs overhead up to +200%, with less leakage but still $O(\log \log M)$. We could reduce overhead by using a fractional base $1 < x < 2$, but fractional exponents are cumbersome in practical padding functions we would prefer to be simple and operate only on integers. Although this second strawman succeeds in achieving asymptotically lower leakage than padding to fixed-size blocks, it is less attractive in practice due to high overhead when $x \geq 2$ and due to computation complexity when $1 < x < 2$.

4.4 Padmé

We now describe our padding scheme Padmé, which limits information leakage about the length of the plaintext for wide range of encrypted data sizes. Similarly to the previous strawman, Padmé also asymptotically leaks $O(\log \log M)$ bits of information, but its overhead is much lower (at most 12% and decreasing with L).

Intuition. In NextP2, any permissible padded length L has the form $L = 2^n$. We can therefore represent L as a binary floating-point number with a $\lfloor \log_2 n \rfloor + 1$-bit exponent and a mantissa of zero, *i.e.,* no fractional bits.

In Padmé, we similarly represent a permissible padded length as a binary floating-point number, but we allow a non-zero mantissa at most as long as the exponent (see Figure 6). This approach doubles the number of bits used to represent an allowed padded length – hence doubling absolute leakage via length – but allows for more fine-grained buckets, reducing overhead. Padmé asymptotically leaks the same number of bits as NextP2, differing only by a constant factor of 2, but reduces space overhead by almost 10× (from +100% to +12%). More importantly, the multiplicative expansion overhead decreases with L (see Figure 7).

$\lfloor \log_2 n \rfloor + 1$-bit exponent	0-bit mantissa

In the strawman NEXTP2, the allowed length $L = 2^n$ can be represented as a binary floating-point number with a $\lfloor \log(n) + 1 \rfloor$ bits of exponent and no mantissa.

$\lfloor \log_2 n \rfloor + 1$-bit exponent	$\lfloor \log_2 n \rfloor + 1$-bit mantissa

Fig. 6. PADMÉ represents lengths as floating-point numbers, allowing the mantissa to be of at most $\lfloor \log_2 n \rfloor + 1$ bits.

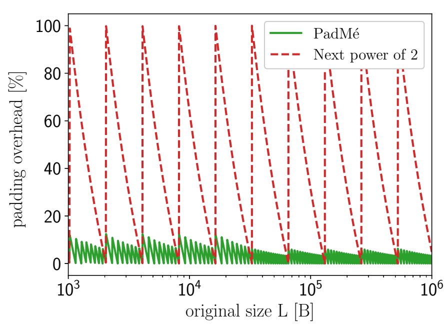


Fig. 7. Maximum multiplicative expansion overhead with respect to the plaintext size L. The naïve approach to pad to the next power of two has a constant maximum overhead of 100%, whereas PADMÉ's maximum overhead decreases with L, following $\frac{1}{2 \log_2 L}$.

Algorithm. To compute the padded size $L' = f(L)$, ensuring that its floating-point representation fits in at most $2 \times \lfloor \log_2 n \rfloor + 1$ bits, we require the last $E - S$ bits of L' to be 0. $E = \lfloor \log_2 L \rfloor$ is the value of the exponent, and $S = \lfloor \log_2 E \rfloor + 1$ is the size of the exponent's binary representation. The reason for the substraction will become clear later. For now, we demonstrate how E and S are computed in Table 1.

Recall that PADMÉ requires the mantissa's bit length to be no longer than that of the exponent. In Table 1, for the value $L = 9$ the mantissa is longer than the exponent: it is "too precise" and therefore not a permitted padded length. The value 10 is permitted, however, so a 9 byte-long ciphertext is padded to 10 bytes.

Table 1. The IEEE floating-point representations of 8, 9 and 10. The value 8 has 1 bit of mantissa (the initial 1 is omitted), and 2 bits of exponents; 9 has a 3-bits mantissa and a 2-bit exponent, while the value 10 as 2 bits of mantissa and exponents. PADMÉ enforces the mantissa to be no longer than the exponent, hence 9 gets rounded up to the next permitted length 10.

L	L	E	S	IEEE representation
8	0b**1000**	3	2	0b**1.0** * 2^0b**11**
9	0b**1001**	3	2	0b**1.001** * 2^0b**11**
10	0b**1010**	3	2	0b**1.01** * 2^0b**11**

To understand why PADMÉ requires the low $E - S$ bits to be 0, notice that forcing all the last E bits to 0 is equivalent to padding to a power of two. In comparison, PADMÉ allows S extra bits to represent the padded size, with S defined as the bit length of the exponent.

Algorithm 1 specifies the PADMÉ function precisely.

Algorithm 1: PADMÉ

Data: length of content L
Result: length of padded content L'
$E \leftarrow \lfloor \log_2 L \rfloor$ `// L's floating-point exponent`
$S \leftarrow \lfloor \log_2 E \rfloor + 1$ `// # of bits to represent E`
$z \leftarrow E - S$ `// # of low bits to set to 0`
$m \leftarrow (1 \ll z) - 1$ `// mask of z 1's in LSB`
`// round up using mask m to clear last z bits`
$L' \leftarrow (L + m)$ & $\sim m$

Leakage and Overhead. By design, if the maximum plaintext size is M, PADMÉ's leakage is $O(\log \log M)$ bits, the length of the binary representation of the largest plaintext. As we fix $E-S$ bits to 0 and round up, the maximum overhead is $2^{E-S} - 1$. We can estimate the maximum multiplicative overhead as follows:

$$\begin{aligned} \text{max overhead} &= \frac{2^{E-S} - 1}{L} < \frac{2^{E-S}}{L} \\ &\approx \frac{2^{\lfloor \log_2 L \rfloor - \lfloor \log_2 \log_2 L \rfloor - 1}}{L} \\ &\approx \frac{1}{2 \cdot 2^{\log_2 \log_2 L}} \\ &= \frac{1}{2 \log_2 L} \end{aligned} \tag{1}$$

Thus, PADMÉ's maximum multiplicative overhead decreases with respect to the file size L. The maximum overhead is $+11.\overline{11}\%$, when padding a 9-byte file into 10 bytes. For bigger files, the overhead is smaller.

On Optimality. There is no clear sweet spot on the leakage-to-overhead curve. We could easily force the last $\frac{1}{2}(E - S)$ bits to be 0 instead of the last $E - S$ bits, for example, to reduce overhead and increase leakage. Still, what matters in practice is the relationship between L and the overhead. We show in §5.3 how this choice performs with various real-world datasets.

5 Evaluation

Our evaluation is two-fold. First, we show the performance and overhead of the PURB encoding and decoding. Second, using several datasets, we show how Padmé facilitates hiding information about data length.

5.1 Implementation

We implemented a prototype of the PURB encoding and padding schemes in Go. The implementation follows the algorithms in §3.7, and it consists of 2 kLOC. Our implementation relies on the open-source Kyber library[1] 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 Padmé, as well as scripts for reproducing all experiments, are available in the main repository[2].

5.2 Performance of the PURB Encoding

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.

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 [13, 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.).

5.2.1 Methodology

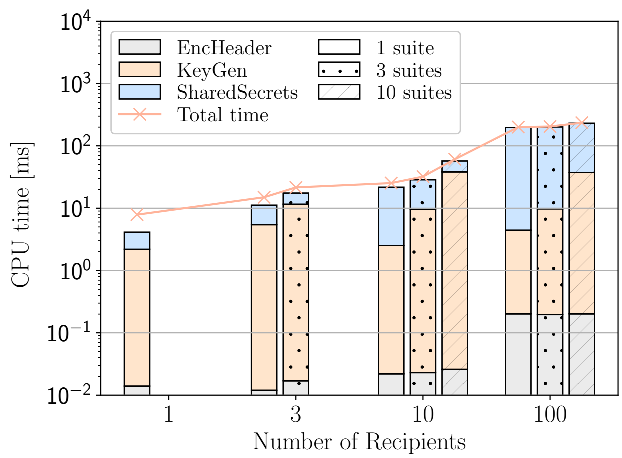
We ran the encoding experiments on a consumer-grade laptop, with a quad-core 2.2 GHz Intel Core i7 processor and 16 GB of RAM, using Go 1.12.5. To compare with an OpenPGP implementation, we use and modify Keybase's fork[3] of the default Golang crypto library[4], as the fork adds support for the ECDH scheme on Curve25519.

We further modify Keybase's implementation to add the 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 §3.8. 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.

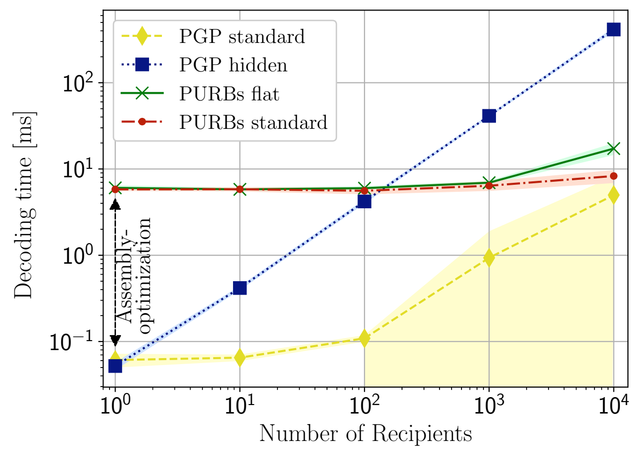
5.2.2 Results

Encoding Performance. In this experiment, we first evaluate how the time required to encode a PURB changes with a growing number of recipients and cipher suites, and second, how the main computational components contribute to this duration. We divide the total encoding time into three components. The first is authenticated encryption of entry points. The second is the generation and Elligator encoding of sender's public keys, one per suite. 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

1 https://github.com/dedis/kyber
2 https://github.com/dedis/purb
3 https://github.com/keybase/go-crypto
4 https://github.com/golang/crypto



(a) 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.



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

Fig. 8. Performance of the PURBs encoding.

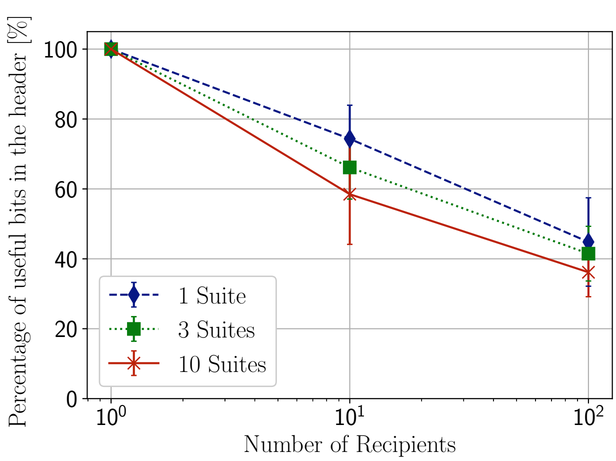


Fig. 9. Compactness of the PURB header (% of non-random bits).

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 cases using one, three or ten cipher suites. When more than one cipher suite is used, the recipients are equally divided among them.

Figure 8a 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. 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 8b 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, while 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 multiplication regardless of the 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. Compared with placing the header elements linearly, our expanding hash table design is less compact, but enables more efficient decoding. Figure 8b shows an example of this trade-off, PGP hidden versus PURBs standard.

In Figure 9, 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: 45% for 100 recipients and 1 suite, 36% for 100 recipients and 10 suites. In the most common case of having one recipient in one suite, however, the header is perfectly compact. Finally, there is a trade-off between compactness and efficient decryption. We can easily increase compactness by resolving entry point hash table collisions linearly, instead of directly moving to the next hash table. The downside is that the recipient has more entry points to try.

5.3 Performance of Padmé Padding

In evaluating a padding scheme, one important metric is overhead incurred in terms of bits added to the plaintexts. By design, PADMÉ's overhead is bounded by $\frac{1}{2 \cdot \log_2 L}$. As discussed in §4.4, PADMÉ does not escape the typical overhead-to-leakage trade-off, hence PADMÉ's novelty does not lie in this tradeoff. Rather, the novelty lies in the practical relation between L and the overhead. PADMÉ's overhead is moderate, at most +12% and much less for large PURBs.

A more interesting question is how effectively, given an arbitrary collection of plaintexts P, PADMÉ hides which plaintext is padded. PADMÉ was designed to work with an arbitrary collection of plaintexts P. It remains to be seen how PADMÉ performs when applied to a specific set of plaintexts P, *i.e.,* with a distribution coming from the real world, and to establish how well it groups files into sets of identical length. In the next section, we experiment with four datasets made of various objects: a collection of Ubuntu packages, a set of YouTube videos, a set of user files, and a set of Alexa Top 1M websites.

5.3.1 Datasets and Methodology

The Ubuntu dataset contains 56,517 unique packages, parsed from the official repository of a live Ubuntu 16.04 instance. As packages can be referenced in multiple repositories, we filtered the list by name and architecture. The reason for padding Ubuntu software updates is that the knowledge of updates enables a local eavesdropper to build a list of packages and their versions that are installed on a machine. If some of the packages are outdated and have known vulnerabilities, an adversary might use it as an attack vector. A percentage of software updates still occurs over un-encrypted connections, which is still an issue; but encrypted connections to software-update repositories also expose which distribution and the kind of update being done (security / restricted[5] / multiverse[6] / etc). We hope that this unnecessary leakage will disappear in the near future.

Table 2. Datasets used in the evaluation of anonymity provided by PADMÉ.

Dataset	# of objects
Ubuntu packages	56,517
YouTube videos	191,250
File collections	3,027,460
Alexa top 1M Websites	2,627

The YouTube dataset contains 191,250 unique videos, obtained by iteratively querying the YouTube API. One semantic video is generally represented by 2 – 5 .webm files, which corresponds to various video qualities. Hence, each object in the dataset is a unique (video, quality) pair. We use this dataset as if the videos were downloaded in bulk rather than streamed; that is, we pad the video as a single file. The argument for padding YouTube videos as whole files is that, as shown by related work [42, 43, 50], variable-bitrate encoding combined with streaming leak which video is being watched. If YouTube wanted to protect the privacy of its users, it could re-encode everything to constant-bitrate encoding and still stream it, but then the total length of the stream would still leak information. Alternatively, it could adopt a model similar to that of the iTunes store, where videos have variable bit-rate but are bulk-downloaded; but again, the total downloaded length would leak information, requiring some padding. Hence, we explore how unique the YouTube videos are by length with and without padding.

The files dataset was constituted by collecting the file sizes in the home directories (‘`~user/`’) of 10 co-workers and contains 3,027,460 of both personal files and configuration files. These files were collected on machines running Fedora, Arch, and Mac OS X. The argument for analyzing the uniqueness of those files is not to encrypt each file individually – there is no point in hiding the metadata of a file if the file's location exposes

5 Contains proprietary software and drivers.

6 Contains software restricted by copyright.

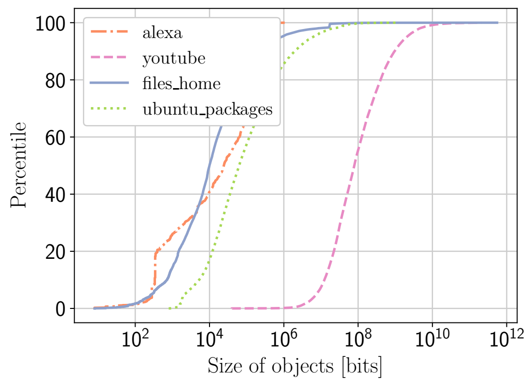


Fig. 10. Distribution of the sizes of the objects in each dataset.

everything about it, e.g. '~user/.ssh' – but rather to quantify the privacy gain when padding those objects.

Finally, the Alexa dataset is made of 2,627 websites from the Alexa Top 1M list. The size of each website is the sum of all the resources loaded by the webpage, which has been recorded by piloting a 'chrome-headless' instance with a script, mimicking real browsing. One reason for padding whole websites – as opposed to padding individual resources – is that related work in website fingerprinting showed the importance of the total downloaded size [20]. The effectiveness of PADMÉ when padding individual resources, or for instance bursts [58], is left as interesting future work.

5.3.2 Evaluation of Padmé

The distribution of the objects sizes for all the datasets is shown in Figure 10. Intuitively, it is harder for an efficient padding scheme to build groups of same-sized files when there are large objects in the dataset. Therefore, we expect the last 5% to 10% of the four datasets to remain somewhat unique, even after padding.

For each dataset, we analyze the anonymity set size of each object. To compute this metric, we group objects by their size, and report the distribution of the sizes of these groups. A large number of small groups indicate that many objects are easily identifiable. For each dataset, we compare three different approaches: the NEXTP2 strawman, PADMÉ, and padding to a fixed block size of 512B, like a Tor cell. The anonymity metrics are shown in Figure 11, and the respective overheads are shown in Table 3.

For all these datasets, despite containing very different objects, a large percentage of objects have a unique size: 87% in the case of YouTube video (Figure 11a), 45% in the case of files (Figure 11b), 83% in the case

Table 3. Analysis of the overhead, in percentage, of various padding approaches. In the first column, we use $b = 512B$ as block size.

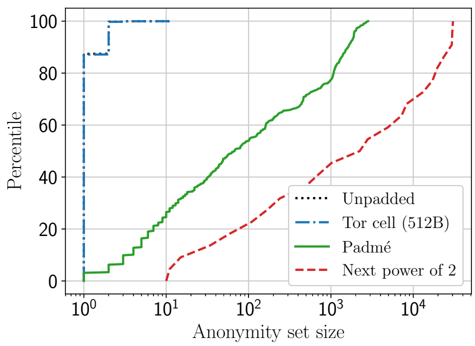
Dataset	Fixed block size	Next power of 2	Padmé
YouTube	0.01	44.12	2.23
files	40.15	44.18	3.64
Ubuntu	14.09	43.21	3.12
Alexa	36.71	47.12	3.07

of Ubuntu packages (Figure 11c), and 68% in the case of Websites Figure 11d). These characteristics persist in traditional block-cipher encryption (blue dashed curves) where objects are padded only to a block size. Even after being padded to 512 bytes, the size of a Tor cell, most object sizes remain as unique as in the unpadded case. We observe similar results when padding to 256 bits, the typical block size for AES (not plotted).

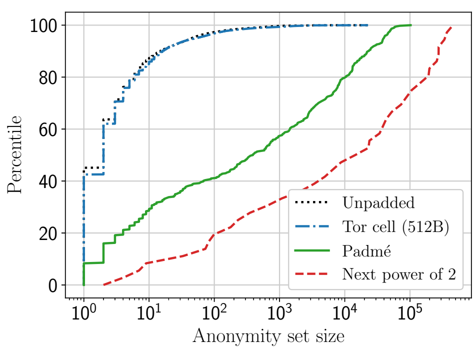
NEXTP2 (red dotted curves) provides the best anonymity: in the YouTube and Ubuntu datasets (Figures 11a and 11c), there is no single object that remains unique with respect to its size; all belong to groups of at least 10 objects. We cannot generalize this statement, of course, as shown by the other two datasets (Figures 11b and 11d). In general, we see a massive improvement with respect to the unpadded case. Recall that this padding scheme is impractically costly, adding +100% to the size in the worst case and +50% in mean. In Table 3, we see that the mean overhead is of +45%.

Finally, we see the anonymity provided by PADMÉ (green solid curves). By design, PADMÉ has an acceptable maximum overhead (maximum +12% and decreasing). In three of the four datasets, there is a constant difference between our expensive reference point NEXTP2 and PADMÉ; despite having a decreasing overhead with respect to L, unlike NEXTP2. This means that although larger files have proportionally less protection (*i.e.,* less padding in percentage) with PADMÉ, this is not critical, as these files are more rare and are harder to protect efficiently, even with a naïve and costly approach. When we observe the percentage of uniquely identifiable objects (objects that trivially reveal their plaintext given our perfect adversary), we see a significant drop by using PADMÉ: from 83% to 3% for the Ubuntu dataset, from 87% to 3% for the Youtube dataset, from 45% to 8% for the files dataset and from 68% to 6% for the Alexa dataset. In Table 3, we see that the mean overhead of PADMÉ is around 3%, more than an order of magnitude smaller than NEXTP2. We also see how using a fixed block size can yield high overhead in percentage, in addition to insufficient protection.

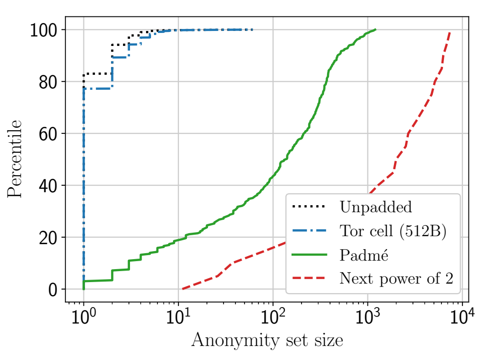
(a) Dataset ‘YouTube’:



(b) Dataset ‘files’:



(c) Dataset ‘Ubuntu’:



(d) Dataset ‘Alexa’:

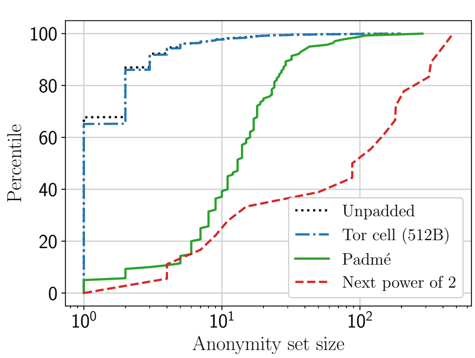


Fig. 11. Analysis of the anonymity provided by various padding approaches: NEXTP2, PADMÉ, padding with a constant block size and no padding. We measure for each object with how many other objects it becomes indistinguishable after being padded, and plot the distribution. NEXTP2 provides better anonymity, at the cost of a drastically higher overhead (at most +100% instead of +12%). Overheads are shown in Table 3.

6 Related Work

The closest related work PURBs build on is Broadcast Encryption [3, 12, 18, 21, 23], which formalizes the security notion behind a ciphertext for multiple recipients. In particular, the most relevant notion in (Private) Broadcast Encryption is Recipient Privacy [3], in which an adversary cannot tell whether a public key is a valid recipient for a given ciphertext. PURBs goes further by enabling multiple simultaneous suites, while achieving indistinguishably from random bits in the ind$-cca2 model. PURBs also addresses size leakage.

Traffic morphing [62] 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 [55] or use the concept of Tor pluggable transport [36, 59, 60] that is applied to preventing Tor traffic from being identified and censored [11]. 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 only hide the traffic of one given tool, whereas PURBs are universal and arguably adaptable to any underlying application. Moreover, it has been argued 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. [22, 28, 54]. 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 potential solution in a different direction.

Traffic analysis aims at inferring the contents of encrypted communication by analyzing metadata. The most well-studied application of it is website fingerprinting [20, 38, 56, 57], but it has also been applied to video identification [42, 43, 50] and VoIP traffic [14, 61]. In website fingerprinting over Tor, research has repeatedly shown that the total website size is the feature that helps an adversary the most [15, 20, 37]. In particular, Dyer et al. [20] show the necessity of padding the whole website, as opposed to individual packets, to prevent an adversary from identifying a website by its observed total size. They also systematized the existing padding approaches. Wang et al. [58] propose deterministic and randomized padding strategies tailored for padding Tor traffic against a perfect attacker, which inspired our §4.

Finally, Sphinx [17] 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 metadata while providing cryptographic agility.

7 Conclusion

Conventional encrypted data formats leak information, via both unencrypted metadata and ciphertext length, that may be used by attackers to infer sensitive information via techniques such as traffic analysis and website fingerprinting. We have argued that this metadata leakage 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 cipher suites. Finally, we have introduced PADMÉ, a padding scheme that reduces the length leakage of ciphertexts and has a modest overhead decreasing with file size. PADMÉ performs significantly better than classic padding schemes with fixed block size in terms of anonymity, and its overhead is asymptotically lower than using exponentially increasing padding.

Acknowledgments

We are thankful to our anonymous reviewers and our meticulous proof shepherd Markulf Kohlweiss for their constructive and thorough feedback that has helped us to improve this paper. We also thank Enis Ceyhun Alp, Cristina Basescu, Kelong Cong, Philipp Jovanovic, Apostolos Pyrgelis and Henry Corrigan-Gibbs for their helpful comments and suggestions, and Holly B. Cogliati for text editing. This project was supported in part by grant #2017-201 of the Strategic Focal Area "Personalized Health and Related Technologies (PHRT)" of the ETH Domain and by grants from the AXA Research Fund, Handshake, and the Swiss Data Science Center.

References

[1] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES. In *Cryptographers' Track at the RSA Conference*, 2001.

[2] Diego F Aranha, Pierre-Alain Fouque, Chen Qian, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. Binary Elligator Squared. In *International Workshop on Selected Areas in Cryptography*, 2014.

[3] Adam Barth, Dan Boneh, and Brent Waters. Privacy in Encrypted Content Distribution Using Private Broadcast Encryption. In *International Conference on Financial Cryptography and Data Security*, 2006.

[4] Tal Be'ery and Amichai Shulman. A Perfect CRIME? Only TIME Will Tell. Black Hat Europe, 2013.

[5] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-Privacy in Public-Key Encryption. In *Advances in Cryptology – ASIACRYPT*, 2001.

[6] Mihir Bellare, Alexandra Boldyreva, Kaoru Kurosawa, and Jessica Staddon. Multi-Recipient Encryption Schemes: Efficient Constructions and Their Security. *IEEE Transactions on Information Theory*, 53(11), 2007.

[7] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptology*, 21(4), 2008.

[8] Mihir Bellare and Björn Tackmann. The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3. In *Annual International Cryptology Conference*, 2016.

[9] Daniel J Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *ACM CCS*, November 2013.

[10] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications, 2015.

[11] Tor Blog. Tor at the heart: Bridges and pluggable transports. https://blog.torproject.org/tor-heart-bridges-and-pluggable-transports, December 2016.

[12] Dan Boneh, Craig Gentry, and Brent Waters. Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys. In *Advances in Cryptology – CRYPTO*, 2005.

[13] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880, RFC Editor, November 2007.

[14] Yu-Chun Chang, Kuan-Ta Chen, Chen-Chi Wu, and Chin-Laung Lei. Inferring speech activity from encrypted Skype traffic. In *IEEE Global Telecommunications Conference*, 2008.

[15] Giovanni Cherubin, Jamie Hayes, and Marc Juarez. Website Fingerprinting Defenses at the Application Layer. In *Privacy Enhancing Technologies Symposium*, 2017.

[16] George Danezis and Richard Clayton. Introducing Traffic Analysis. *Digital Privacy: Theory, Technologies, and Prac-*

tices, December 2007.

[17] George Danezis and Ian Goldberg. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE Symposium on Security and Privacy*, 2009.

[18] Cécile Delerablée. Identity-Based Broadcast Encryption with Constant Size Ciphertexts and Private Keys. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2007.

[19] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008.

[20] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *IEEE Symposium on Security and Privacy*, May 2012.

[21] Nelly Fazio and Irippuge Milinda Perera. Outsider-Anonymous Broadcast Encryption with Sublinear Ciphertexts. In *International Workshop on Public Key Cryptography*, 2012.

[22] Sergey Frolov and Eric Wustrow. The use of TLS in Censorship Circumvention. In *Network and Distributed System Security Symposium*, 2019.

[23] Craig Gentry and Brent Waters. Adaptive Security in Broadcast Encryption Systems (with Short Ciphertexts). In Antoine Joux, editor, *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2009.

[24] Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: reviving the CRIME attack. Black Hat USA, 2013.

[25] B. Greschbach, G. Kreitz, and S. Buchegger. The devil is in the metadata 2014 – New privacy challenges in Decentralised Online Social Networks. In *IEEE International Conference on Pervasive Computing and Communications Workshops*, March 2012.

[26] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-bayes Classifier. In *ACM Workshop on Cloud Computing Security*, 2009.

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

[28] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *IEEE Symposium on Security and Privacy*, 2013.

[29] IDRIX. Veracrypt. https://www.veracrypt.fr/en/Home.html.

[30] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[31] John Kelsey. Compression and information leakage of plaintext. In *International Workshop on Fast Software Encryption*, 2002.

[32] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Annual Cryptology Conference*, 2010.

[33] Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In *International Workshop on Public Key Cryptography*, 2002.

[34] Stevens Le Blond, Chao Zhang, Arnaud Legout, Keith Ross, and Walid Dabbous. I Know Where You Are and What You Are Sharing: Exploiting P2P Communications to Invade Users' Privacy. In *ACM SIGCOMM Conference on Internet Measurement Conference*, 2011.

[35] Jonathan Mayer, Patrick Mutchler, and John C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences*, 113(20), 2016.

[36] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *ACM Conference on Computer and Communications Security*, 2012.

[37] Rebekah Overdorf, Mark Juarez, Gunes Acar, Rachel Greenstadt, and Claudia Diaz. How Unique is Your .onion?: An Analysis of the Fingerprintability of Tor Onion Services. In *ACM Conference on Computer and Communications Security*, 2017.

[38] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Workshop on Privacy in the Electronic Society (WPES)*, October 2011.

[39] Jeffrey Pang, Ben Greenstein, Ramakrishna Gummadi, Srinivasan Seshan, and David Wetherall. 802.11 User Fingerprinting. In *ACM International Conference on Mobile Computing and Networking*, 2007.

[40] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, 2009.

[41] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *USENIX Security Symposium*, 2018.

[42] Andrew Reed and Benjamin Klimkowski. Leaky streams: Identifying variable bitrate DASH videos streamed over encrypted 802.11n connections. In *IEEE Consumer Communications & Networking Conference (CCNC)*, 2016.

[43] Andrew Reed and Michael Kranch. Identifying HTTPS-protected Netflix videos in real-time. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[44] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor, August 2018.

[45] Ring-road: Leaking sensitive data in security protocols. http://www.ringroadbug.com/.

[46] Ivan Ristić. HTTP client fingerprinting using ssl handshake analysis. https://blog.ivanristic.com/2009/06/http-client-fingerprinting-using-ssl-handshake-analysis.html, June 2009.

[47] Tom Ritter and Daniel Kahn Gillmor. Protecting the TLS Handshake. IETF Interim, May 2014.

[48] Juliano Rizzo and Thai Duong. The CRIME attack. Ekoparty, 2012.

[49] Phillip Rogaway. Nonce-based symmetric encryption. In *International Workshop on Fast Software Encryption*, 2004.

[50] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security Symposium*, August 2017.

[51] Mehdi Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In *International Conference on Financial Cryptography and Data Security*, 2014.

[52] Thiago Valverde. Bad life advice - replay attacks against https. http://blog.valverde.me/2015/12/07/bad-life-advice/, December 2015.

[53] Guido Vranken. HTTPS Bicycle Attack. https://guidovranken.com/2015/12/30/https-bicycle-attack/, December 2015.

[54] Liang Wang, Kevin P Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. Seeing through network-protocol obfuscation. In *ACM Conference on Computer and Communications Security*, 2015.

[55] Qiyan Wang, Xun Gong, Giang TK Nguyen, Amir Houmansadr, and Nikita Borisov. Censorspoofer: asymmetric communication using IP spoofing for censorship-resistant web browsing. In *ACM Conference on Computer and Communications Security*, 2012.

[56] Tao Wang and Ian Goldberg. Improved Website Fingerprinting on Tor. In *ACM Workshop on Workshop on Privacy in the Electronic Society*, 2013.

[57] Tao Wang and Ian Goldberg. On realistically attacking Tor with website fingerprinting. In *Privacy Enhancing Technologies Symposium*, 2016.

[58] Tao Wang and Ian Goldberg. Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks. In *USENIX Security Symposium*, 2017.

[59] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: a camouflage proxy for the Tor anonymity system. In *ACM Conference on Computer and Communications Security*, 2012.

[60] Philipp Winter, Tobias Pulls, and Juergen Fuss. ScrambleSuit: A polymorphic network protocol to circumvent censorship. In *ACM Workshop on Workshop on Privacy in the Electronic Society*, 2013.

[61] Charles V Wright, Lucas Ballard, Fabian Monrose, and Gerald M Masson. Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In *USENIX Security Symposium*, 2007.

[62] Charles V. Wright, Scott E. Coull, and Fabian Monrose. Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis. In *Network and Distributed System Security Symposium*, 2009.

[63] Fan Zhang, Wenbo He, Xue Liu, and Patrick G. Bridges Bridges. Inferring Users' Online Activities Through Traffic Analysis. In *WiSec*, June 2011.

[64] Philip R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995.

A Layout

Algorithm 2 presents the LAYOUT algorithm a sender uses in step (8) of MsPURB.Enc. LAYOUT arranges a PURB's components in a continuous byte array.

Notation. We denote by $a[i:j] \leftarrow b$, the operation of copying the bits of b at the positions $a[i], a[i+1], \cdots a[j-1]$. When written like this, b always has correct length of $j-i$ bits, and we assume $i<j$. 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.

B Positions for Public Keys

This section provides an example of possible sets of allowed public key positions for the suites in the PURB encoding. We emphasize that finding an optimal set of positions was not the focus of this work. The intention is merely to show that such sets exist and to offer a concrete example (which is used for the compactness experiment, Figure 9).

Example. We use the required and recommended suites in the latest draft of TLS 1.3 [44] as an example of suites a PURB could theoretically support. The suites and groups are shown in Table 4.

The PURB concept of "suite" combines both "suite" and "group" in TLS. For instance, a PURB suite could be PURB_AES_128_GCM_SHA_256-_SECP256R1. We show possible PURB suites in Table 6. For the sake of simplicity, we introduce aliases in the table, and will further refer to those suites as suite A-F. In Table 5, we show a possible assignment. For instance, if only suites A and C are used, the public key for A would be placed in $[0,64]$, while value in $[96,160]$ is changed so that the XOR of $[0,64]$ and $[96,160]$ equals the key for B. Note that a sender must respect the suite order A-F during encoding. We provide a simple python script to design such sets in the code repository.

C Default Schemes for Payload

In addition to PURB suites, a list of suitable candidates for a payload encryption scheme (Enc, 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 suites included in the PURB, hence these schemes must match the security level of the suite with the highest bit-wise security. An example of suitable candidates, given the suites from Table 6, is (Enc, Dec) = AES256-CBC, MAC = HMAC-SHA384, and H' = SHA3-384.

Algorithm 2: 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 ← PADMÉ (|layout| + |c_payload| + mac_len);
mac_pos ← purb_len - mac_len;
while not pubkey_fixed.isFree(mac_pos, purb_len) do
    // MAC mustn't overlap public-key positions:
    // if so, we pad to the next Padmé size
    purb_len ← PADMÉ (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
```

Table 4. Suites and groups described in the latest draft of TLS 1.3.

Symmetric/Hash Algorithms	
TLS_AES_128_GCM_SHA256	Required
TLS_AES_256_GCM_SHA384	Recommended
TLS_CHACHA20_POLY1305_SHA256	Recommended
TLS_AES_128_CCM_SHA256	Optional
TLS_AES_128_CCM_8_SHA256	Optional
Key Exchange Groups	
secp256r1	Required
x25519	Recommended
secp384r1	Optional
secp521r1	Optional
x448	Optional
ffdhe2048	Optional
ffdhe3072	Optional
ffdhe4096	Optional
ffdhe6144	Optional
ffdhe8192	Optional

Table 5. Example of Allowed Positions per suite. Here, the algorithm simply finds any mapping so that each suite can coexist in a PURB. The receiver must XOR the values at all possible positions of a suite to obtain an encoded public key..

Suite	Possible positions
A	$\mathbf{\{0\}}$
B	$\mathbf{\{0, 64\}}$
C	$\mathbf{\{0, 96\}}$
D	$\mathbf{\{0, 32, 64, 160\}}$
E	$\mathbf{\{0, 64, 128, 192\}}$
F	$\mathbf{\{0, 32, 64, 96, 128, 256\}}$

D Security Proofs

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

D.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 [30], we begin by defining KEM as a tuple of PPT algorithms.

<u>Syntax KEM</u>.

$\mathsf{KEM.Setup}(1^\lambda) \to S$: 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.

$\mathsf{KEM.Decap}(sk, c) \to k/\bot$: Given a private key sk and a ciphertext c, output a key k or a special symbol $\bot$ denoting failure.

Consider an ind-cca2 security game against an adaptive adversary $\mathcal{A}$:

<u>Game KEM</u>.

The KEM ind-cca2 game for a security parameter λ is between a challenger and an adaptive adversary $\mathcal{A}$. 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 $\mathcal{A}$.

Phase 1: $\mathcal{A}$ 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 $\mathcal{A}$.

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

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

Table 6. PURB Suites. "Suite A" is a shorthand for the first suite.

Alias	PURB Suite	|Public key| [B]	|EntryPoint| [B]
A	PURB_AES_128_GCM_SHA_256_SECP256R1	64	48
B	PURB_AES_128_GCM_SHA_256_X25519	32	48
C	PURB_AES_256_GCM_SHA_384_SECP256R1	64	80
D	PURB_AES_256_GCM_SHA_384_X25519	32	80
E	PURB_CHACHA20_POLY1305_SHA_256_SECP256R1	64	64
F	PURB_CHACHA20_POLY1305_SHA_256_X25519	32	64

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

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

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

Definition 3. We 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 $k = \mathsf{KEM.Decap}(sk, c)$.

<u>Instantiation IES-KEM</u>.
We instantiate a KEM based on the Integrated Encryption Scheme [1] (see §2.2 for details).

$\mathsf{IES.Setup}(1^\lambda)$: 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 4 (Theorem 11.22 [30] and 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. [12].

<u>Syntax MSBE</u>.

$\mathsf{MSBE.Setup}(1^\lambda) \to S$: 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 $R = \{pk_1, \dots, pk_r\}$ with corresponding cipher suites $S_1, \dots, S_r$ and a message m, generate a ciphertext c.

$\mathsf{MSBE.Dec}(sk, c) \to m/\perp$: 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 §3.7 satisfies the syntax of a multi-suite broadcast encryption scheme.

Barth et al. [3] define the security of broadcast encryption schemes under adaptive chosen-chiphertext 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).

<u>Game MSBE</u>.
The MSBE ind\$-cca2 game for a security parameter λ is between a challenger and an adversary $\mathcal{A}$. 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 $S_1, \dots, S_r$ it wants to attack. Let s be the number of unique cipher suites. The challenger verifies, for each $i \in \{1, \dots, r\}$, 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 $\mathcal{A}$ by running $(sk_i, pk_i) \leftarrow \mathsf{MSBE.KeyGen}(S_i)$ and gives $R = \{pk_1, \dots, pk_r\}$ to $\mathcal{A}$.

Phase 1: $\mathcal{A}$ 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: $\mathcal{A}$ 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 $\mathcal{A}$.

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

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

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

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

We say that a MSBE scheme is ind\$-cca2-secure if $\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathcal{A}}(1^\lambda)$ 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 $\mathsf{MAC.KeyGen}, \mathcal{M}$, and $\mathcal{V}$, where $\mathsf{MAC.KeyGen}(1^\lambda)$ outputs a key K_{mac}. To compute a tag on the message m, run $\sigma = \mathcal{M}_{K_{mac}}(m)$. 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.

<u>Game MAC-sforge</u>.
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.

<u>Game MAC-IND\$</u>.
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.

D.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 $\mathcal{B}_1, \ldots, \mathcal{B}_5$ such that

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

Thus, given our assumptions, $\mathsf{Adv}^{\mathsf{cca2\text{-}out}}_{\mathsf{msbe},\mathcal{A}}(1^\lambda)$ 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.

<u>Game G_0</u>.
This game is as the original MSBE ind\$-cca2 game where $b = 0$.

<u>Game G_1</u>.
As in G_0, but the challenger will no longer call $\mathsf{HdrPURB.Decap}$ 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 $(X^\star_j, k^\star_i)$ 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 $\mathsf{MsPURB.Dec}$. If the encoded public key τ recovered in step (1) of $\mathsf{MsPURB.Dec}$ is such that $\mathtt{Unhide}(\tau) = X^\star_j$, then we use $k_i = k^\star_i$ (as stored when creating the challenge ciphertext) directly, rather than computing $k_i = \mathsf{HdrPURB.Decap}(y_i, \tau)$ in step (3) of $\mathsf{MsPURB.Dec}$. If the encoded public key τ does not match $X^\star_j$, then the challenger proceeds as before.

<u>Game G_2</u>.
As in G_1, but we change how the keys $k^\star_1, \ldots, k^\star_r$ for the *challenge ciphertext* are computed in $\mathsf{HdrPURB.Encap}$. Rather than computing $k^\star_i = \mathrm{H}(Y^x_i)$ as in step (2) of $\mathsf{HdrPURB.Encap}$, we set $k^\star_i \xleftarrow{\$} \{0,1\}^{\lambda_H}$ 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 $k^\star_i$ 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 (8) 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 (7) 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 $\mathcal{A}$ outputs $b^\star = 1$ in game G_i. We aim to show that

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

is negligible. To do so, we show that each of the steps in the sequence of games is negligible, i.e., that $\big|\Pr[W_i] - \Pr[W_{i+1}]\big|$ 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:

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

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 $\mathcal{A}$ cannot distinguish H_{j-1} from H_j. Let $S_j = \langle \mathbb{G}, p, g, \mathtt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$, be the suite corresponding to recipient j. Suppose $\mathcal{A}$ can distinguish H_{j-1} from H_j, then we can build a distinguisher $\mathcal{B}$ against the ind\$-cca2 security of the IES KEM for the suite $S_j' = \langle \mathbb{G}, p, g, \mathrm{H} \rangle$. Recall that $\mathcal{B}$ 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, $\mathcal{B}$ will set $pk_j = Y$, so that the public key of recipient j matches that of its IES KEM challenger. Note that $\mathcal{B}$ does not know the corresponding private key y_j. For all other recipients i, $\mathcal{B}$ sets $(sk_i = y_i, pk_i = Y_i) = \mathsf{MsPURB.KeyGen}(S_i)$.

The distinguisher $\mathcal{B}$ will use its challenge $(X^\star, k^\star)$ to construct the challenge ciphertext for $\mathcal{A}$. In particular, when running HdrPURB.Encap for a suite S_j, it sets $X = X^\star$ in step (1) of 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 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, $\mathcal{B}$ continues running $\mathsf{MsPURB.Enc}$ as before.

Whenever $\mathcal{B}$ 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 $\mathcal{B}$ 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 $\mathcal{B}$'s IES KEM challenge, then recipient j's key $k_j = \mathrm{H}(Y^{x^\star})$, and hence $\mathcal{B}$ perfectly simulates H_{j-1}. If $b = 1$ in $\mathcal{B}$'s IES KEM challenge, then j's key $k_j \xleftarrow{\$} \{0,1\}^{\lambda_H}$ and, hence, $\mathcal{B}$ perfectly simulates H_j. If $\mathcal{A}$ distinguishes H_{j-1} from H_j then $\mathcal{B}$ 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:

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

$\underline{G_0 <-> G_1}$.

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

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

$\underline{G_3 <-> G_4}$.

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 $\mathcal{A}$ cannot distinguish H_{j-1} from H_j. Let $S_j = \langle \mathbb{G}, p, g, \mathtt{Hide}(\cdot), \Pi, \mathrm{H}, \hat{\mathrm{H}} \rangle$, be the suite corresponding to recipient j. We show that if $\mathcal{A}$ distinguishes H_{j-1} from H_j then we can build a distinguisher $\mathcal{B}$ against the ind\$-cca2 security of Π. $\mathcal{B}$ 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 $e^\star = \mathcal{E}_Z(m)$ if $b = 0$ or $e^\star$ is a random string if $b = 1$;
– a decryption oracle $\mathcal{D}_Z(\cdot)$.

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

If $b = 0$, $\mathcal{B}$ simulates H_{j-1}, and if $b = 1$, it simulates H_j. Therefore, if $\mathcal{A}$ distinguishes between H_{j-1} and H_j, then $\mathcal{B}$ breaks the ind\$-cca2 security of Π. To show that G_3 is indistinguishable from G_4, repeat this argument r times. More precisely:

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

$\underline{G_4 <-> G_5}$.

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

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

Simulator $\mathcal{B}$ now proceeds as follows. When creating the challenge ciphertext $c^\star$, it does not compute σ in step (9) of $\mathsf{MsPURB.Enc}$ using $K^\star$, but instead uses its oracle $\mathcal{M}$ and sets $\sigma = \mathcal{M}(c')$. Note that because of the random oracle model for H' and the fact that $\mathcal{A}$'s view is independent of $K^\star$, this change of K_{mac} remains undetected.

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

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

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

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

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

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

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

G_6 <−> G_7.
If $\mathcal{A}$ can distinguish between G_6 and G_7, then we can build a distinguisher $\mathcal{B}$ that breaks the ind$-cpa property of $(\mathsf{Enc}, \mathsf{Dec})$. In the ind$-cpa game [49], $\mathcal{B}$ receives:
- a challenge ciphertext $c_{\mathsf{payload}} = c_b$, s.t. $c_0 = \mathsf{Enc}_{K_{enc}}(m)$ on a chosen-by-$\mathcal{B}$ m, $c_1 \xleftarrow{\$} \{0,1\}^{|c_0|}$, and $b \xleftarrow{\$} \{0,1\}$.

$\mathcal{B}$ runs MsPURB.Dec as before to create a challenge for $\mathcal{A}$, except that $\mathcal{B}$ uses the ind$-cpa challenge ciphertext c_{payload} in step (7), instead of encrypting, as $\mathcal{B}$ does not know K_{enc}. As before, $\mathcal{A}$'s view is independent of $K^\star$, so also this change of K_{enc} remains undetected.

$\mathcal{B}$ answers decryption queries $\mathsf{qDec}(pk_i(S_i), c)$ from $\mathcal{A}$ as before. In particular
- if $\mathtt{Unhide}(\tau) = X_i^\star$ and $e_i = e_i^\star$, $\mathcal{B}$ returns $\perp$ as per the changes in G_5;
- Otherwise, $\mathcal{B}$ runs $\mathsf{MsPURB.Dec}(\cdot)$.

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

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

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

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

completing the proof. □

D.3 Proof of Theorem 2

For our MsPURB ind$-cpa recipient-privacy game, we take inspiration from the single-suite recipient-privacy game defined by Barth et al. [3], but we restate it in the ind$-cpa setting.

Game Recipient-Privacy.
The game is between a challenger and an adversary $\mathcal{A}$, 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 $\mathcal{A}$ 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 $\mathcal{A}$, the challenger runs $(sk_i, pk_i) \leftarrow \mathsf{MsPURB.KeyGen}(S_i)$, where S_i is previously chosen by $\mathcal{A}$. 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 $\mathcal{A}$, where R_0, R_1 are the generated public keys of the recipients N_0, N_1 respectively. The challenger also gives to $\mathcal{A}$ all sk_i that correspond to $i \in N_0 \cap N_1$.

Challenge: $\mathcal{A}$ 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 $\mathcal{A}$.

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

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

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

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

The conditions on N_0 and N_1 in the game ensure that $\mathcal{A}$ 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 [3]. 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

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

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

Proof. Similarly to Barth et al. [3], 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:

<u>Game G_0</u>.
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.

<u>Game G_1</u>.
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 $k_i^\star$.

<u>Game G_2</u>.
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 $\mathcal{A}$ cannot distinguish between G_0 and G_2, $\mathcal{A}$ does not have an advantage in winning the recipient-privacy game.

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

<u>$G_0 <-> G_1$</u>.
If $\mathcal{A}$ can distinguish between G_0 and G_1, we can build a distinguisher $\mathcal{B}$ against the ind\$-cca2 security of the IES KEM. Recall that $\mathcal{B}$ 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, $\mathcal{B}$ will set $pk_i = Y$, so that the public key of recipient i matches that of its IES KEM challenger. Note that $\mathcal{B}$ does not know the corresponding private key y_i. For all other recipients h, $\mathcal{B}$ sets $(sk_h = y_h, pk_h = Y_h) = \mathsf{MsPURB.KeyGen}(S_h)$. As $\mathcal{A}$ plays an ind\$-cpa game, $\mathcal{B}$ 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 $\mathcal{B}$ simulates G_0, and, If $b = 1$, $\mathcal{B}$ simulates G_1. Hence, if $\mathcal{A}$ distinguishes between G_0 and G_1, $\mathcal{B}$ wins in the ind\$-cca2 IES-KEM game. Therefore:

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

<u>$G_1 <-> G_2$</u>.
The proof follows the same steps as the proof of $G_0 <-> G_1$. Therefore:

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

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:

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

as desired. □