Breaking and Fixing Content-Defined Chunking

Kien Tuong Truong
kientuong.truong@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Simon-Philipp Merz
research@simon-philipp.com
ETH Zurich
Zurich, Switzerland

Matteo Scarlata
matteo.scarlata@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Felix Günther
mail@felixguenther.info
IBM Research Europe – Zurich
Rüschlikon, Switzerland

Kenneth G. Paterson
kenny.paterson@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

Content-defined chunking (CDC) algorithms split streams of data into smaller blocks, called chunks, in a way that preserves chunk boundaries when the data is partially changed. CDC is ubiquitous in applications that deduplicate data such as backup solutions, software patching systems, and file hosting platforms. Much like compression, CDC can introduce leakage when combined with encryption: fingerprinting attacks can exploit chunk length patterns to infer information about the data.

To address these risks, many systems—mainly in the cloud backup setting—have developed bespoke mitigations by mixing a cryptographic key into the chunking process. We study these keyed CDC (KCDC) schemes "in the wild", presenting efficient key recovery attacks against five different KCDC schemes, deployed in the backup solutions Borg, Bupstash, Duplicacy, Restic, and Tarsnap. Our attacks are in a realistic threat model that relies only on weak known- or chosen-plaintext capabilities. This shows, in particular, that they fail to protect against fingerprinting attacks. To demonstrate practical exploitability, we also present "end-to-end" attacks on three complete encrypted backup applications, namely Borg, Restic and Tarsnap. These build on our attacks on the underlying KCDC schemes.

In an effort to tackle these problems, we introduce the first formal treatment for KCDC schemes and propose a provably secure construction that fulfills a strong notion of security. We benchmark our construction against existing (broken) approaches, showing that it has competitive performance. In doing so, we take a step towards making real-world systems that rely on KCDC more resilient to attacks.

1 Introduction

Today, more than ever, users store and communicate vast amounts of data. Minimizing network bandwidth and storage space is essential: data deduplication—checking for duplicates in data and storing or transmitting those only once—helps achieve that, particularly when it comes to large and redundant datasets.

Traditionally, deduplication is done on a per-file level. *Chunking* up data (i.e., splitting it into smaller blocks) before deduplication, and then deduplicating *at the level of chunks* rather than entire files however greatly improves efficiency: Chunk deduplication can reduce storage by up to 49% [11] on virtual machine image datasets, and by up to 83% for backup workloads [19].

Content-defined chunking. The most näive approach to chunking is to split data into fixed-size chunks. Fixed-size chunking has the disadvantage that the insertion or deletion of a single byte shifts all of the subsequent data, causing all later chunks to change, thus preventing efficient deduplication. For this reason, many applications resort to *content-defined chunking* (CDC). With CDC, the boundary of a block is not pre-determined, but dynamically computed depending on the data itself: a *chunking scheme* scans through the data's bytes, using only a small byte window to decide whether to cut a chunk at a given position. This means that if a byte is inserted or deleted at some point in a data sequence, the boundaries of most chunks before and after the modification will (with high probability) not change: only the affected chunk will grow or shrink in size.

CDC in the wild. CDC is used everywhere: its effectiveness makes it the tool of choice for many applications, ranging from backup solutions such as Borg [6], Bupstash [8], Duplicacy [18], Restic [25], and Tarsnap [34], to the League of Legend software patcher [35], the HuggingFace backend [12] and the IPFS distributed file system [15]. CDC schemes are typically implemented using lightweight hash functions having some algebraic structure, such as polynomial hashes. This allows not only efficient computation of hash values, but also efficient updating of the hash value in a "rolling" fashion, one byte at a time.

Fingerprinting attacks. Consider the common setting of file backups outsourced to an untrusted cloud server, such as chat message history or medical document folders. For privacy-sensitive data like this, client-side encryption is used with the expectation that the cloud server then cannot learn information about the data being stored.

However, if the files are chunked using CDC, the length of each chunk will depend on its content. Since most encryption algorithms do not (fully) hide the length of plaintexts, if encryption is applied at the chunk level (as is typically the case), then the length of the encrypted chunks can leak information about the plaintext data being chunked. While full recovery of plaintext data from this leakage is challenging, *fingerprinting attacks* can be much easier to mount. The idea is that the vector (or just the set) of chunk lengths for a file may provide a *fingerprint* that uniquely identifies the file amongst a set of files known to the adversary. This would enable the adversary to check whether, for instance, a particular target file is present in a backup or not. Fingerprints can also be helpful if only part of the data is known: if the unknown part is small enough, the adversary can enumerate all possibilities for the unknown part and use the resulting chunk sizes as a fingerprint to determine whether a guess was correct or not.

Fingerprinting attacks were first systematically explored in the CDC context in [28]. They are, of course, analogous to other forms of fingerprinting attacks, e.g. on encrypted web traffic [9, 32] and the setting is broadly similar to that of compression side channels [16, 29], where the length of the ciphertext after a compression-then-encryption operation leaks some information that can be used to recover plaintext.

Keyed content-defined chunking. The danger of fingerprinting attacks in the context of CDC is well understood in the developer community. Indeed, many applications use bespoke CDC schemes whose behavior depends on secret inputs. We dub these *keyed content-defined chunking* (KCDC) schemes. Using a key necessarily limits the benefits of deduplication after KCDC to the set of parties with access to the key. Nevertheless, KCDC schemes are now used in most major encrypted backup software: notably Borg [6], Bupstash [8], Duplicacy [18], Restic [25], and Tarsnap [34] all have their own, distinct keyed chunking implementations.

Breaking KCDC. Most of these keyed chunking schemes we find "in the wild" are the result of adapting existing unkeyed CDC schemes, applying folklore mitigations in order to make them keyed while preserving their performance. The resulting designs are, however, cryptographically unprincipled, and rely on unclear or unstated assumptions for their security. Indeed, no formal security notion for KCDC has been proposed in the literature, so it is not even precisely defined what "security" should mean in the context of KCDC.

Here, we show that KCDC constructions used in practice can be broken in a realistic threat model: an adversary who can observe the chunking fingerprint for a single *known* file (and in one case, Tarsnap, a single *chosen* file) can recover the key of the KCDC scheme. Thereafter, the adversary can carry out standard fingerprinting attacks. Such an adversary could be situated on the network between a client and a cloud server, or could be based at the server itself. Since each of the KCDC schemes we look at involves a different approach, we require a bespoke analysis for each one. Specifically, based on their popularity and diversity of approaches, we analyzed the five different KCDC schemes in the backup solutions Borg, Bupstash, Duplicacy, Restic, and Tarsnap.

Breaking applications using KCDC. While we show that it is possible to break KCDC schemes in isolation, our attacks may not always be exploitable in the wider, "end-to-end" setting of the applications using these KCDC schemes. In particular, KCDC is typically used in combination with subsequent compression, padding, encryption and other features that could negate attempts to carry out end-to-end attacks. This raises the question of whether our attacks on KCDC schemes are actually meaningful in practice. To answer this question, we present full attacks against the Borg, Restic and Tarsnap systems. These build on our attacks on their underlying KCDC schemes, showing that their weaknesses do allow us to run fingerprinting attacks against the full systems and not just the standalone KCDC schemes, albeit requiring some assumptions on how the backup is created. We also briefly explain how various features frustrate our attacks on the other systems, pointing towards possible attack countermeasures.

Fixing KCDC. Our attacks demonstrate that unprincipled, folklore mitigations used to turn unkeyed CDC schemes into keyed ones are error-prone and often insecure. This calls for a principled approach to KCDC, which we initiate through the first formal treatment of such schemes and their security. We propose strong game-based security notions for KCDC, which guarantee that chunking fingerprints do not leak information about underlying data. We also present a provably secure construction for an efficient chunking scheme, which we benchmark against the state-of-the-art KCDC schemes "in the wild". Our construction is based on the composition of a polynomial hash function, which acts as a universal hash function (UHF) with a rolling property, and a block cipher (e.g. AES), which acts as a PRF. Since many schemes already rely on a polynomial hash function, we only incur a minimal overhead of one block cipher evaluation per update operation. This makes our construction reasonably competitive with existing (broken) KCDC schemes, thanks to widespread support for AES instructions in hardware. We also show that the hash function used in Restic is already universal (with a certain collision probability of about 2^{-44}) meaning that Restic can be directly repaired simply by post-processing its hash values with, say, AES.

1.1 Our Contributions

In this work, we contribute the following novel results:

- We analyze a sample of five different systems using keyed content-defined chunking (KCDC), focusing on those which employ mitigations to thwart fingerprinting attacks. We show that, in each case, these mitigations are insufficient. More precisely, we show that an adversary who knows the chunking behaviour on a single known or chosen file can recover the key of the KCDC scheme. After this, standard fingerprinting attacks become possible again. We also show that our attacks on KCDC do extend to attacks on deployed systems in three cases: Borg, Restic and Tarsnap. We discuss barriers to extending our attacks for other systems.
- We provide the first formal treatment of KCDC schemes. We propose and relate strong game-based security notions for KCDC. These guarantee that the chunking fingerprint does not leak information about underlying data, beyond what is necessarily revealed by the deterministic nature of KCDC schemes (e.g., the possible repetition of complete chunks through equality of chunk sizes). We discuss to what extent this definition protects against fingerprinting and other forms of attack.
- Finally, we present a provably secure, efficient construction for a KCDC scheme. Our construction is based on a "standard" PRF construction: apply a Universal Hash Function (UHF) to a window of data, then a PRF (instantiated using a truncated block cipher for efficiency), and chunk based on whether the PRF output is the zero string or not. The overhead of our construction as compared to existing (but broken!) KCDC schemes "in the wild" is one block cipher call per byte of processed data. In practice, this does lead to a reduced throughput. For example, applying our construction on top of the Restic hash function reduces throughput by 53%–165% in our measurements. However, we consider this to be a reasonable price to pay for gaining a cryptographically sound KCDC scheme.

1.2 Related Work

Content-defined chunking. Due to its efficiency properties, CDC has been widely studied in the literature and is used in many deployed systems. On the academic side, the Low-Bandwidth File System (LBFS) [20] was the first to introduce the technique. Other, more recent, proposals harness CDC to reduce the data footprint of encrypted Docker images [33] or to optimize cloud storage systems [38]. These two works, despite operating on encrypted data, fail to consider leakage from the chunking algorithm.

The literature on algorithms for CDC is rich, with works such as [36] and its successive improvements in [21, 37, 39]. These present a variety of chunking algorithms optimized for data throughput. These works focus on efficiency and do not consider security.

Deduplication. There is a significant amount of research on and deployment of deduplication schemes for cloud storage systems, but much of this work considers deduplication at the file level, and does not treat CDC, i.e., deduplication below the level of files. The works [4, 7] lay formal cryptographic foundations for cloud storage systems with deduplication and analyse industry-wide practices such as *convergent encryption* as introduced in [10].

Fingerprinting attacks. Fingerprinting as a general attack class of course goes well beyond CDC. For example, [9, 14, 23] study website fingerprinting attacks in anonymity networks like Tor, using statistical methods to predict websites from traffic features such as include packet size, but also cover timing information, directionality, and number of packets. Gellert *et al.* [13] study fingerprinting based solely on message length on a number of real-world datasets, and propose the use of length-hiding encryption as mitigation.

In the specific setting of CDC, Ritzdorf *et al.* [28] study fingerprinting attacks on storage systems. Their analysis does not treat keyed CDC schemes. Their focus is on quantifying the information leakage of such schemes. This is done by estimating the number of distinct fingerprints via a statistical model and then experimenting with their own CDC scheme based on Rabin hashing. In contrast, we study specific, deployed, keyed CDC schemes, showing practical key recovery attacks on both the KCDC schemes in isolation and on their usage in real systems. In turn our attacks enable fingerprinting attacks like those studied in [28]. We also introduce a formally sound approach to building KCDC schemes and provide an instantiation and comparison with state-of-the-art KCDC schemes, a topic not addressed in [28].

Attacks on KCDC Schemes. While contacting the developers of Tarsnap, we learned that an Alexeev, Percival, and Zhang independently discovered similar attacks in 2023, which was made public on 22 Mar 2025 [2]. Our work has been developed independently of theirs, and we have not had access to their paper prior to our disclosure. We have, however, had the opportunity to discuss our results with them and to contribute opinions to their suggested fixes. With respect to their attacks, we include a more performant key-recovery attack on Borg and Restic, alternative attacks on Tarsnap, and additional attacks on Duplicacy and Bupstash, which were not covered in their paper.

1.3 Ethical Considerations

We disclosed our findings to the developers of Borg, Bupstash, Restic, and Tarsnap on 9 Jan 2025, proposing a 60-day non-disclosure period. The author of Tarsnap responded on 10 Jan 2025, stating that they had been notified of similar issues and later released a preprint version of a paper containing independent work on fingerprinting attacks [2]. Version 1.0.41 of Tarsnap includes a targeted fix which makes both our attacks and the attacks described in their paper infeasible. The author of Borg responded on 3 Mar 2025, acknowledging our results and discussing the difficulty of implementing fixes in the current version of Borg. The authors of Restic and of Bupstash also acknowledged our results, both on 16 Mar 2025. We did not contact the developers of Duplicacy, as they already guard against fingerprinting attacks by padding chunks to a multiple of 256 bytes. While the core KCDC scheme they propose is insecure on its own and their mitigation is not provably secure, the padding prevented us from developing a full attack on Duplicacy. We discuss the effectiveness of this mitigation in Section 5.

1.4 Structure of the Paper

The remainder of the paper is organised as follows. In Section 2 we give basic definitions and further background. Section 3 describes our threat model and justifies our adversarial assumptions, and then presents our attacks against the five different KCDC schemes. In Section 4 we give our formal security definitions for KCDC, along with our construction and a proof of its security. We also report benchmarking results there. In Section 5 we explain how our attacks for standalone KCDC schemes extend to the system level. We also discuss how our security definition should be interpreted, as well as its limitations. Section 6 provides our final conclusions.

2 Background

2.1 Notation and Conventions

Ordered lists are denoted with square brackets and have a method $\ell.append(x)$ which adds x as the last element of the list ℓ. For a byte string $s \in \{0, \dots, 255\}^*$, we denote by $|s|$ the length of s in bytes. For both lists and byte strings, we use $x[i]$ to denote the i-th element of x, starting from 0 and we use slice notation $x[i:j]$ to denote the sub-list (resp. the substring) of x from index i to j (exclusive). If i (resp. j) is omitted it takes the value 0 (resp. $|x|$). Slices can also take negative indices: for $0 \leq j \leq i \leq |x|$, $x[-i:-j] = x[|x|-i:|x|-j]$. We denote by $x||y$ the concatenation of two byte strings x and y.

Return values of probabilistic algorithms are denoted by $\leftarrow\!\!\$$, while the output of deterministic algorithms is indicated using $\leftarrow$. We also use $x \leftarrow y$ to denote that we assign the value from the right-hand side to the variable x.

For any bit-guessing security game G and adversary $\mathcal{A}$, we denote by $\Pr[G]$ the probability that $\mathcal{A}$ outputs 0 as its guess in G.

We will use the following lemma (from [5, Thm. 4.7]):

Lemma 2.1 (Difference Lemma). *Let Z, W_0, W_1 be events defined on the same probability space. Suppose $W_0 \wedge \neg Z$ occurs if and only if $W_1 \wedge \neg Z$ occurs. Then* $|\Pr[W_0] - \Pr[W_1]| \leq \Pr[Z]$.

2.2 Keyed Content-Defined Chunking

A chunking scheme is usually applied to big files or blobs of data, in order to split up, or *chunk*, the file into smaller *chunks*, which can then be stored, manipulated, or transmitted over the network more easily. Here we give a formal definition of chunking schemes, and position our definition with respect to how chunking schemes are used in practical applications.

We focus mostly on *keyed content-defined chunking* (KCDC) schemes – schemes where the position of the chunk boundary depends on the data being chunked and a key – and give a general treatment. Unkeyed chunking schemes can be seen as a particular case where the key is fixed and public.

Definition 2.2 (Keyed content-defined chunking scheme). A chunking scheme $\mathsf{Chk} = (\mathsf{Chk.KGen}, \mathsf{Chk.Init}, \mathsf{Chk.Update}, \mathsf{Chk.Eval})$ is a tuple of algorithms, where:

- $K \leftarrow\$ \mathsf{Chk.KGen}()$ is the key generation algorithm outputting a key K.
- $\sigma \leftarrow \mathsf{Chk.Init}(K)$ is the initialization algorithm, outputting an initial state σ on input a key K.
- $\sigma' \leftarrow \mathsf{Chk.Update}(K, \sigma, B)$ on input a key K, the current state σ, and the next byte $B \in \{0, \ldots, 255\}$ of the to-be-chunked data, outputs an updated state σ'.
- $0/1 \leftarrow \mathsf{Chk.Eval}(K, \sigma)$ on input a key K and the current state σ, outputs a bit $b \in \{0, 1\}$ indicating the chunking decision at the current position.

Concretely, an application using a KCDC scheme would read the input data byte-by-byte, feeding each byte to Update, and then using Eval to decide whether to insert a chunk boundary after each byte is processed. We denote by $\sigma' \leftarrow \mathsf{Chk.Update}^*(K, \sigma, \text{data})$ the state σ' obtained by repeated calls to Chk.Update with every byte of data $\in \{0, ..., 255\}^*$ in order, starting from the initial state σ.

Since chunks are the basic unit of data for further processing (e.g., for deduplication), their average length must be appropriately chosen as part of the design of a chunking scheme. For deduplication, chunks should be small enough to have a good probability of being repeated in different files, but not too small that overhead becomes an issue. The specific value depends on the application, but is usually in the order of megabytes; we will make the choices of the schemes we analyze explicit.

Our definition yields a canonical way of chunking a sequence of bytes, as shown in Fig. 1. Let Chk be a KCDC scheme and let K be the key output by a call to Chk.KGen. We define a function f_{Chk} that given as input some data $\in \{0, \ldots, 255\}^*$, *chunking parameters* (minL, maxL), and K, outputs a sequence of chunks $(c_0, \ldots, c_n)$, such that $c_0 || \ldots || c_n = \text{data}$ and $\text{minL} \leq |c_i| \leq \text{maxL}, \forall 0 \leq i < n$. That is, each chunk (except the last) has a minimum size of minL bytes and a maximum size of maxL bytes. We call $f_{\mathsf{Chk}}(\text{data, minL, maxL, K})$ the *chunked view* of data.

All the applications we surveyed share similar implementations of f_{Chk}: the minimum chunk size is not guaranteed by the chunking scheme itself, but rather by skipping the minL bytes of data and then invoking the chunking scheme until either Chk.Eval returns 1 or maxL bytes have been processed, as captured by f_{Chk} in Fig. 1.

```
f_Chk(data, minL, maxL, K):
i ← 0; j ← minL
c ← []
While j < |data|:
    σ ← Chk.Init(K)
    While j < |data| ∧ j < maxL ∧ Chk.Eval(K, σ) = 0:
        σ ← Chk.Update(K, σ, data[j])
        j ← j + 1
    c.append(data[i : j])
    i ← j; j ← j + minL
If i < |data|:
    c.append(data[i : j])
Return c
```

Figure 1: Canonical chunking implementation of f_{Chk}.

3 Key Recovery Attacks

In this section, we present our attacks on five keyed content-defined chunking (KCDC) implementations deployed in Borg [6], Bupstash [8], Duplicacy [18], Restic [25], and Tarsnap [34]. They all apply their own, distinct folklore mitigations to existing, unkeyed chunking schemes to prevent fingerprinting attacks, and we accordingly require a bespoke analysis for each system. The goal our attacks achieve however is always the same: to *recover the key* used for chunking, showing that these mitigations are not effective. Indeed, once an attacker has recovered the key via our attacks, it can carry out the standard fingerprinting attacks that these schemes intended to prevent.

Our attacks on the folklore mitigations work in a realistic threat model, and are possible if an adversary can observe the chunking fingerprint for a certain amount of data, possibly a single file. This includes both network adversaries, observing traffic between a client and a cloud server, as well as the cloud server itself being malicious. In all but one case, the adversary merely needs to *know* the data content (i.e., the attacks are *known-plaintext* ones); only in the case of Tarsnap does the attack require a *chosen plaintext*. All known-plaintext attacks merely require the data contents to be "independent enough" (i.e., non-redundant), so that the attacker can actually observe sufficiently many distinct chunks. We deem such known-plaintext attacks to be very viable in cloud backup settings, where user backups regularly include large amounts of known data (e.g., system files). Similarly, a malicious cloud server can plausibly even mount a chosen-plaintext attack, e.g., by sending the user some files which are stored and automatically included in a backup. Notably, we will show through our construction in Section 4 that these (and even stronger) attacks can be avoided with minimal overhead, through a principled cryptographic design.

3.1 Borg

Borg's chunking scheme is based on Buzhash [6], a simple rolling hash which is widespread in unkeyed CDC implementations.

Buzhash operates over the quotient ring $R := \mathbb{F}_2[x]/(x^{32} + 1)$. Note that, in this ring, every element can be seen as a 32-bit string, with bits corresponding to polynomial coefficients. In R, summing or subtracting two elements corresponds to the XOR of their bit-representations, while multiplying by x corresponds to a bit rotation to the left (due to the equivalence $x^{32} = 1$). Borg creates a secret mapping $\phi_K \colon \{0, \ldots, 255\} \to R$ from bytes to ring elements at initialization of the local backup. This mapping is created by composing a public and fixed mapping $\psi \colon \{0, \ldots, 255\} \to \mathbb{F}_2^{32}$ with a secret key $K \in R$: $\phi_K(b) := \psi(b) + K$.

Table 1: Summary of the attacks on the various backup systems we considered. When not explicitly noted otherwise, we consider the default parameters for chunking algorithm, compression settings and padding. Plaintext access ranges from none (○), to known-plaintext (◑) and chosen plaintext (●). The attacks are either tested (in red) or theoretical (in yellow).

	Version	CDC Key Recovery	E2E Attack	Ptxt Access	Chunks Required	λ	CDC Algorithm	Compression	Padding
Kopia (unkeyed)	v0.18.2	✓	✓	○	0	2MiB	Buzhash	s2-default	none
Borg (§3.1)	v1.4.0	✓	✓	◑	1	2MiB	Buzhash (secret offset)	lz4	none
	(`obfs L1`)		(✓)		(~10)				(random-%)
Bupstash (§3.2)	v0.12.0	✓	✓	◑	~1000	2MiB	GearHash (secret table)	zstd	none
Duplicacy (§3.3)	v3.2.4	✓	✗	◑	~745	4MiB	Buzhash (secret table)	lz4	multiple of 256
Restic (§3.4)	≤ v0.14.0	✓	✓	◑	~492	1MiB	Rabin-hash (secret field)	none	none
	v0.15.2		✗		n/a			zstd	none
Tarsnap (§3.5)	v1.0.40	✓	✓	●	$5 + {\sim}3 \cdot 254$	64 KiB	Rabin-like (secret mapping)	zlib	none

The state of the Buzhash scheme consists of a hash value s initialized to $0 \in R$ and a queue Q with 4095 positions containing all bytes in the current sliding window. Then, for each ingested byte b, s is updated as $s \longleftarrow (s - \phi_K(Q[0]) \cdot x^{31}) \cdot x + \phi_K(b)$, where $Q[0]$ is the byte leaving the window and Q is updated accordingly by appending b. A chunk is cut if the lowest 21 bits of s are all 0.

Let $(b_0, \ldots, b_{4094})$ be a sliding window that caused chunking. This implies that

$$\sum_{i=0}^{4094} \phi_K(b_i) \cdot x^{(4094-i) \bmod 32} \bmod x^{21}$$
$$= \sum_{i=0}^{4094} \left(\psi(b_i) \cdot x^{(30-i) \bmod 32} + K \cdot x^{(30-i) \bmod 32} \right) \bmod x^{21} = 0,$$

where the left part of the summand is publicly known under a known-plaintext assumption.

Due to the fact that $4094 = 127 \cdot 32 + 30$, each power x^j for $j \in [0, 30]$ appears an even number of times (128 times, specifically) which then get cancelled out completely, while x^{31} appears 127 times. This implies that the sum above can be simplified to

$$\sum_{i=0}^{4094} \left(\psi(b_i) \cdot x^{(30-i) \bmod 32} \right) + K \cdot x^{31} \bmod x^{21} = 0.$$

It is then trivial to recover the lowest 21 bits of K from a known plaintext of a single chunk (i.e. 2 MiB on average). Since we are not given the higher bits of the hash, we cannot recover the full key. However, this is not needed: only the lowest 21 bits of K contribute to the chunking decision. As a consequence, the effective entropy of the secret is only 21 bits. We have implemented this attack on a Borg repository with a known plaintext and successfully recovered the secret key.

3.2 Bupstash

The Bupstash chunking scheme is based on GearHash [37]. At initialization, Bupstash samples a 32-byte key K, which is then used in a cryptographic pseudo-random number generator to create a secret mapping $\phi\colon \{0, \ldots, 255\} \longrightarrow \mathbb{Z}/2^{32}\mathbb{Z}$ from bytes to 32-bit strings, to be interpreted as elements of the ring of integers modulo 2^{32}. For optimization purposes, the version used by Bupstash is an interleaved variant which reads the file 8 bytes at a time, computing 8 GearHashes in parallel, one for each byte: the i-th instance of GearHash will ingest bytes $i, i+8, i+16, \ldots$ of the file. GearHash, as used by Bupstash, is a rolling hash function that maintains a 32-bit state, initialized at 0, and ingests bytes one at a time, operating over $\mathbb{Z}/2^{32}\mathbb{Z}$. The state is updated with a new byte b by shifting the state by one bit to the left (equivalent to multiplying by 2) and adding the value of $\phi(b)$. It is immediate to see that only the last 32 bytes processed contribute to the hash.

The chunking condition is then evaluated for each hash output by checking whether the 21 leading bits of the hash are all 0 (equivalent to the hash representing a number smaller than 2^{11}). A chunk boundary is created if any of the hashes fulfills this condition, in which case the boundary is created at the position of the byte that was ingested by the GearHash instance that triggered chunking. By checking the length of the chunk modulo 8, it is possible to determine which GearHash instance triggered the chunking condition, and thus the sequence of bytes that caused the chunking condition to be fulfilled. Henceforth, we disregard this interleaving and consider only the 32 bytes that contributed to a positive chunking decision, which we still call the "sliding window".

Let $B = (b_0, \ldots, b_{31})$ be a sliding window that caused chunking. Then, over $\mathbb{Z}/2^{32}\mathbb{Z}$, it holds that $\sum_{i=0}^{31} \phi(b_i) \cdot 2^{31-i} = 2^{10} + e$ with $|e| \leq 2^{10}$, since the result must be smaller than 2^{11} over $\mathbb{Z}$. The left-hand side of the equation can be rewritten by grouping the terms which correspond to the same byte value: let $c_i = \sum_{j \in \{0,\ldots,31\} \mid b_j = i} 2^{31-j}$. Then the equation becomes $\sum_{i=0}^{255} c_i \cdot \phi(i) = 2^{10} + e$. By observing m sliding windows, we can collect m equations of this form, which can be interpreted as an instance of the Learning With Errors (LWE) problem over $\mathbb{Z}/2^{32}\mathbb{Z}$, with error terms $e_1, \ldots, e_m$ sampled uniformly from $[0, 2^{10})$ and secrets $\phi(0), \ldots, \phi(255)$ sampled uniformly from $\mathbb{Z}/2^{32}\mathbb{Z}$.

We solved a reduced instance of this LWE problem by assuming a file with a reduced character set, containing only 180 distinct bytes. This assumption is reasonable as, for example, the ASCII character set only uses 128 different byte values. We implemented the attack for $m = 500$ (equivalent to approximately 1 GiB of data) using the uSVP method, with a combination of flatter [31] and G6K [1] for lattice basis reduction. Our implementation has a one-time cost of approximately 8 hours runtime on an AMD EPYC 7742 64-core CPU to recover the key of a user.

3.3 Duplicacy

Similarly to Borg, Duplicacy's content-defined chunking algorithm is also based on Buzhash. However, it operates over the larger quotient ring $R := \mathbb{F}_2[x]/(x^{64}+1)$ and Duplicacy creates the mapping $\phi\colon \{0,\dots,255\} \to R$ from bytes to the ring by using SHA-256. More specifically, a random 32-byte key K is generated. Then, SHA-256(K) is evaluated and split into four 64-bit integers, corresponding to $\phi(0),\dots,\phi(3)$. The resulting hash is then hashed again to obtain the next four values for ϕ, and so on. Concretely, let $i \in \{0,\dots,255\}$ then $\phi(i) = \text{SHA-256}^{\lfloor i/4 \rfloor}(K)[(i \bmod 4)\cdot 64 : (i \bmod 4+1)\cdot 64]$. This means that knowing the mapping for $i = 0,\dots,3$ allows to compute the mapping for all other bytes.

The chunking algorithm otherwise follows the same principle as the one of Borg. The internal state s is set to 0; then, for each ingested byte b, s is rotated by one bit to the left and $\phi(b)$ is added the state. A bitmask is applied to the state in order to decide whether to chunk. The main difference to Borg is that the state is 64 bits long and that the chunking decision requires that the internal state has $n = 22$ trailing zeros, inducing an average chunk size of 4 MiB.

Our key recovery attack only relies on a known plaintext assumption and requires approximately 3 GiB of known data. The main observation is that each value of the internal state s is an $\mathbb{F}_2$-linear combination of the bits in each $\phi(b)$. Let $B = (b_0,\dots,b_{w-1})$ be the w bytes that have been involved in the chunking decision. Each zero bit in the internal state when the chunking occurs corresponds to an equation in (a subset of) the bits of $\phi(b_i)$ for $i \in \{0,\dots,w-1\}$. Since each $\phi(b_i)$ is 64 bits long, it suffices to observe $256\cdot 64 = 16384$ linearly independent equations to recover all 256 values of ϕ. Since each chunk gives information about 22 trailing zeros, this is equivalent to about 745 chunks or 3 GiB of known data. After collecting sufficient data, it is trivial to recover the entire description of ϕ using linear algebra techniques. We implemented and verified this attack in practice.

The attack can be made more efficient in a chosen plaintext setting. Since all values of $\phi(i)$ for $i > 3$ are related to $\phi(0),\dots,\phi(3)$ via a public hash function, it suffices to recover only those first four values. By chunking approximately 46 MiB of data containing only the bytes $0, 1, 2, 3$, the attacker can efficiently recover the entire mapping using the techniques delineated above.

3.4 Restic

Restic's content-defined chunking algorithm is based on evaluating a fixed-size sliding window by mapping it to a polynomial in a finite field $\mathbb{F} := \mathbb{F}_2[x]/(P)$ and checking whether its $n = 20$ least significant coefficients are all equal to 0. Here, key generation samples the polynomial P randomly from the irreducible polynomials of degree $\delta = 53$ in $\mathbb{F}_2[x]$, $P \leftarrow\!\!\$\ \text{Restic.KGen}()$.

Restic uses minL = 512 kiB − w B and maxL = 8 MiB., where $w = 64$ B is the window length, and sets the initial state as $\sigma = 1 \in \mathbb{F}_p[x] \leftarrow \text{Restic.Init}(P)$. For a byte b, let poly(b) $\in \mathbb{F}_2[x]$ be the polynomial of degree at most 7 where coefficients correspond to the bits of b. Ingesting one byte of data at a time and Restic computes $\sigma' = (\sigma \cdot x^8 \bmod x^{512}) + \text{poly}(b) \leftarrow \text{Restic.Update}(P,\sigma,b)$. The chunking decision Restic.Eval(P,σ) is 1 whenever ($\sigma \bmod P$) mod $x^n = 0$ with $n = 20$ and 0 otherwise.

Let V be the 512-dimensional $\mathbb{F}_2$-vector space of polynomials of degree at most 511 and let $\mathcal{F}_P\colon V \to V$ denote the linear map which reduces $\mathbb{F}_2[x]$ polynomials modulo P and then modulo x^n. Every chunk of length less than 8 MiB defines a polynomial equal to 0 after reducing modulo P and subsequently modulo x^n, i.e., it defines an element in the kernel of $\mathcal{F}_P$. Clearly, $\ker(\mathcal{F}_P)$ is a $512-n$ dimensional $\mathbb{F}_2$-vector subspace of V.

Observing $512-n$ linearly independent windows that trigger chunking provides us with an $\mathbb{F}_2$-basis of the kernel subspace. We consider the coefficient matrix where each row corresponds to an element of the basis of $\ker(\mathcal{F}_P)$. The row echelon form of this matrix corresponds to another basis of $\ker(\mathcal{F}_P)$ containing one polynomial of each degree from 20 to 511 once. Thus, we may assume without loss of generality that after observing $512-n$ distinct sliding windows that trigger chunking, we know at least one polynomial in the kernel subspace of degree i for $i = 20,\dots,511$.

Next, we will see how such a basis can be used to iteratively recover coefficients of the secret irreducible polynomial $P = \sum_{i=0}^{\delta} p_i x^i \in \mathbb{F}_2[x]$. As P is irreducible and of degree δ, we know that $p_\delta = p_0 = 1$. Assume now that the $k > 0$ most significant coefficients of P are known and that a polynomial $\gamma = \sum_{i=0}^{\delta+k} \gamma_i x^i$ with $\gamma_i \in \mathbb{F}_2$ of degree $\delta + k$ in the kernel subspace is known. We will show that this can be used to compute the $k+1$ most significant coefficients of P.

Any polynomial γ in the kernel subspace is of the form

$$\gamma = \alpha \cdot x^n + \beta \cdot P, \tag{1}$$

with $\alpha, \beta \in \mathbb{F}_2[x]$, and degrees $\deg(\beta) = \deg(\gamma) - \delta$ and $\deg(\alpha) \le \delta - n - 1$.

Thus, for γ of degree $\delta + k$, we know that β is of degree exactly k, say $\beta = \sum_{i=0}^{k} \beta_i x^i$ with $\beta_k = 1$ and $\beta_i \in \mathbb{F}_2$ for $i = 0,\dots,k-1$.

Consider the product of polynomials

$$\beta \cdot P = \sum_{i=0}^{\delta+k} \left(\sum_{j=0}^{\min(i,k)} \beta_j \cdot p_{i-j} \right) \cdot x^i. \tag{2}$$

As $\deg(\alpha \cdot x^n) < \delta$, the $k+1$ most significant coefficients of $\beta \cdot P$ are the $k+1$ most significant coefficients of γ known to us.

Using this equality and $p_\delta = 1$, we can compute the coefficients $\beta_k,\dots,\beta_1$ with Eq. (2) iteratively from the k most significant bits of γ and the k known bits of P. Namely, we have $\beta_{k-j} = \gamma_{\delta+k-j} + \sum_{\ell=0}^{j-1} \beta_{k-\ell} p_{\delta-j+\ell}$. Finally, we can read off $\beta_0 = \gamma_0$. This equality follows from Eq. (1), $p_0 = 1$ and the observation that αx^n has no constant term.

Having recovered the coefficients of β corresponding to the given polynomial γ, we can compute one more coefficient of P with the identity $p_{\delta-k-1} = \gamma_{\delta-1} + \sum_{\ell=0}^{k-1} \beta_\ell p_{\delta-\ell-1}$. Here, we used again the formula of Eq. (2), that the $(k+1)$th most significant coefficient of $\beta \cdot P$ is equal to $\gamma_{\delta-1}$ and that $\beta_k = 1$.

In summary, observing $512-n$ distinct sliding windows that trigger chunking allows us to compute a basis of $\ker(\mathcal{F}_P)$ with one polynomial of degree i each for $i = 20,\dots,511$. In particular, we may assume that we know polynomials of degrees $\delta+1,\dots,2\delta-1$ in $\ker(\mathcal{F}_P)$.

Given a polynomial in $\ker(\mathcal{F}_P)$ of degree $\delta + k$ as well as the k most significant coefficients of P, we have seen how to recover the $k+1$ most significant coefficients of P. Since the most significant coefficient p_δ of P equals 1, we can use the polynomials from our basis to iteratively apply the procedure and recover the whole polynomial P which is the chunking key.

We implemented and verified the attack outlined above. With an expected chunk size of 1.5 MiB, we expect a known-data complexity of $(512 - 20) \cdot 1.5 < 750$ MiB random data for the attack to work.

3.5 Tarsnap

Contrary to the previously seen content-defined chunking algorithms, Tarsnap uses *variable-sized* windows to break files into smaller pieces.

Let $\mu = 2^{16}$. The chunking key $K = (p, \alpha, \phi) \leftarrow\!\!\$\ \mathsf{Tarsnap.KGen}()$ consists of a prime $p \approx 2^{24} = \mu^{3/2}$, $\alpha \in \mathbb{F}_p$ of order greater than μ and a map $\phi\colon \{0, \dots, 255\} \to \mathbb{F}_p$ which maps bytes to randomly chosen elements in $\mathbb{F}_p$.

In Tarsnap we have minL $= 2^{14} = \mu/4$ B and maxL $= 2^{10} \cdot 3 \cdot 5 \cdot 17$. Let $S_{0,k} := (b_0, \dots, b_k)$ be a slice of data, where each b_i denotes a byte and consider the polynomials

$$P_{S_{j,k}}(x) := \sum_{i=j}^{k} \phi(b_i) x^i.$$

The initialization algorithm $\sigma = (q, T, 0) \leftarrow \mathsf{Tarsnap.Init}(K)$ outputs an empty queue q with 32 positions, an empty hash table T, and a counter initialized to 0. Afterwards, the chunking algorithm starts ingesting $b_0, b_1, \dots$ one byte at a time. The update algorithm, $\mathsf{Tarsnap.Update}(K, \sigma, b_k)$, computes the updated state $\sigma' = (q', T', k)$, where q' is obtained by enqueueing $P_{S_{0,k}}(\alpha)$ in q and dequeueing the first element of q if the queue has reached its maximum length. If an element is dequeued, then it is added to the hash table T, along with the *current* position k (which is the position at which the element was enqueued, plus 32), yielding T'. Otherwise, $T' = T$. Every time a byte is ingested, the counter is increased by one. Then, $\mathsf{Tarsnap.Eval}(K, (q, T, k))$ outputs 1 if $P_{S_{0,k}}(\alpha) \in T$, i.e., if there was an entry in T of the form $(P_{S_{0,i}}, i + 32)$ with $i < k$ such that $P_{S_{0,i}}(\alpha) = P_{S_{0,k}}(\alpha)$, and $k - (i + 32) - 1 < \sqrt{4k - 3}$. We call the required inequality on k and i the *recency condition.*

Note that when $P_{S_{0,i}}(\alpha) = P_{S_{0,k}}(\alpha)$ for some $i < k$, then $P_{S_{0,k}}(\alpha) - P_{S_{0,i}}(\alpha) = \alpha^{k-i} P_{S_{i,k}}(\alpha) = 0$ which implies $P_{S_{i,k}}(\alpha) = 0$. Whenever the recency condition is satisfied, i.e., $k - (i + 32) - 1 < \sqrt{4k - 3}$, the data is chunked and we call $(b_i, \dots, b_k)$ the *chunking relation.*

Due to the queue used, we always have $k - i \geq 33$ and thus chunking relations are always of length at least 33. From maxL and the recency condition, we also get an upper bound of 1014 for the length of a chunking relation. While we cannot in general tell the length of a chunking relation from a given chunk, we know the relation has to be one of the at most $1014 - 33 = 981$ possible options (for shorter chunks we get a lower upper bound).

The average chunk size produced by the algorithm in Tarsnap is $\mu = 2^{16}$ B for all the keys.

In the following, we describe how to recover Tarsnap's chunking key by observing the chunking algorithm's behaviour on a *chosen* plaintext. We start with two observations: the way the secret prime p is chosen in Tarsnap's key generation leaves little entropy and there are only 17 possible values for p. Further, note that $P_{S_{j,k}}(\alpha) = 0$ if and only if $cP_{S_{j,k}}(\alpha) = 0$ for any nonzero $c \in \mathbb{F}_p$. Thus, as long as $\phi(b_i) \neq 0$ for some byte b_i, we can rescale all images of ϕ by $\phi(b_i)^{-1}$. Therefore, it is sufficient to recover ϕ up to a scalar $c \in \mathbb{F}_p$.

Our key recovery attack proceeds in two steps. First, we use the chunking behaviour of the algorithm on a sequence only containing two distinct bytes, say b_0 and b_1, to recover the precise value of p, α and the values of $c\phi(b_0), c\phi(b_1)$ up to an unknown $c \in \mathbb{F}_p$. Second, we retrieve the remaining images of ϕ up to a scalar by observing the chunking behaviour on byte sequences where we expect a sufficient number of resulting chunking relations to contain a mixture of bytes for which we know the image under ϕ (up to scalar) already and one new byte.

For the first step, we provide two variants both relying on a chosen plaintext. Variant 1 describes a solution which partially brute-forces the chunking key, and thus has larger complexity, but requiring relatively little chosen plaintext. Variant 2 provides a more efficient key recovery attack but requires an attacker to be able to choose a larger plaintext to be chunked.

Variant 1 (Bruteforcing relation lengths). Assume we are given the chunks of a file consisting of a random sequence of b_0 and b_1 bytes. As mentioned before, we may assume that $\phi(b_0) = 1$.

Given any two chunks, we can guess their corresponding relation lengths $33 \leq i, j \leq 1014$ out of the 981 possible values. Note that shorter chunking relations are more likely to appear giving us a natural order for guessing the relation lengths.

For each i, j, we compute the polynomials $p_0^{(i)}, p_1^{(i)}, p_0^{(j)}, p_1^{(j)} \in \mathbb{F}_p[x]$ of degree at most i and j, respectively. If our guess is correct, then it holds that

$$\begin{pmatrix} p_0^{(i)}(x) & p_1^{(i)}(x) \\ p_0^{(j)}(x) & p_1^{(j)}(x) \end{pmatrix} \cdot \begin{pmatrix} \phi(b_0) \\ \phi(b_1) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{3}$$

As long as $\phi(b_0)$ and $\phi(b_1)$ are not both zero, we know that α has to be a root of the determinant of the (2×2)-matrix. For all (17) choices of the prime p and a guess for the pair (i, j), we compute all roots of the determinant, a polynomial of degree at most $i + j$ to retrieve candidate values for α. For each candidate tuple (p, i, j, α), computing the right kernel of the matrix provides us with the value for $\phi(b_0)$ and $\phi(b_1)$ up to a constant.

We then check on additional chunks whether the chunking behaviour for a candidate solution $(p, \alpha, \phi(b_0), \phi(b_1))$ is consistent, i.e. there exists a suffix for which the chunking condition is satisfied. Otherwise, we discard the candidate. Under the heuristic that the evaluation of a spurious solutions over all possible chunking relations behaves uniformly at random in $\mathbb{F}_p$ and given that the number of possible lengths of chunking relations is bounded by 981, checking spurious solutions against even a single other chunk will allow us to dismiss a spurious solution for $(p, \alpha, \phi(b_0), \phi(b_1))$ with probability at least $1 - \frac{981}{2^{24}}$.

For each possible (17) value of p and guess for (i, j), we obtain at most $17 \cdot (i + j) < 17 \cdot 2^{11}$ candidates for α, i.e. with three chunks not used to construct the matrix we can expect to filter through all the spurious solutions.

With as little as 0.33 MiB of random data consisting of b_0 and b_1 bytes, we expect to get 5 chunks needed to mount the attack. The time complexity of this variant is then dominated by the computation and filtering of the at most $(i + j) < 2^{11}$ roots of the determinant for each of the $17 \cdot 2^{20}$ possible guesses for p and (i, j), i.e. at worst $\approx 2^{35}$ solutions to check.

Note that this is a worst-case complexity and it is likely that this computation is much faster in the average case. For example, it could be that not all roots of the determinant polynomial lie in $\mathbb{F}_p$, leading to fewer candidates to check. This computation is also embarassingly parallelizable as the resultants can be computed independently for each guess.

Variant 2 (De Bruijn-like sequences). In this variant we construct a file consisting of 0 and 1 bytes such that we can guess the relation lengths more easily at the cost of more chosen plaintext data. We construct the file to have a large number of distinct consecutive subsequences of length 33 preceded by both a 0 and 1 byte in different places of the file. We call such subsequences *conjugate pairs.* Note that there are a total of 2^{33} polynomials of degree 32 with coefficients $\phi(0)$ or $\phi(1)$ as long as $\phi(0) \neq \phi(1)$. Under the heuristic assumption that these polynomials behave randomly in $\mathbb{F}_p$ when evaluated at α, we expect roughly $\frac{2^{33}}{2^{24}} = 2^9$ of them to evaluate to 0 and thus trigger chunking with a short chunking relation of length 33. Unfortunately, given the chunks of a file we cannot tell which of them correspond to such short chunking relations. However, if we observe that in two places a chunk ends in the same 33-byte suffix preceded by distinct bytes, then the probability is large that the chunking relation is the 33-byte suffix. Indeed, the probability is low that two fixed byte sequences with a shared 33-byte suffix both satisfy the chunking condition under the condition that the evaluation of the polynomial corresponding to the shared suffix is non-zero. Assuming independence between the entries of both sequences of bytes and a uniform distribution of the evaluation of polynomials corresponding to all the suffixes at α, we can upper bound the conditional probability by the product of a byte sequence having a valid chunking relation as suffix divided by the probability that the shared suffix of length 33-bytes is non-zero. Thus, the sought after probability is bounded by $\left(\frac{981}{2^{24}}\right)^2 \cdot \left(1 - \frac{2^{33}}{2^{24}}\right)^{-1} \approx 2^{-28}$.

By modifying techniques originally used to construct de Bruijn sequences [17], we can build a sequence of length $9 \cdot 2^{26}$ bytes (or 576 MiB) containing only 0 and 1 byte entries such that there are $t = 2^{26}$ distinct conjugate pairs of length $u = 34$ appearing in the file. We refer to Appendix A for the details of how to construct such an (n, u, t) sequence, for relatively small n. Given 2^{26} distinct conjugate pairs of length 34 in the file, we expect roughly $\frac{2^{26}}{2^{24}} = 2^2$ of these subsequences to trigger chunking in both locations because of the common 33-byte suffix. Note that this estimate is a little optimistic as some (parts of) subsequences might fall into the minimum chunk length not considered by the chunking algorithm. Experimentally, we observe that 12% of the conjugate pairs in our house 576 MiB sequence are at a distance smaller than the minimum chunk length, rendering them ineffective. Still, the expected number of subsequences that trigger chunking remains at about $(1 - 0.12) \cdot 2^2 \approx 3.5$ which is large enough for an acceptable success rate.

After observing the chunking behaviour on our file, we assume that if chunking happened after both sequences of a conjugate pair, then their common suffix is the chunking relation of length 33. Observing two such pairs provides us with two chunking relations, i.e., we can compute the polynomials $p_0^{(33)}, p_1^{(33)}, p_0^{(33)}, p_1^{(33)} \in \mathbb{F}_p[x]$ of degree at most 32 as in Eq. 3.

As in Variant 1, as long as $\phi(0)$ and $\phi(1)$ are not both zero, we know that α has to be a root of the determinant of the first matrix. For all (17) choices of the prime p we compute the roots of the determinant, a polynomial of degree at most 64, and compute the corresponding right kernel of the matrix to retrieve a value for $\phi(0)$ and $\phi(1)$ up to a constant. Using the remaining chunks given, we can sieve through the at most $17 \cdot 64$ potential solutions for p, α, $\phi(0)$ and $\phi(1)$ and dismiss potential solutions for which no valid suffix that can act as a chunking relation can be found for some chunk. Under the same heuristic as in Variant 1, we can dismiss any spurious solution with probability at least $1 - \frac{981}{2^{24}}$ even when checking just against a single additional chunk.

Recovering the remaining chunking key. Having recovered p, α as well as $\phi(0)$ and $\phi(1)$ up to a scalar, we are left to recover the remaining images of ϕ up to the same scalar to obtain an equivalent chunking key. We craft our chosen plaintext such that for each $i = 2, \ldots, 255$ we expect 3-4 chunks containing bytes i and bytes for which we know the image, e.g. 0 and 1. For each chunk and all possible lengths of chunking relations, we compute the potential image of $\phi(i)$ by solving the corresponding linear system. This way we obtain one possible image for each guess of the length and chunk. Under the assumption that spurious candidate solutions for $\phi(i)$ are randomly distributed in $\mathbb{F}_p$ and because chunking relations can be at most of length 990 (albeit the median is 216 and average 240), we expect a three-way collision of spurious solutions to happen in the worst possible setting only with a probability less than $\frac{990^3}{6 \cdot 2^{48}} \approx 2^{-20}$. Thus, in the intersection of all the candidates for the 3-4 chunks, we expect only one possible image $\phi(i)$ to be left. The time complexity of this second step of the attack is negligible and given the average chunk size of 2^{16} B, we need to choose a plaintext roughly of size $254 \cdot 4 \cdot 2^{16}$ B ≈ 63 MiB.

Using Variant 1, we thus require as little as 64 MiB of chosen-data for the whole attack to work, and approximately 639 MiB of chosen-data using Variant 2 with a 576 MiB payload constructed using techniques of Appendix A. Note that the data requirements could be lowered further if one was willing to accept a non-negligible failure probability.

4 Secure Chunking

All the attacks we have discussed so far are based on the fact that the attacker can predict the chunking behaviour of a file. This is because an adversary can run the chunking algorithm themselves after executing a key-recovery attack.

In this section, we turn to a formal treatment of chunking algorithms, specifically for the class of *fixed-size window* chunking (FSWC) schemes, that is KCDC schemes which make their chunking decision depending only on the last w bytes seen, for some fixed w. We focus on FSWC schemes as they are the most common in practice and the most amenable to formal analysis.

We first define the syntax for a FSWC scheme and introduce our security notion for such schemes. We then provide a provably secure construction based on polynomial hashing and a block cipher. Finally, we provide benchmarks for our implementation.

4.1 FSW Chunking and its Security

Fixed-size window chunking schemes are a special class of KCDC schemes which are characterized by *w-locality*: the decision of whether to cut a chunk of data only depends on the last w bytes.

Definition 4.1 (w-locality). Let w be a positive integer. A chunking scheme Chk is said to have w-locality if:

$$\begin{aligned}&\forall \mathrm{K} \leftarrow\!\!\$\ \mathsf{Chk.KGen}(), \sigma \leftarrow \mathsf{Chk.Init}(\mathrm{K}),\\ &\forall s_1, s_2 \in \{0,\ldots,255\}^*, s_1[-w:] = s_2[-w:] :\\ &\quad \mathsf{Chk.Eval}(\mathrm{K}, \mathsf{Chk.Update}^*(K,\sigma,s_1)) =\\ &\quad \mathsf{Chk.Eval}(\mathrm{K}, \mathsf{Chk.Update}^*(K,\sigma,s_2)).\end{aligned}$$

Definition 4.2 (Fixed-size window chunking scheme). Let w be a positive integer. A w-FSWC scheme Chk is a KCDC scheme which additionally fulfills w-locality.

In this setting, we can define a "stateless" version of the chunking decision algorithm, which takes a data window of w bytes and returns the chunking decision for that window. This algorithm, denoted $\mathsf{Chk.WindowEval}(K, W)$, with $|W| = w$, is equivalent to an initialization with key K, a call to Update^* with the window W, and a final chunking decision evaluation.

Security notions. We provide two notions for the security of a chunking algorithm. The first notion, FSWC-FtG ("Find-then-Guess", following notation from [3]) intuitively captures security against fingerprinting. Informally, an attacker that sees the chunking behaviour of known data should not be able to predict the chunking behaviour on unseen data.

The FSWC-FtG game, formalized in Fig. 2, proceeds in two phases: in the Find phase, the adversary gets access to an oracle that provides the chunking decision for arbitrary windows. In the Guess phase, the adversary provides a window (different from any window queried in the Find phase) and gets a chunking decision made by either the chunking algorithm or by sampling a random coin. The random coin is sampled following the Bernoulli distribution $\mathrm{Ber}(\lambda^{-1})$, i.e., taking value 1 with probabiltiy $\frac{1}{\lambda}$, where λ is a parameter of the game. This parameter corresponds to the expected chunk length of the scheme: for arbitrary data input and randomly-chosen key, Chk.Update is expected to output 1 after processing on average λ many bytes. Since the random coin is independent of the underlying data, this intuitively means that the chunking pattern does not reveal more information to the adversary than what it already knows.

We define the advantage of an adversary $\mathcal{A}$ in the FSWC-FtG game against Chk as

$$\mathrm{Adv}^{\text{FSWC-FtG}}_{\mathsf{Chk},\lambda}(\mathcal{A}) = \left|\Pr[\text{FSWC-FtG}(\mathcal{A}, \mathsf{Chk}, \lambda)] - \frac{1}{2}\right|.$$

The second notion we define, FSWC-RoR ("Real-or-Random", in Fig. 3), is a more standard, "indistinguishability-type" notion of security. Informally, an adversary should not be able to distinguish between chunking decisions made by Chk or made by sampling a random coin. We model this as a game in which the adversary can ask a chunking oracle for the chunking decision associated to an arbitrary window of bytes. This decision is computed according to the value of a secret bit b: if $b = 0$, the oracle replies by using the chunking algorithm, otherwise by tossing an accordingly biased random coin, again following $\mathrm{Ber}(\lambda^{-1})$ for parameter λ.

```
Game FSWC-FtG($\mathcal{A}$, Chk, $\lambda$):
  $b \leftarrow\!\!\$ \{0,1\}$
  $S \leftarrow \emptyset$
  $K \leftarrow\!\!\$ \mathsf{Chk.KGen}()$
  $(W^*, \mathsf{st}) \leftarrow\!\!\$ \mathcal{A}^{O_{\mathsf{find}}}(\mathsf{find})$
  If $b = 0$:
      $d \leftarrow \mathsf{Chk.WindowEval}(K, W^*)$
  Else:
      $d \leftarrow\!\!\$ \mathrm{Ber}(\lambda^{-1})$
  $b' \leftarrow\!\!\$ \mathcal{A}(\mathsf{guess}, d, \mathsf{st})$
  Return $b' = b \wedge W^* \notin S$

Oracle $O_{\mathsf{find}}(W)$:
  $S \leftarrow S \cup \{W\}$
  $d \leftarrow \mathsf{Chk.WindowEval}(K, W)$
  Return $d$
```

Figure 2: The FSWC-FtG security game for a FSW chunking algorithm Chk. Code in gray prevents trivial wins.

```
Game FSWC-RoR($\mathcal{A}$, Chk, $\lambda$):
  $b \leftarrow\!\!\$ \{0,1\}$
  $K \leftarrow\!\!\$ \mathsf{Chk.KGen}()$
  $T \leftarrow \emptyset$
  $b' \leftarrow\!\!\$ \mathcal{A}^{O}$
  Return $b' = b$

Oracle $O(W)$:
  If $T[W] \neq \perp$:
      Return $T[W]$
  If $b = 0$:
      $d \leftarrow \mathsf{Chk.WindowEval}(K, W)$
  Else:
      $d \leftarrow\!\!\$ \mathrm{Ber}(\lambda^{-1})$
  $T[W] \leftarrow d$
  Return $d$
```

Figure 3: The FSWC-RoR security game for a FSW chunking algorithm Chk. Code in gray prevents trivial wins.

We define the advantage of an adversary $\mathcal{A}$ in the FSWC-RoR game against Chk as

$$\mathrm{Adv}^{\text{FSWC-RoR}}_{\mathsf{Chk},\lambda}(\mathcal{A}) = \left|\Pr[\text{FSWC-RoR}(\mathcal{A}, \mathsf{Chk}, \lambda)] - \frac{1}{2}\right|.$$

While these two notions seem different, we prove that they are asymptotically equivalent in Appendix B: FSWC-RoR tightly implies FSWC-FtG, vice versa losing a factor of number of O_{find} queries. From here one, we thus only use the FSWC-RoR notion, which is simpler to work with.

It is also easy to see that the FSWC-RoR notion induces a simulation-based notion, where the leakage given to the simulator is equivalent to the table T in the game.

From chunking decisions to chunks. To conclude our analysis, we prove that the FSWC-RoR security of a chunking scheme implies that the lengths of chunks are computationally indistinguishable from a (shifted) geometric distribution when using the canonical chunking implementation of Fig. 1.

Theorem 4.3 (FSWC-RoR Implies Geometric Distribution of Chunk Lengths). *Let Chk be a FSWC-RoR-secure w-FSWC scheme for parameter λ. Let minL, maxL be positive integers, with maxL > minL. Then, $\forall K \leftarrow\!\!\$$ Chk.KGen() no efficient adversary $\mathcal{A}$ choosing data $\in \{0,\ldots,255\}^*$, can distinguish the sequence of lengths of* *the chunks output by $f_{Chk}(data, minL, maxL, K)$, (except the last one) from a sequence of random variables of the same length $(L_i)_i$ where $L_i = \min(minL + X_i, maxL)$ with $X_i \sim Geom(\lambda^{-1})$.*

Proof sketch. Let O be an oracle that either implements f_{Chk} or simulates it by tossing a random coin sampled from $\mathrm{Ber}(\lambda^{-1})$ whenever f_{Chk} calls Chk.Eval. Note that the latter case is equivalent to having lengths sampled as $\min(\mathrm{minL} + X_i, \mathrm{maxL})$ with $X_i \sim \mathrm{Geom}(\lambda^{-1})$. Let $\mathcal{A}$ be an adversary distinguishing between the two cases. We construct an adversary $\mathcal{B}$ for the FSWC-RoR security game by simulating f_{Chk}: $\mathcal{B}$ skips the first minL bytes and then starts requesting chunking decisions for every window in the data chosen by $\mathcal{A}$. Whenever $\mathcal{B}$'s oracle returns 1 or no more bytes are left, $\mathcal{B}$ returns all bytes processed up to that point. Finally, $\mathcal{B}$ outputs the same bit as $\mathcal{A}$. The adversary $\mathcal{B}$ perfectly simulates either the real or random world for $\mathcal{A}$, depending on the hidden bit of its oracle. It can be clearly seen that the advantage of $\mathcal{B}$ in the FSWC-RoR is the same as the advantage of $\mathcal{A}$ in its own game. □

On folklore mitigations. All our attacks from Section 3 achieve key recovery and most of them rely only on a known-plaintext assumption. This shows that the folklore mitigations fail to provide security according to a much weaker security notion than the one we propose here. Indeed, our notions are in a chosen-plaintext model and set a much weaker winning condition for the adversary: it needs only to distinguish between real chunking behaviour and plaintext-independent decisions, rather than having to recover the key. A successful key recovery attack immediately allows an adversary to win in our security games with overwhelming probability.

Scope of security guarantees. Our security notions captures that the leakage from individual chunks should be minimal. Within the overall system a KCDC scheme is deployed, there might be other leakage though which needs to be considered by system designers. For example, our security definition from Theorem 4.3 explicitly does not capture the leakage given by the length of the last chunk. We discuss the scope of security guarantees provided by our notions in Section 5, within the bigger context of attacks and mitigations.

4.2 Poly-hashing-then-Encrypt Construction

We now introduce our FWS chunking construction Chk-PHTE ("Poly-hashing-then-Encrypt") which uses a polynomial hash for the (rolling) processing of data windows and a block cipher for the chunking decision evaluation.

Let $\mathbb{F}$ be a finite field of size p. Let $\lambda = 2^\ell$ be a power of 2 and let $w \in \mathbb{N}$.[1] Let E be a PRF-secure block cipher with block length n (fulfilling $n > \log_2 p$ and $n > \log_2 \lambda$) and key length k. Let $\mathsf{btof}\colon \{0,1\}^8 \to \mathbb{F}$ and $\mathsf{ftob}\colon \mathbb{F} \to \{0,1\}^n$ be injective encoding functions mapping bytes to field elements and field elements to bit-strings of length n, respectively. Our formal w-FSWC algorithm Chk-PHTE is depicted in Fig. 4.

Informally, each algorithm does the following:

- $\mathsf{KGen}()$: samples keys for the polynomial hashing and for the block cipher.
- $\mathsf{Init}(K)$: initializes the current polynomial evaluation to $0 \in \mathbb{F}$ and the current window to an empty list.
- $\mathsf{Update}(K, B)$: updates the current polynomial evaluation by removing the oldest byte (which is the term of degree $w-1$), shifting the polynomial by multiplying with K_{poly} and adding the new byte, which will be the constant term. The window is updated accordingly and has the invariant that it contains the last w bytes seen, from the oldest to the newest.
- $\mathsf{Eval}(K, \sigma)$: evaluates the block cipher on the current polynomial evaluation and checks whether the output has more than ℓ trailing 0s, returning 1 in that case, otherwise 0.

It is immediate to see that Chk-PHTE is of FSW type: WindowEval is equivalent to the Init algorithm followed by w Update calls with the bytes in the window, followed by a call to Eval. Inspection of the Update algorithm also establishes w-locality: everytime the queue is updated, the oldest byte is removed and the new byte is added. Thus, the state does not include any byte that is more than w bytes distant from the newest byte and uses only those that are in the state to make a decision.

Our Chk-PHTE construction permits efficient updating, as it requires a constant number of field operations to update the state of the chunker.

Before proving security, we establish the following lemma.

Lemma 4.4 (Polynomial hashing is Universal). *Let $\mathbb{F}$ be a finite field. Let w be a fixed positive integer. Let $u_K((\beta_0, \ldots, \beta_{w-1})) = \sum_{i=0}^{w-1} \beta^i \cdot K^{w-1-i}$. Then, u_K is a $\frac{w-1}{|\mathbb{F}|}$-universal hash function family for $K \in \mathbb{F}$.*

Proof. We seek to upper bound the probability that an adversary finds two messages $(\beta_0, \ldots, \beta_{w-1})$ and $(\beta'_0, \ldots, \beta'_{w-1})$ such that their hashes collide for a randomly chosen K. Consider the two polynomials $U(x) = \sum_{i=0}^{w-1} \beta_i \cdot x^{w-1-i}$ and $V(x) = \sum_{i=0}^{w-1} \beta'_i \cdot x^{w-1-i}$. They are both of degree at most $w-1$, and so is their difference $U(x) - V(x)$. There are at most $w-1$ values of x for which the difference polynomial evaluates to 0, and so the probability that K is a root is $\frac{w-1}{|\mathbb{F}|}$, by the Schwartz-Zippel lemma. □

We now show that our construction achieves FSWC-RoR security for parameter λ, assuming PRF security of the block cipher E.

Theorem 4.5 (Chk-PHTE is FSWC-RoR-secure). *Let Chk-PHTE be the w-FSWC algorithm in Fig. 4. For every adversary $\mathcal{A}$ against the FSWC-RoR security (with parameter λ) of Chk-PHTE making at most q queries, there exist an adversary $\mathcal{B}$ running in approximately the same time as $\mathcal{A}$ such that*

$$Adv^{FSWC\text{-}RoR}_{Chk}(\mathcal{A}) \leq Adv^{PRF}_{E}(\mathcal{B}) + \frac{q^2(w-1)}{2|\mathbb{F}|} \cdot \frac{1}{\lambda},$$

where $\mathcal{B}$ makes at most q queries to its PRF oracle.

Proof. We proceed via the following a sequence of games (see Fig. 7 in Appendix C for a code-based representation).

Game 0: The original game.

Game 1: We replace the block cipher E with a random function f.

Game 2: We next replace the outputs of f with uniformly random ones. This gives us the intermediate game G'_2 where the output is chosen as $c \leftarrow\$ \{0,1\}^\ell$ (we do not need to sample n bits, since only the last ℓ bits are considered). Thus, the probability of returning 1 is $2^{-\ell} = \lambda^{-1}$. In every other case we return 0. This is exactly

[1]Having λ be power of 2 leads to natural chunk sizes and slightly simplifies the analysis; other values of λ are also possible.

```
$\mathsf{KGen}()$:
    $K_{\text{poly}} \leftarrow\$ \mathbb{F}$
    $K_{\mathsf{E}} \leftarrow\$ \{0,1\}^k$
    Return $(K_{\text{poly}}, K_{\mathsf{E}})$

$\mathsf{Init}(K)$:
    $\sigma \leftarrow (0 \in \mathbb{F}, [])$
    Return $\sigma$
```

```
$\mathsf{Update}(K, B)$:
    $(K_{\text{poly}}, \cdot) \leftarrow K$
    $(u, W) \leftarrow \sigma$
    If $|W| = w$:
        $B_{\text{old}} \leftarrow W[0]$
        $W \leftarrow W[1:]$
    Else: $B_{\text{old}} \leftarrow 0$
    $u \leftarrow u - \mathsf{btof}(B_{\text{old}}) \cdot K_{\text{poly}}^{w-1}$
    $u \leftarrow u \cdot K_{\text{poly}} + \mathsf{btof}(B)$
    $W.append(B)$
    Return $(u, W)$
```

```
$\mathsf{Eval}(K, \sigma)$:
    $(\cdot, K_{\mathsf{E}}) \leftarrow K$
    $(u, \cdot) \leftarrow \sigma$
    $c \leftarrow \mathsf{E}(K_{\mathsf{E}}, \mathsf{ftob}(u))$
    If $c \mod \lambda = 0$:
        Return 1
    Else:
        Return 0
```

```
$\mathsf{WindowEval}(K, W)$:
    $(K_{\text{poly}}, K_{\mathsf{E}}) \leftarrow K$
    $(B_0, \ldots, B_{w-1}) \leftarrow W$
    $u \leftarrow \sum_{i=0}^{w-1} \mathsf{btof}(B_i) \cdot K_{\text{poly}}^{w-1-i}$
    $c \leftarrow \mathsf{E}(K_{\mathsf{E}}, \mathsf{ftob}(u))$
    If $c \mod \lambda = 0$: Return 1
    Else: Return 0
```

Figure 4: Our w-FSW chunking construction Chk-PHTE, based on polynomial hashing over $\mathbb{F}$ and a block cipher E.

equivalent to sampling a decision d from a Bernoulli distribution with parameter λ^{-1}. Note that this makes G_2' equivalent to G_2, so we consider only the latter.

We now rewrite the advantage of the adversary in the FSWC-RoR game:

$$\mathrm{Adv}^{\text{FSWC-RoR}}_{\text{Chk}}(\mathcal{A}) = \left|\Pr[G_0] - \frac{1}{2}\right|$$

$$\leq |\Pr[G_0] - \Pr[G_1]| + |\Pr[G_1] - \Pr[G_2]| + \left|\Pr[G_2] - \frac{1}{2}\right|$$

Clearly, $\Pr[G_2]$ is $\frac{1}{2}$, since the oracle response d is sampled from a Bernoulli distribution, independent of the secret bit b. The transition from G_0 to G_1 is standard and the term $|\Pr[G_0]-\Pr[G_1]|$ is bounded by the advantage of an adversary $\mathcal{B}$ against the PRF security of E. It remains to bound the term $|\Pr[G_1] - \Pr[G_2]|$.

Note that G_1 and G_2 are identical to the adversary until the following "bad event" in G_2: the adversary queries windows $W_0, \ldots, W_{q-1}$, (1) there exist two windows W_i, W_j such that the corresponding values of u (u_i and u_j) are equal, but (2) the decision bits d_i and d_j are different. Since in G_2 the value of d is independent of the value of u, we can compute the probability of the bad event as the product of the probability of (1) and (2) separately.

For (1), the probability that $u_i = u_j$ is at most $\frac{w-1}{|\mathbb{F}|}$, by Lemma 4.4. Over all q queries, the probability that there exist two queries W_i and W_j such that $u_i = u_j$ is then at most $\frac{q^2(w-1)}{2|\mathbb{F}|}$, by union bound. For (2), the probability that two bits d_i and d_j, both sampled from $\mathrm{Ber}(\lambda^{-1})$, are different is $\lambda^{-1}(1-\lambda^{-1}) \leq \lambda^{-1}$.

By Lemma 2.1, with Z being the bad event, we have that:

$$|\Pr[G_1] - \Pr[G_2]| \leq \frac{q^2(w-1)}{2|\mathbb{F}|} \cdot \lambda^{-1}$$

Plugging in the above bounds in the rewritten advantage term yields the claim. □

4.3 Patching Restic and Benchmarking

In the proof, the only requirement we need from the polynomial hashing is that it is universal. As a consequence, we can plug any UHF with with a rolling property in place of the polynomial hashing and obtain another FSWC-RoR-secure chunking scheme. In Appendix D, we show that Restic's chunking scheme is also a UHF, making it a suitable candidate. In fact, we have implemented a patch for Restic that adds a block cipher evaluation to the state and checks the chunking condition on the block cipher output.[2]

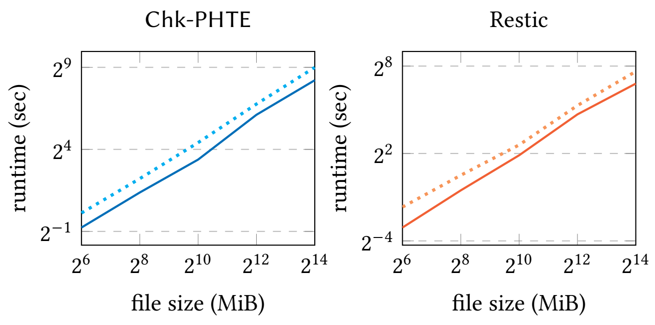

Figure 5: Runtime of our Chk-PHTE implementation and of Restic. Solid lines represent the runtime of the respective chunking algorithm without applying a block cipher, while dotted lines are with AES. Slowdown for Chk-PHTE is between 57% and 104%. For Restic it is between 53% and 165%.

In Fig. 5, we present benchmarks for both our implementation of Chk-PHTE and of Restic. The benchmarks were run on a single thread on a commodity laptop with an Intel Core i7-1260P, running at 3GHz. All benchmarks exclusively target the performance of the chunking scheme, rather than the entire backup process.

Our fairly unoptimized implementation of Chk-PHTE incurs a slowdown between 57% and 104% in our measurements when applying the block cipher as opposed to checking the chunking condition on the result of ftob directly. The runtime of Restic with our patches similarly is 53%–165% slower than the original Restic.

5 Discussion

Real-world applicability of our attacks on backup systems. In Section 3, we have presented efficient key recovery attacks against KCDC schemes as deployed in the wild. However, (K)CDC schemes are typically part of larger systems and the information

[2]Both our implementation and the Restic patch are anonymously available at https://figshare.com/s/6afabccf228ccf79cb5f.

required to accomplish our key-recovery attacks – specifically, the length of chunks of a known or chosen file – may not be available to an attacker. Our attacks in particular rely on noiseless information about the length of chunks; uncertainty (e.g., due to padding) makes exploitation more challenging. As a main obstacle, compression, which is common in backup systems, introduces uncertainty in the length of chunks. Still in the context of backup systems, clients may upload multiple encrypted chunks together, which makes it difficult to discern chunk boundaries. Finally, the order of chunks must be known to the adversary.

Nonetheless, these obstacles do not compensate for a lack of mitigations. For example, Restic only provides compression from version 0.14.0 onwards [26] (released in August 2022) but does not retroactively compress previously uploaded data. Moreover, data which has already been compressed resists further compression, rendering the effect of compression on chunk length negligible. We have created a script that can reliably recover the original chunk lengths from the length of chunks compressed using zlib (used by Tarsnap) and zstd (used by Restic), provided that the plaintext has been compressed prior to processing.

Restic also tries to conceal chunk boundaries by concatenating multiple chunks together to obtain the so-called *pack format* [27]. However, the attacker can still infer the number of chunks by observing the length of the pack header, which is a known constant plus a fixed number per chunk. Assuming the chunk order in the pack matches the original file, the attacker can infer the boundary of the last chunk in the pack, reducing our attack's efficiency but not its feasibility. While the order of chunks in the pack is non-deterministic, due to Restic's usage of multithreading when creating pack files, in single-core systems or systems where Restic is configured to use a single thread, the order of chunks in the pack becomes deterministic, making our attacks applicable. We have tested our attack against Restic in this configuration (no compression, single-threaded backup) and successfully recovered the chunking key.

As another example of a system where our attacks are applicable, Tarsnap sends chunks to the server one at a time and in a deterministic order, immediately exposing chunk lengths to the adversary. One obstacle is that, along with file chunks, Tarsnap also sends chunks composed of metadata which, from the point of view of the server, are indistinguishable from file chunks. This can be bypassed by using smaller payloads that do not incur a metadata chunk until the last chunk is sent, or by splitting the payload into multiple files, which have to be uploaded separately (i.e., with individual calls to the `tarsnap` command). In particular, the first variant of our attack against Tarsnap is immediately applicable in this scenario, while the payload to recover the chunking key ϕ requires splitting the payload into two files.

Another instance where we completed the attack is Borg, which, in the default configuration, stores chunks in a deterministic order on the server without additional padding. We have successfully recovered the chunking key from a Borg repository.

Although the aforementioned obstacles generally frustrate our attack efforts, note that we target the security of content-defined chunking as a general primitive, rather than in these specific systems. Any system that wishes to use KCDC schemes must consider their leakage. Overall, our results should be compared to the ones for compression side-channel attacks. Both require some observable "signal" that is visible to the attacker, such as the length of chunks or the length of compressed data, respectively. Depending on the system, this signal may be noisy or not visible to the attacker. Still, this does not guarantee the absence of leakage from the primitive.

We proceed to put this in more precise terms. Assume a sequence of chunks $(C_i)_{i=1,2,\ldots}$ is created and post-processed in a way that leaves ℓ_i possibilities for the length of chunk C_i. Any attack that requires N chunks to be known exactly will have to bruteforce through $\prod_{i=1}^{N} \ell_i$ possibilities. Since this grows exponentially with N, most attacks become infeasible in the face of even a small uncertainty. This is the case for Restic, Duplicacy, and Bupstash. For attacks that, instead, only require few chunks (one in the case of Borg, two for Tarsnap), the uncertainty might not be enough to prevent attacks. In fact, even if the uncertainty leaves a relatively large number of possibilities for the chunking key, running the attack multiple times allows discarding candidates very quickly.

Note also that all our attacks specifically target key recovery, leaving open the possibility of more effective attacks that allow an attacker to fingerprint without recovering the entire key. For example, in Restic the attacker can recover information about some chunk boundaries with a basis of only a subspace of the entire kernel. In fact, the adversary will be able to recover all the chunk boundaries where the chunking polynomial lies in the recovered subspace.

Interpreting our security notion. In Section 4, we have proposed a new security notion for CDC schemes. It remains to discuss the real-world guarantees this notion provides, as well as its inherent limitations.

Our security notion resembles the ones for deterministic encryption [30] in that both provide "indistinguishable from random" guarantees as long as the adversary chooses distinct messages. Much like deterministic encryption, KCDC schemes will always reveal patterns for identical chunks. This is inherent to KCDC schemes: they must be deterministic to allow subsequent deduplication.

In other words, our security notion guarantees the least possible leakage *when looking at individual chunks*. Indeed, we have proved that the chunk lengths created by a FSWC-RoR KCDC scheme are computationally indistinguishable from lengths sampled from a geometric distribution independently of the plaintext. However, our security notion does not prevent an adversary from learning information about the plaintexts by looking at *all* the chunks in a file, for example, because similarity patterns within the file are revealed by duplicated chunks or because the adversary can infer the length of the file by looking at the length of the chunks.

In practice, developers and protocol designers should be aware of the inherent limitations of KCDC schemes, as highlighted above, and design their systems accordingly. For example, using suitable padding prior to chunking can hide the total file length, while hiding deduplication patterns within and across files may prevent an adversary from learning additional information about the plaintext. We propose formally analyzing the security guarantees of complete data deduplication systems built from KCDC schemes operating alongside chunk deduplication and other system-level operations as a challenging topic for future work. The existing formal work on file deduplication such as [4, 7] may serve as a good starting point. Such a system-wide analysis should also capture alternative mitigations such as padding, which we further discuss below.

The cost of mitigations. Mitigating fingerprinting attacks in a cryptographically principled way does not come for free. While the polynomial evaluation of our Chk-PHTE construction is cheap, the evaluation of a block cipher like AES remains expensive in comparison, even when harnessing AES-NI instructions.

To assess the impact of patching existing systems, we have implemented a patch for Restic that applies a block cipher to the state polynomial, which we presented in Section 4.3. While the performance overhead is noticeable, we deem it reasonable for obtaining a cryptographically sound KCDC solution, especially for security-conscious users.

Alternative mitigations. The performance overhead of our construction raises the question of whether there are other mitigations that can be applied to KCDC schemes. As previously discussed, the cost of an attack in the presence of uncertainty grows exponentially with the number of chunks N required by the attack. Then, *under the assumption that the best attack requires exact knowledge of the length of at least N chunks*, obfuscating the length of each chunk appropriately could be a feasible mitigation. This could be done, for example, by applying a padding to each chunk. A more refined approach would be applying length-hiding padding to each chunk, or encrypting each chunk using a length-hiding encryption scheme [24].

While such an approach would thwart most of our current attacks, as is the case of Duplicacy, Restic, and Bupstash, it would still leave the chunker insecure within our formal model. Furthermore, since we have not proved optimality of our attacks, the possibility is still open that new attacks with a lower chunk requirement exist. Indeed, cryptography has a long history that supports the truism that "attacks only get better".

Furthermore, these approaches are not a panacea. In the particular case of random-length padding, if the adversary is given the possibility of repeatedly observing the same chunk padded randomly each time, it might be able to cancel the padding statistically. One example of this approach is the Borg Obfuscate pseudo-compressor which adds random padding. Even worse, since the attack on Borg only requires one chunk, most configurations of Borg's padding scheme are not aggressive enough to prevent our attacks.

At the same time, these mitigations come at a cost: clearly, padding to the maximum chunk size provides zero bits of information to the attacker, but it also massively increases the storage size of the backup. For Borg, the storage size can increase massively depending on the user configuration, with a 100-fold length increase in some cases. The solution chosen by Duplicity is to pad to the next multiple of 256 bytes, which incurs an overhead of at most 255 bytes per chunk. Even optimal length-hiding padding such as in [22] will incur a storage cost proportional at least to the logarithm of the maximum size of the chunk, while still leaking some bits of information. Concretely, the scheme of [22] leaks $O(\log \log M)$ bits, where M is the maximum length of a plaintext.

Again, we reiterate that these mitigations only work if there exist no attacks (significantly) better in terms of the number of chunks required than the one we present, which is still an open question.

6 Conclusion

Content-defined chunking (CDC) is an approach that is ubiquitous in applications using deduplication. Fingerprinting attacks against CDC schemes are a threat that has been recognised by developers, but the approaches they have taken to defend against such attacks have not been analysed before.

On the destructive side, we showcased key-recovery attacks against five distinct keyed CDC (KCDC) schemes as found in the wild. We also showed that the attacks extend from the isolated KCDC schemes to the full system for Borg, Restic and Tarsnap.

On the constructive side, we proposed formal syntax and security notions for KCDC schemes. We also provided a provably secure construction, which we implemented and benchmarked. We developed a provably secure patch for Restic, one of the most popular backup tools, and benchmarked its performance overhead.

Our work highlights that currently deployed approaches to making KCDC resistant to fingerprinting attacks are inadequate. It also provides a way forward for developers, albeit one with a performance penalty. Our work also points to the need for further analysis of the security guarantees provided by chunk-based deduplication systems in general, both from an attack perspective as well as from the viewpoint of formal foundations.

References

[1] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. 2019. The General Sieve Kernel and New Records in Lattice Reduction. In *EUROCRYPT 2019, Part II (LNCS, Vol. 11477)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Cham, 717–746. https://doi.org/10.1007/978-3-030-17656-3_25

[2] Boris Alexeev, Colin Percival, and Yan X Zhang. 2025. Chunking Attacks on File Backup Services using Content-Defined Chunking. Cryptology ePrint Archive, Paper 2025/532. https://eprint.iacr.org/2025/532

[3] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. 1997. A Concrete Security Treatment of Symmetric Encryption. In *38th FOCS*. IEEE Computer Society Press, 394–403. https://doi.org/10.1109/SFCS.1997.646128

[4] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. 2013. Message-Locked Encryption and Secure Deduplication. In *EUROCRYPT 2013 (LNCS, Vol. 7881)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Berlin, Heidelberg, 296–312. https://doi.org/10.1007/978-3-642-38348-9_18

[5] Dan Boneh and Victor Shoup. 2023. A Graduate Course in Applied Cryptography. https://toc.cryptobook.us/ Version 0.6. Accessed on 7 Jan 2025.

[6] Borg. [n. d.]. Buzhash Chunker. https://borgbackup.readthedocs.io/en/stable/internals/data-structures.html#buzhash-chunker. Accessed on 18 Jul 2023.

[7] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, Håvard Raddum, and Mohsen Toorani. 2018. Security Notions for Cloud Storage and Deduplication. In *Provable Security - 12th International Conference, ProvSec 2018, Jeju, South Korea, October 25-28, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11192)*, Joonsang Baek, Willy Susilo, and Jongkil Kim (Eds.). Springer, 347–365. https://doi.org/10.1007/978-3-030-01446-9_20

[8] Bupstash. [n. d.]. Bupstash Github. https://github.com/andrewchambers/bupstash. Accessed on 16 Nov 2023.

[9] Giovanni Cherubin, Rob Jansen, and Carmela Troncoso. 2022. Online Website Fingerprinting: Evaluating Website Fingerprinting Attacks on Tor in the Real World. In *USENIX Security 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 753–770.

[10] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. 2002. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria, July 2-5, 2002*. IEEE Computer Society, 617–624. https://doi.org/10.1109/ICDCS.2002.1022312

[11] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. 2012. Primary Data Deduplication - Large Scale Study and System Design. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 285–296. https://www.usenix.org/conference/atc12/technical-sessions/presentation/el-shimi

[12] Hugging Face. 2024. Improving Parquet Dedupe on Hugging Face Hub. https://huggingface.co/blog/improve_parquet_dedupe

[13] Kai Gellert, Tibor Jager, Lin Lyu, and Tom Neuschulten. 2022. On Fingerprinting Attacks and Length-Hiding Encryption. In *CT-RSA 2022 (LNCS, Vol. 13161)*, Steven D. Galbraith (Ed.). Springer, Cham, 345–369. https://doi.org/10.1007/978-3-030-95312-6_15

[14] Jamie Hayes and George Danezis. 2016. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *USENIX Security 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 1187–1203.

[15] IPFS. [n. d.]. Unix File System (UnixFS). https://docs.ipfs.tech/concepts/file-systems/#unix-file-system-unixfs

[16] John Kelsey. 2002. Compression and Information Leakage of Plaintext. In *FSE 2002 (LNCS, Vol. 2365)*, Joan Daemen and Vincent Rijmen (Eds.). Springer, Berlin, Heidelberg, 263–276. https://doi.org/10.1007/3-540-45661-9_21

[17] Abraham Lempel. 1970. On a Homomorphism of the de Bruijn Graph and its Applications to the Design of Feedback Shift Registers. *IEEE Trans. Computers* 19, 12 (1970), 1204–1209. https://doi.org/10.1109/T-C.1970.222859

[18] Acrosync LLC. [n. d.]. Duplicacy Homepage. https://duplicacy.com/. Accessed on 16 Nov 2023.

[19] Dutch T. Meyer and William J. Bolosky. 2012. A study of practical deduplication. *ACM Trans. Storage* 7, 4 (2012), 14:1–14:20. https://doi.org/10.1145/2078861.2078864

[20] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, Keith Marzullo and Mahadev Satyanarayanan (Eds.). ACM, 174–187. https://doi.org/10.1145/502034.502052

[21] Fan Ni and Song Jiang. 2019. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-based Deduplication Systems. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 220–232. https://doi.org/10.1145/3357223.3362731

[22] Kirill Nikitin, Ludovic Barman, Wouter Lueks, Matthew Underwood, Jean-Pierre Hubaux, and Bryan Ford. 2019. Reducing Metadata Leakage from Encrypted Files and Communication with PURBs. *PoPETs* 2019, 4 (Oct. 2019), 6–33. https://doi.org/10.2478/popets-2019-0056

[23] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. 2016. Website Fingerprinting at Internet Scale. In *NDSS 2016*. The Internet Society. https://doi.org/10.14722/ndss.2016.23477

[24] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. 2011. Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol. In *ASIACRYPT 2011 (LNCS, Vol. 7073)*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer, Berlin, Heidelberg, 372–389. https://doi.org/10.1007/978-3-642-25385-0_20

[25] Restic. [n. d.]. Chunker Github. https://github.com/restic/chunker. Accessed on 25 Aug 2023.

[26] Restic. 2022. Restic 0.14.0 Released. https://restic.net/blog/2022-08-25/restic-0.14.0-released/

[27] Restic. 2025. Terminology. https://restic.readthedocs.io/en/stable/design.html#pack-format

[28] Hubert Ritzdorf, Ghassan Karame, Claudio Soriente, and Srdjan Capkun. 2016. On Information Leakage in Deduplicated Storage Systems. In *Proceedings of the 2016 ACM on Cloud Computing Security Workshop, CCSW 2016, Vienna, Austria, October 28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Mathias Payer, Stefan Mangard, Elli Androulaki, and Michael K. Reiter (Eds.). ACM, 61–72. https://doi.org/10.1145/2996429.2996432

[29] Juliano Rizzo and Thai Duong. 2012. CRIME attack: CVE-2012-4929. Available from MITRE, CVE-ID CVE-2012-4929. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2012-4929

[30] Phillip Rogaway and Thomas Shrimpton. 2006. A Provable-Security Treatment of the Key-Wrap Problem. In *EUROCRYPT 2006 (LNCS, Vol. 4004)*, Serge Vaudenay (Ed.). Springer, Berlin, Heidelberg, 373–390. https://doi.org/10.1007/11761679_23

[31] Keegan Ryan and Nadia Heninger. 2023. Fast Practical Lattice Reduction Through Iterated Compression. In *CRYPTO 2023, Part III (LNCS, Vol. 14083)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, Cham, 3–36. https://doi.org/10.1007/978-3-031-38548-3_1

[32] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. 2002. Statistical Identification of Encrypted Web Browsing Traffic. In *2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 19–30. https://doi.org/10.1109/SECPRI.2002.1004359

[33] Tong Sun, Bowen Jiang, Borui Li, Jiamei Lv, Yi Gao, and Wei Dong. 2024. SimEnc: A High-Performance Similarity-Preserving Encryption Approach for Deduplication of Encrypted Docker Images. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 615–630. https://www.usenix.org/conference/atc24/presentation/sun

[34] Tarsnap. [n. d.]. Tarsnap Homepage. https://tarsnap.com. Accessed on 16 Nov 2023.

[35] Riot Games Technology. 2019. Supercharging Data Delivery with the New League Patcher. https://technology.riotgames.com/news/supercharging-data-delivery-new-league-patcher

[36] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. 2014. Ddelta: A deduplication-inspired fast delta compression approach. *Perform. Evaluation* 79 (2014), 258–272. https://doi.org/10.1016/J.PEVA.2014.07.016

[37] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: a Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 101–114. https://www.usenix.org/conference/atc16/technical-sessions/presentation/xia

[38] Zuoru Yang, Jingwei Li, and Patrick P. C. Lee. 2022. Secure and Lightweight Deduplicated Storage via Shielded Deduplication-Before-Encryption. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 37–52. https://www.usenix.org/conference/atc22/presentation/yang-zuoru

[39] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. 2015. AE: An Asymmetric Extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*. IEEE, 1337–1345. https://doi.org/10.1109/INFOCOM.2015.7218510

Appendix

A Construction of Conjugate Sequences

Notation. In this section we work with binary sequences, that is sequences with terms from $\{0, 1\}$. Such a sequence $s = s_0, s_1, \ldots$ is said to be periodic with period n if $s_{i+n} = s_i$ for all $i \geq 0$; moreover, s has least period n if n is the smallest positive integer such that

$s_{i+n} = s_i$ for all $i \geq 0$. A sequence of period n is uniquely defined by its first n terms; we write $s = [s_0, s_1, \ldots, s_{n-1}]$ to denote such a sequence.

For $u \geq 1$, let $w_i := (s_i, s_{i+1}, \ldots, s_{i+u-1})$ denote a length u vector produced by taking a subsequence of s. We say that s contains vector w_i as a window at position i. A sequence s of least period n is said to have span u if the n windows w_i, $0 \leq i < n$, are all distinct. A sequence of span u and period 2^u is called a span u de Bruijn sequence; clearly such a sequence has the property that every possible length u binary vector occurs exactly once as a window of s. As an example, $s = [0, 0, 0, 1, 0, 1, 1, 1]$ is a span 3 de Bruijn sequence, while $s = [0, 0, 0, 1, 1, 1]$ is a span 3 sequence of period 6 that is *not* a de Bruijn sequence. A span u de Bruijn sequence can be seen as a Hamiltonian cycle in the de Bruijn graph of order u, this being the graph on 2^u vertices, each vertex labelled by a bit vector $(w_0, \ldots, w_{u-1})$ and each such vertex being connected by directed edges to the vertices $(w_1, \ldots, w_{u-1}, 0)$ and $(w_1, \ldots, w_{u-1}, 1)$. Thus, the span 3 de Bruijn sequence $s = [0, 0, 0, 1, 0, 1, 1, 1]$ corresponds to the cycle of vertices in the corresponding graph beginning with $(0, 0, 0)$, then $(0, 0, 1)$, then $(0, 1, 0)$, and so on, and ending with vertex $(1, 0, 0)$.

Let $u \geq 1$ and let $w = (w_0, w_1, \ldots, w_{u-1})$ be a binary vector of length u. We write $C(w) := (1 + w_0, w_1, \ldots, w_{u-1})$ for the *conjugate* of w [17]. Here '+' denotes addition modulo 2, so $C(w)$ is the length u vector in which the first bit of w is complemented but the remaining bits remain the same as those in w. For example, if $u = 3$ and $w = (0, 0, 0)$ then $C(w) = (1, 0, 0)$.

Problem statement. We are interested in the construction of periodic binary sequences s that, for some u, have the property that they contain as length u windows many distinct conjugate pairs, i.e. pairs of the form w, $C(w)$. Given a target value u and target number of conjugate pairs t, we wish to construct an s containing t distinct conjugate pairs (by which we mean that the $2t$ length u vectors arising from these t pairs are all distinct). We refer to a period n sequence s with the required properties as being an (n, u, t) conjugate sequence. Such a sequence corresponds to a cycle of length n in the order u de Bruijn graph, in which $2t$ of the vertices in the cycle are distinct and form t conjugate pairs. (We do not care if the other vertices in the cycle are distinct from each other and from those arising in the conjugate pairs or not; in our construction, they will be.)

Notice that a de Bruijn sequence s of span u is automatically a $(2^u, u, 2^{u-1})$ conjugate sequence, since for every vector w of length u, both w and $C(w)$ arise at some positions in s; hence such a sequence contains 2^{u-1} conjugate pairs. However, in our attack we do not necessarily need as many as 2^{u-1} conjugate pairs at the target value of u. We also wish to minimise the period n of our sequence, since s will be used as a chosen plaintext, and we wish to minimise its bit-length. Thus, a de Bruijn sequence is not necessarily the ideal solution to our problem. In particular, for our attack, we require $u = 34$ and $t \approx 2^{26}$.

With this notation established, we can now state the main problem that we address in this appendix: given target values u and t construct an (n, u, t) conjugate sequence s whose period n is as small as possible.

There is an additional property that we require: namely, for every conjugate pair $(w, C(w))$ in s, we would like the two vectors w and $C(w)$ to be well-separated in s, namely if w and $C(w)$ appear at positions i, j in s, respectively, then we would like $|i - j| > \delta$ where δ is a fixed design parameter coming from the specifics of the Tarsnap chunking algorithm. Since this property is harder to achieve, we will focus on constructing (n, u, t) conjugate sequences s, and then counting how many of the t conjugate pairs in s satisfy this additional property. Let $t_\delta(s) \leq t$ denote this number. If $t_\delta(s)$ is still sufficiently large, then the sequence s will still be good enough to use in our attack.

We define the efficiency of an (n, u, t) conjugate sequence s to be $2t/n$. This reflects how densely packed in the sequence the conjugate pairs are. A de Bruijn sequence has efficiency $2t/n = 2 \cdot 2^{u-1}/2^u = 1$, the maximum possible value.

(Inverse) Lempel homomorphism. Given a binary sequence s of period n, we define the Lempel homomorphism on s to be the sequence $s_0 + s_1, s_1 + s_2, s_2 + s_3, \ldots$. Note that if s has period n, then its image under the Lempel homomorphism has period dividing n (in fact its period is either n or $n/2$). For example, if $s = [0, 0, 1, 1]$ of period 4, then applying the Lempel homomorphism, we obtain $[0, 1, 0, 1] = [0, 1]$ of period 2. The Lempel homomorphism can also be seen as a graph homomorphism from the de Bruijn graph of order $u + 1$ to the de Bruijn graph of order u [17].

We consider the inverse of the Lempel homomorphism and its effect on sequence period and span. There are two cases:

- If s has least period n and even weight (i.e. $s_0, s_1, \ldots, s_{n-1}$ contains an even number of 1's), then s has two pre-images under the Lempel homomorphism, namely:

  $$0, s_0, s_0 + s_1, s_0 + s_1 + s_2, \ldots$$

  and its complement:

  $$1, 1 + s_0, 1 + s_0 + s_1, 1 + s_0 + s_1 + s_2, \ldots.$$

  Each pre-image also has least period n. Moreover, if s has span u then the two pre-image sequences have span $u + 1$ and together contain $2n$ distinct windows of length $u + 1$.
- If s has least period n and odd weight, then s has a single pre-image under the Lempel homomorphism, namely:

  $$0, s_0, s_0 + s_1, s_0 + s_1 + s_2, \ldots$$

  This pre-image has least period $2n$ and is self-complementary (i.e. its second half is the bitwise complement of the first half). Moreover, if s has span u then the pre-image has span $u + 1$.

For proofs of these results, see [17].

For example, if $s = [0, 0, 0, 1, 1, 1]$ of period 6, span 3 and odd weight, then its pre-image under the Lempel homomorphism is $[0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1]$ of period 12 and span 4. On the other hand, if $s = [0, 0, 0, 0, 1, 1, 1, 1]$ of period 8, span 4 and even weight, then its two pre-images under the Lempel homomorphism are $[0, 0, 0, 0, 0, 1, 0, 1]$ and $[1, 1, 1, 1, 1, 0, 1, 0]$ both of period 8 and having (joint) span 5.

Note that if a sequence s contains $0^u := (0, 0, \ldots, 0)$ as a window, then its pre-image(s) (under the Lempel homomorphism) contain both 0^{u+1} and $1^{u+1} := (1, 1, \dots, 1)$ as windows. Moreover if a sequence s contains 1^u as a window, then its pre-image(s) contain both $(0, 1, 0, 1, \dots)$ and $(1, 0, 1, 0, \dots)$ as length $u + 1$ windows.

Cycle joining. Let a and b be two binary, periodic sequences of periods m and n, respectively. Suppose that a and b together contain $m + n$ distinct windows of length u, so that both have span u and contain disjoint sets of windows. Suppose further that at some positions i and j we have $a_i, a_{i+1}, \dots, a_{i+u-2} = b_j, b_{j+1}, \dots, b_{j+u-2}$. By the assumed uniqueness of length u windows in a and b, we must then have $a_{i-1} \neq b_{j-1}$ and $a_{i+u-1} \neq b_{j+u-1}$. Hence the windows $(a_{i-1}, a_i, \dots, a_{i+u-2})$ and $(b_{j-1}, b_j, \dots, b_{j+u-2})$ form a conjugate pair. Then consider the sequence:

$$\begin{aligned} c \quad = \quad & [a_0, \dots, a_i, a_{i+1}, \dots, a_{i+u-2}, \\ & b_{j+u-1}, b_{j+u}, \dots, b_{n-1}, b_0, \dots, b_{j-1}, \\ & a_i, a_{i+1}, \dots, a_{m-1}]. \end{aligned}$$

By inspection, c has period $m + n$ and span u, and contains as its length u windows all the windows of a and b together.

When viewing sequences of span u as cycles in a de Bruijn graph of order u, this construction can be seen as joining the two cycles corresponding to a and b into a single cycle on $m + n$ vertices by using a so-called cross-join pair of vertices [17].

As an example, consider the pair of sequences $a = [0, 0, 0, 0, 0, 1, 0, 1]$ and $b = [1, 1, 1, 1, 1, 0, 1, 0]$ of period 8 and (joint) span 5 constructed as pre-images of $s = [0, 0, 0, 0, 1, 1, 1, 1]$ under the Lempel homomorphism. Notice how $a_4, a_5, a_6, a_7, a_8 = 0, 1, 0, 1, 0$ while $b_5, b_6, b_7, b_8, b_9 = 0, 1, 0, 1, 1$. We can then join a and b to make a sequence c of period 16 and span 5:

$$c = [0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1].$$

In fact, this cycle joining technique, in combination with the inverse Lempel homomorphism, yields one of the classical construction techniques for de Bruijn sequences: start with a de Bruijn sequence of order u, apply the inverse Lempel homomorphism to obtain two sequences of period 2^u that jointly contain 2^{u+1} distinct windows of length $u + 1$, and join them using a suitable cross-join pair to create a single sequence of period 2^{u+1} that now contains 2^{u+1} distinct windows, that is a de Bruijn sequence of order $u + 1$.

The construction. We have assembled the ingredients necessary for our construction of conjugate sequences. Our construction is iterative in nature.

Suppose we have a sequence s with the following two properties:

- s is an (n, u, t) conjugate sequence; and
- s contains both 0^u and 1^u as length u windows.

We compute the pre-image(s) of s under the Lempel homomorphism. Using properties established above, this results in either one cycle of period $2n$ or two cycles of period n (in case s has even weight). In either case, since s contains 0^u, the constructed cycle(s) contain(s) 0^{u+1} and 1^{u+1} as length $u+1$ windows. Moreover in the second case, because s contains 1^u, the constructed cycles contain the complementary vectors $(0, 1, 0, 1, \dots)$ and $(1, 0, 1, 0, \dots)$ as length $u + 1$ windows, one in each of the two cycles. Then the window following $(1, 0, 1, 0, \dots)$ in the relevant cycle must be $(0, 1, 0, 1, \dots)$ (ending in either two 0 bits or two 1 bits, depending on the parity of u). This means that the two cycles meet the necessary condition to be joined into a single cycle of period $2n$. Either way, we obtain a new sequence c of period $2n$ containing both 0^{u+1} and 1^{u+1}.

We claim that c is a $(2n, u + 1, 2t)$ conjugate sequence, that is, c contains twice as many length $u + 1$ conjugate pairs as the starting sequence s does conjugate pairs of length u. To see why this claim holds, consider any conjugate pair $(w_0, w_1, \dots, w_{u-1})$ and $(1 + w_0, w_1, \dots, w_{u-1})$ in s. These two windows result in four distinct length $u + 1$ windows in c, namely:

$$\begin{aligned} & 0, w_0, w_0 + w_1, \dots, w_0 + w_1 + \cdots + w_{u-1} \\ & 1, 1 + w_0, 1 + w_0 + w_1, \dots, 1 + w_0 + w_1 + \cdots + w_{u-1} \\ & 0, 1 + w_0, 1 + w_0 + w_1, \dots, 1 + w_0 + w_1 + \cdots + w_{u-1} \\ & 1, w_0, w_0 + w_1, \dots, w_0 + w_1 + \cdots + w_{u-1} \end{aligned}$$

By inspection, the first and fourth of these windows form a conjugate pair, as do the second and third. Moreover, because the Lempel homomorphism is a 2-to-1 map, we are assured that the t distinct conjugate pairs in s (consisting of $2t$ distinct length u vectors) lead to $2t$ distinct conjugate pairs in c (consisting of $4t$ distinct length $u + 1$ vectors).

In summary, from s satisfying the two properties above, we have constructed c also satisfying those properties, but with u replaced by $u + 1$ and s, an (n, u, t) conjugate sequence, being replaced by c, a $(2n, u + 1, 2t)$ conjugate sequence. Notice that the efficiency of c is $2 \cdot 2t/2n = 2t/n$, which is the same as that of s. In other words, the construction preserves efficiency.

All that remains is to identify suitable starting sequences for the construction. Consider the sequence $s = [0^u 1^u]$ (consisting of u 0's followed by u 1's in its least period). This s clearly satisfies the second condition, and by inspection, contains 2 conjugate pairs, namely $0^u, 10^{u-1}$ and $1^u, 01^{u-1}$. Thus it is a $(2u, u, 2)$ conjugate sequence, with efficiency $2 \cdot 2/2u = 2/u$.

Thus, for example, starting with $u = 9$ and a $(2 \cdot 9, 9, 2)$ conjugate sequence, we can apply the above construction $34 - 9 = 25$ times to obtain a $(2^{26} \cdot 9, 34, 2^{26})$ conjugate sequence s, i.e. a sequence of period $2^{26} \cdot 9$ containing 2^{26} conjugate pairs of length 34 vectors. This sequence is sufficient for our attack on Tarsnap in Section 3.5: the relevant value of $t_\delta(s)$ is (by direct computation) about $0.88 \cdot 2^{26}$ and this suffices for our attack on Tarsnap to have a reasonable success rate.

B Relations Between Security Notions

We now show that the two security notions FSWC-FtG and FSWC-RoR are asymptotically equivalent. We begin by proving that FSWC-RoR $\Rightarrow$ FSWC-FtG with a tight reduction.

Theorem B.1 (RoR implies FtG). *Let Chk be a FSW chunking algorithm. For all adversaries $\mathcal{A}$ in the FSWC-FtG game against Chk and making q queries to its oracle, there exists adversaries $\mathcal{B}_0$ and $\mathcal{B}_1$ such that*

$$\begin{aligned} & Adv^{FSWC\text{-}FtG}_{Chk}(\mathcal{A}) \\ & \leq Adv^{FSWC\text{-}RoR}_{Chk}(\mathcal{B}_0) + Adv^{FSWC\text{-}RoR}_{Chk}(\mathcal{B}_1) \end{aligned}$$

making q queries to its oracle and running approximately in the same time.

Proof. We proceed through a sequence of games, depicted in Fig. 6. Informally, the games are as follows:

- **Game** G_0: The case $b = 0$ for the Find-then-Guess game.
- **Game** G_1: The Find-then-Guess game, where the oracle instead replies with a randomly sampled decision for both the find and guess phases.
- **Game** G_2: The case $b = 1$ for the Find-then-Guess game.

The advantage of the adversary in the FSWC-FtG game is:

$$\begin{aligned}\mathrm{Adv}^{\text{FSWC-FtG}}_{\mathsf{Chk}}(\mathcal{A}) &= |\Pr[G_2] - \Pr[G_0]| \\ &\le |\Pr[G_2] - \Pr[G_1]| + |\Pr[G_1] - \Pr[G_0]|\end{aligned}$$

We claim that each term can be upper bounded by the advantage of an adversary ($\mathcal{B}_0$ and $\mathcal{B}_1$, respectively) in the FSWC-RoR game.

Bounding $\Pr[G_1] - \Pr[G_0]$**.** We construct adversary $\mathcal{B}_0$, simulating all queries for $\mathcal{A}$. To do so, it relays all queries to its own oracle and outputs the same bit b' as $\mathcal{A}$. Depending on the hidden bit b of the oracle, $\mathcal{A}$ plays in G_0 (case $b = 0$) or G_1 (case $b = 1$). The advantage of $\mathcal{B}_0$ is as follows:

$$\begin{aligned}&\mathrm{Adv}^{\text{FSWC-RoR}}_{\mathsf{Chk}}(\mathcal{B}_0) = \\ &= |\Pr[b' = 0|b = 1] - \Pr[b' = 0|b = 0]| = \\ &= |\Pr[G_1] - \Pr[G_0]|\end{aligned}$$

Bounding $\Pr[G_2] - \Pr[G_1]$**.** Similarly, we construct adversary $\mathcal{B}_1$, simulating all oracle queries for the find phase by relaying them to its own oracle. For the guess phase, it samples a random bit d and outputs it. In the end, it outputs the same bit b' as $\mathcal{A}$. The advantage of $\mathcal{B}_1$ is as follows:

$$\begin{aligned}&\mathrm{Adv}^{\text{FSWC-RoR}}_{\mathsf{Chk}}(\mathcal{B}_1) = \\ &= |\Pr[b' = 0|b = 1] - \Pr[b' = 0|b = 0]| = \\ &= |\Pr[G_1] - \Pr[G_2]| = |\Pr[G_2] - \Pr[G_1]|\end{aligned}$$

Plugging in the upper bounds concludes the proof.

□

The other direction is more involved and leads to a non-tight relationship between the two notions.

Theorem B.2 (FtG implies RoR). *Let Chk be a FSW chunking algorithm. For any adversary $\mathcal{A}$ be an adversary that has advantage ε in the FSWC-RoR game against Chk and making q queries to its oracle. Then, there exists an adversary $\mathcal{A}'$ that has advantage $\frac{\varepsilon}{q}$ in the FSWC-FtG game against Chk, making at most q queries to its oracle and running approximately in the same time.*

Proof sketch. The proof relies on a hybrid argument. We begin with the FSWC-RoR game in the case $b = 1$ (G_0), where each query is answered by sampling a random decision. In game G_i for $i \in 1, \ldots, q$, queries up to the i-th are answered by the real chunker and the remainder are answered by sampling a random decision. In the end, game G_q corresponds to the FSWC-RoR game in the case $b = 0$.

The difference in advantage of adversary $\mathcal{A}$ in transition from game G_i to G_{i+1} is upper bounded by the advantage of an adversary $\mathcal{B}_i$ in the FSWC-FtG game. The adversary $\mathcal{B}_i$ relays the first i queries from $\mathcal{A}$ to its find oracle. The $i + 1$-th query is answered using the challenge response in the guess phase. Depending on the hidden bit of the Find-then-Guess game, $\mathcal{A}$ is playing in either G_i ($b = 1$) or G_{i+1} ($b = 0$) which proves the advantage bound for each transition.

We thus prove the stated advantage bound

$$\mathrm{Adv}^{\text{FSWC-RoR}}_{\mathsf{Chk}}(\mathcal{A}) \le q \cdot \mathrm{Adv}^{\text{FSWC-FtG}}_{\mathsf{Chk}}(\mathcal{B}).$$

□

```
Game $G_0$:
$S \leftarrow \emptyset$
$K \leftarrow\$ \mathsf{Chk.KGen}()$
$(W^*, \mathsf{st}) \leftarrow\$ \mathcal{A}^{O_{\mathsf{real}}}(\mathsf{find})$
$d \leftarrow \mathsf{Chk.WindowEval}(K, W^*)$
$b' \leftarrow\$ \mathcal{A}(\mathsf{guess}, d, \mathsf{st})$
Return $b' = b \wedge W^* \notin S$

Game $G_1$:
$S \leftarrow \emptyset$
$(W^*, \mathsf{st}) \leftarrow\$ \mathcal{A}^{O_{\mathsf{random}}}(\mathsf{find})$
$d \leftarrow\$ \mathsf{Ber}(\lambda^{-1})$
$b' \leftarrow\$ \mathcal{A}(\mathsf{guess}, d, \mathsf{st})$
Return $b' = b \wedge W^* \notin S$

Game $G_2$:
$S \leftarrow \emptyset$
$K \leftarrow\$ \mathsf{Chk.KGen}()$
$(W^*, \mathsf{st}) \leftarrow\$ \mathcal{A}^{O_{\mathsf{real}}}(\mathsf{find})$
$d \leftarrow\$ \mathsf{Ber}(\lambda^{-1})$
$b' \leftarrow\$ \mathcal{A}(\mathsf{guess}, d, \mathsf{st})$
Return $b' = b \wedge W^* \notin S$

Oracle $O_{\mathsf{real}}(W)$:
$S \leftarrow S \cup \{W\}$
$d \leftarrow \mathsf{Chk.WindowEval}(K, W)$
Return $d$

Oracle $O_{\mathsf{random}}(W)$:
$S \leftarrow S \cup \{W\}$
$d \leftarrow\$ \mathsf{Ber}(\lambda^{-1})$
Return $d$
```

Figure 6: The sequence of games for the proof of Theorem B.1. Code in gray prevents trivial wins.

C Proof Details: Chk-PHTE is FSWC-RoR-secure

Fig. 7 shows the code for Chk-PHTE in each game game hop of the proof of Theorem 4.5.

D UHF Security of Restic's Polynomial Hashing

The following lemma shows that Restic's polynomial hash is a 2^{-44}-universal hash function.

Lemma D.1. *Let $\mathcal{H}_P$ be the family of polynomial hash functions used in Restic which consists of functions reducing binary polynomials*

Game G_0

```
Chk.KGen():
    $K_{\text{poly}} \leftarrow\$ \mathbb{F}$
    $K_{\text{E}} \leftarrow\$ \{0,1\}^k$
    Return $(K_{\text{poly}}, K_{\text{E}})$

Chk.WindowEval($K, W = (B_0, \ldots, B_{w-1})$):
    $(K_{\text{poly}}, K_{\text{E}}) \leftarrow K$
    $u \leftarrow \sum_{i=0}^{w-1} \text{btof}(B_i) \cdot K_{\text{poly}}^{w-1-i}$
    $c \leftarrow \text{E}(K_{\text{E}}, \text{ftob}(u))$
    If $c \mod \lambda = 0$: Return 1
    Else: Return 0
```

Game G_1

```
Chk.KGen():
    $K_{\text{poly}} \leftarrow\$ \mathbb{F}$
    $f \leftarrow\$ \text{Funcs}[\{0,1\}^n, \{0,1\}^n]$
    Return $(K_{\text{poly}}, f)$

Chk.WindowEval($K, (B_0, \ldots, B_{w-1})$):
    $(K_{\text{poly}}, f) \leftarrow K$
    $u \leftarrow \sum_{i=0}^{w-1} \text{btof}(B_i) \cdot K_{\text{poly}}^{w-1-i}$
    $c \leftarrow f(\text{ftob}(u))$
    If $c \mod \lambda = 0$: Return 1
    Else: Return 0
```

Game G_2'

```
Chk.KGen():
    $K_{\text{poly}} \leftarrow\$ \mathbb{F}$
    Return $K_{\text{poly}}$

Chk.WindowEval($K, (B_0, \ldots, B_{w-1})$):
    $K_{\text{poly}} \leftarrow K$
    $u \leftarrow \sum_{i=0}^{w-1} \text{btof}(B_i) \cdot K_{\text{poly}}^{w-1-i}$
    $c \leftarrow\$ \{0,1\}^\ell$
    If $c \mod \lambda = 0$: Return 1
    Else: Return 0
```

Game G_2

```
Chk.KGen():
    $K_{\text{poly}} \leftarrow\$ \mathbb{F}$
    Return $K_{\text{poly}}$

Chk.WindowEval($K, (B_0, \ldots, B_{w-1})$):
    $K_{\text{poly}} \leftarrow K$
    $u \leftarrow \sum_{i=0}^{w-1} \text{btof}(B_i) \cdot K_{\text{poly}}^{w-1-i}$
    $d \leftarrow\$ \text{Ber}(\lambda^{-1})$
    Return $d$
```

Figure 7: The sequence of games for the security proof of the FSW chunking algorithm. The highlighted parts are the changes made in each game wrt. the previous one. All the code of the challenger and oracle is unchanged across all games.

of degree at most 511 *modulo an irreducible polynomial P of degree* 53*. Then $\mathcal{H}_P$ is a 2^{-44}-universal hash function family.*

Proof. Let $\mathcal{K}$ be the key space containing all $\frac{2^{53}-2}{53}$ irreducible binary polynomials of degree 53. For any distinct binary polynomials A, B of degree at most 511, we have

$$\Pr_{P \leftarrow\$ \mathcal{K}}[\mathcal{H}_P(A) = \mathcal{H}_P(B)] = \Pr_{P \leftarrow\$ \mathcal{K}}[P \text{ divides } A - B]$$
$$\leq \left\lfloor \frac{\deg(A-B)}{\deg(P)} \right\rfloor \cdot \frac{1}{|\mathcal{K}|} \leq \left\lfloor \frac{511}{53} \right\rfloor \cdot \frac{53}{2^{53}-2} \leq 2^{-44}$$

Here, the first inequality follows from $A-B$ having at most $\left\lfloor \frac{\deg(A-B)}{\deg(P)} \right\rfloor$ distinct factors of degree $\deg(P)$. Plugging in the parameters of Restic and observing that $\deg(A-B)$ is bounded by the maximum of the degrees of A and B, the result follows. □