Corruption-Localizing Hashing

Giovanni Di Crescenzo[1], Shaoquan Jiang[2], and Reihaneh Safavi-Naini[3]

[1] Telcordia Technologies, Piscataway, NJ, USA
giovanni@research.telcordia.com
[2] School of Computer Science,
University of Electronic Science and Technology of China, China
jiangshq@calliope.uwaterloo.ca
[3] Department of Computer Science, University of Calgary, Canada
rei@ucalgary.ca

Abstract. Collision-intractable hashing is an important cryptographic primitive with numerous applications including efficient integrity checking for transmitted and stored data, and software. In several of these applications, it is important that in addition to detecting corruption of the data we also localize the corruptions. This motivates us to introduce and investigate the new notion of *corruption-localizing hashing*, defined as a natural extension of collision-intractable hashing. Our main contribution is in formally defining corruption-localizing hash schemes and designing two such schemes, one starting from any collision-intractable hash function, and the other starting from any collision-intractable keyed hash function. Both schemes have attractive efficiency properties in three important metrics: localization factor, tag length and localization running time, capturing the quality of localization, and performance in terms of storage and time complexity, respectively. The closest previous results, when modified to satisfy our formal definitions, only achieve similar properties in the case of a single corruption.

1 Introduction

A collision-intractable hash function is a fundamental cryptographic primitive, that maps arbitrarily long inputs to fixed-length outputs, with the required property that it is computationally infeasible to obtain two inputs that are mapped to the same output. One popular application of such functions is in the authentication and integrity protection of communicated data (i.e., as building blocks in the construction of digital signatures and message authentication codes). Other popular and more direct applications include practical scenarios that demand reliability of downloaded software files and/or protection of stored data against malicious viruses, as we now detail.

Software Reliability. Downloading software is a frequent need for computer users and checking the reliability of such software has become a task of crucial importance. One routinely used technique consists of accompanying software files with a short tag, computed as the output returned by a collision-intractable hash function on input the file itself. Later, the same function is used to detect whether the file has changed (assuming that no modification was done to the tag), and thus detect whether the software file was corrupted. An important example of the success of this technique is Tripwire

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 489–504, 2009.

[12], a widely available and recommended integrity checking program for the UNIX environment. However, with this approach even if one byte error (beyond the error-correction/detection capability of transmission protocols such as TCP) occurs in the transmission, the user has to download the whole file again. This is a waste of bandwidth and time. Alternatively, it would be desired to use a new kind of tag for which one can determine which blocks are corrupted and only retransmit those.

Virus Detection. Some of the most successful modern techniques attempting to solve the problem of virus detection fall into the general paradigm of integrity checking; see, e.g. [20,21] (in addition to other well-known paradigms, such as virus signature detection, which we do not deal with here). As before, tags computed using cryptographic hash functions detect any undesired changes in a given file or, more generally, file system (see, e.g., [5]) due to viruses. A taxonomy of virus strategies for changing files is given in [20]. With respect to that terminology, in the rest of the paper we consider the so-called 'rewriting infection strategies', where any single virus is allowed to rewrite up to a given number of consecutive blocks in a file (or, similarly, of consecutive files in a file system). In the context of virus defense, in the so-called 'virus diagnostics' [20] phase, it would be desirable to focus this phase on the localized area in the file rather than the entire file (we stress that this phase is usually both very resource-expensive and failure-prone, especially as the paradigm of integrity checking is typically used when not much information is available about the attacking virus).

In both above scenarios, in addition to detecting that after the data was detected to be corrupted, some potentially expensive procedure is required to deal with the corruption. For instance, in the case of software file download, the download procedure needs to be repeated from scratch; and in the case of stored data integrity, the impact of the corruption needs to be carefully analyzed so to potentially recover the data, sometimes triggering an expensive, human-driven, virus diagnostics procedure. Thus, in these scenarios, in addition to detecting that the data was corrupted, it would be of interest to obtain some information about the location of such corruptions (i.e., a relatively small area that includes all corrupted data blocks). For our two scenarios, such information would immediately imply savings in communication complexity (as only part of the download procedure is repeated), and reduce human resource costs (as the virus diagnostic phase will just focus on the infected data). This motivates us to formally define and investigate a new notion for cryptographic hashing, called *corruption-localizing hashing*, that naturally extends cryptographic hashing to achieve such goals.

Our contribution. Extending a concept put forward in [8], we formally define and investigate corruption-localizing hashing schemes (consisting of a hashing algorithm and a localization algorithm), defined as a natural generalization of collision-intractable hashing functions. With our formal definition of corruption-localizing hashing we define three important metrics: localization factor, tag length and localization running time, to capture the effectiveness of the localization, and efficiency of the system in terms of storage and time complexity, respectively. Localization factor is the ratio of the size of the area that is output by the localization algorithm to the size of corrupted area, where the former is required to contain the latter. We observe that simple techniques imply corruption-localizing hashing schemes with linear localization factor, or with small localization factor but with either a large localization running time or a large

Scheme	Localization factor	Storage complexity	Original Hash Function	Remark	Constraint
Trivial_1	$O(n/v)$	O(1)			
Trivial_2	1	$n\sigma$	cr		
[8]	$O(1)$	$O(\sigma \log n)$	cr		$\lvert S\rvert < n/4$
HS	$O(n^c)$	$O(\sigma \log n)$	cr	for some $c < 1$	$\lvert S\rvert < n/4$
HS	$O(n^d)$	$O(\sigma \log^2 n)$	cr	for any $0 < d < 1$	$\lvert S\rvert < n/2(v+1)$
KHS	$O(v^3)$	$O(\sigma v^2 \lambda \log_v n)$	cr-keyed		$\lvert S\rvert < n/2(v+1)$

Fig. 1. Asymptotical performance of 2 trivial schemes detailed at the end of Section 2, of a previous result from [8] for a single corruption, of 2 instantiations of our first scheme HS, and of our second scheme KHS for v corruptions. The term 'cr' (resp., 'cr-keyed') is an abbreviation for 'collision-resistant' (resp., 'collision-resistant, keyed'). Also, n denotes the file length, λ a security parameter that can be set $= O(\log^{1+\epsilon} n)$, for some $\epsilon > 0$, σ the output length of the (atomic) collision-resistant (keyed) hash function, and $|S|$ denotes the size of the largest corruption returned by the adversary. The value v for HS in the table is assumed to be constant; the general case can be found in Theorem 1.

tag length. We then target the construction of hashing schemes that achieve sub-linear localization without significantly increasing tag length or running time. Our main results are two schemes with provable corruption-localization whose properties are detailed in Figure 1, where HS is presented for constant v and the general case is stated in Theorem 1. Note that our schemes significantly improve the localization of $v \geq 1$ corruptions, at the cost of only slightly increasing storage complexity and running time of a conventional collision-resistant hash function. For instance, when v is constant, our first scheme, based on any collision-intractable hash function, achieves *sub-linear* localization factor and *logarithmic* tag length. Moreover, our second scheme, based on any collision-intractable keyed hash function, has *constant* localization factor and *poly-logarithmic* tag length. Using our schemes, in the software downloading scenario above, one can first obtain the (maybe corrupted) file and its tag (authentic), then use the latter to localize the corrupted parts and finally request retransmission of the localized parts only. Here, the tag used by our schemes is short and thus its authenticity can be guaranteed with small redundancy by standard error-correcting techniques (or, in certain applications, using a low-capacity channel).

Previous work. The concept of localization is clearly not new, and can be considered as intermediate between the two concepts of detection and correction, which are well studied, for instance, in the coding theory and watermarking literatures. In general terms, localization is expected to provide better benefits and demand more resources than detection and provide worse benefits and demand less resources than correction, where, depending on applications and on benefit/resource tradeoffs, one concept may be preferable over the other two. Moreover, our paper differs crucially from research in both fields of coding theory and watermarking in that it specifically targets constructions based on cryptographic hash functions, and their applications. This difference translates in different construction techniques, security properties (as the collision-intractability and corruption-localization of cryptographic hash functions and the correction property

in coding theory are substantially different properties), and adversary models (typically, in coding theory one considers arbitrary changes which can be modeled as unbounded adversaries, while we only consider polynomial-time bounded adversaries). By definition, the collision-intractability property of cryptographic hash functions already provides a computational version of the detection property but falls short of providing non-trivial localization, which we target here.

We also note that several aspects in the mentioned example applications have also been studied from various angles. A first example is from [10] which studied the security of software download in mobile e-commerce. This paper and follow-up ones mainly focus on software-based security and risks involved in this procedure. A second example is from [4], which introduced a theoretical model for checking the correctness of memories. This paper and follow-up ones do not target constructions based on cryptographic hash functions, and the constructions exhibit similar differences and tradeoffs with our paper, as for the previously mentioned detection and correction concepts. A third example, apparently the closest line of research to the one from our paper, is from (non-adaptive) combinatorial group testing [9]. In this area, the goal is to devise combinatorial tests to efficiently find which objects out of a pool are defective. Note that testing whether a collision-resistant hash function maps two messages to the same tag could be considered a combinatorial test, and thus the technique from this area might be applicable to our problem. However, one main crucial difference here is that combinatorial group testing refers to same-size objects, while in this paper we recognize that practical corruptions may have very different sizes. Thus, even the best approaches from this area (exactly finding w defective objects out of a pool of n using $O(w^2 \log n)$ storage) do not scale well as a single corruption, as defined in our model, may imply $w = \omega(\sqrt{n})$ and thus super-linear storage, which is worse than the Trivial_2 construction in Figure 1. Other important differences include the following: this area implements the above correction concept, while our paper focuses on localization; moreover, our paper works out the exact security analysis of the hashing functions, while the combinatorial group testing area only focuses on combinatorial aspects.

Overall, the closest previous result to ours appeared in [8], which informally introduced a notion equivalent to corruption-localization hashing, for the case of a single corruption. One of their schemes satisfies our formal definition in the case of a single corruption, and is a special case of our first scheme. We stress that the extension to multiple corruptions is quite non-trivial both with respect to the formal definition (see Section 2) and with respect to the constructions and proofs (see Sections 3, 4).

2 Definitions and Model

We assume familiarity with families of (conventional and keyed) cryptographic hash functions and pseudo-random function families. Here, we present our new notions and formal definitions of corruption-localizing hash schemes.

Corruption-Localizing Hashing: Notations. We assume that the input x to a (keyed) hash function consists of a number of atomic *blocks* (e.g., a bit or a byte or a line); let $x[i]$ denote the i-th *block* of x; i is called *index* of $x[i]$; let $x[i, j]$ denote the sequence of *consecutive* blocks $x[i], x[i + 1], \ldots, x[j - 1], x[j]$, also called a *segment*. In general,

for $S = \{i_1, \cdots, i_t\} \subseteq \{0, \cdots, n-1\}$, define $x[S] = x[i_1]x[i_2]\cdots x[i_t]$. A sequence of segments $(x[i_1, j_1], \cdots, x[i_k, j_k])$ is also called a *segment list*. We define a left cyclic shift operator $\mathcal{L}$ for x by $\mathcal{L}(x) = x[1]x[2]\cdots x[n-1]x[0]$. Iteratively applying $\mathcal{L}$, we have $\mathcal{L}^i(x) = x[i]\cdots x[n-1]x[0]\cdots x[i-1]$ for any $i \geq 0$. For a set S, $|S|$ denotes the number of elements in it. For any (possibly probabilistic) algorithm A, an *oracle algorithm* is denoted as A^O, where O is an (oracle) function, and the notation $a \leftarrow A(x, y, z, \ldots)$ denotes the random process that runs algorithm A on input $x, y, z, \ldots$, and denotes the resulting output as a.

Corruption-Localizing Hashing: Formal Model. Our generalization of collision-intractable hash functions into hash schemes and keyed hash schemes is in having, in addition to the hashing algorithm, a second algorithm, called the *localizer*, which, given a corrupted input x' and the hash value (also called *tag*) for the original input x, returns some indices of input blocks. If strings x and x' are a message (or file) x and its corrupted version x', then the localizer's output are indices of all corrupted segments of the input file. This improves over conventional hashing which typically reveals that a corruption happened, but does not offer any further information about which input blocks it happened at. To measure the quality of the localization, we introduce a parameter, called *localization factor*, that determines the accuracy of localizer and is defined (roughly speaking) as the ratio of the size of the localizer output to the size of the actual corrupted blocks. (Note that since the file size is measured in terms of the number of blocks, we only need to consider the number of blocks.)

In this model, we only consider a *replacement attack*: given input x, adversary replaces up to v *segments* of x by new ones while each replaced segment preserves its original length (i.e., containing the same number of blocks). Our model allows each segment to contain *arbitrary and unknown* number of blocks. This adversary model well captures the applications described in the introduction. For instance, when a software file is downloaded over the Internet some packets (regarding the payload in one packet as one block) get noisy or even lost. In rewriting infections by viruses, some lines in an executable might be replaced by malicious commands. Our objective for localization is to output a small set T of indices that contains the corrupted blocks. Then, in case of software download, we only need to request retransmission of blocks in T. We will be mainly interested in partially corrupted files, for which a localization solution for the applications mentioned in the introduction is of much more interest. Thus, when designing our schemes, we assume a (sufficiently large) upper bound β on the size of the maximum corruption segment.

Before describing the model, we define the difference between x and its corrupted version x'. We generally consider the case where x' is corrupted from x by v segments (instead of blocks). Given as input two n-block strings x and x', we define a function Diff_v as follow. For $S \subset \{0, \cdots, n-1\}$, let $\overline{S} = \{0, \cdots, n-1\} \backslash S$.

$\mathsf{Diff}_v[x, x'] = \min \sum_{i=1}^{v} |S_i|$, where each $S_i \subset \{0, \cdots, n-1\}$ is a segment, and the minimum is over all possible $\{S_i\}_{i=1}^{v}$ such that $x[\overline{\cup_{i=1}^{v} S_i}] = x'[\overline{\cup_{i=1}^{v} S_i}]$.

Here $S_i \subseteq \{0, \cdots, n-1\}$ and thus it might be empty, and $x[\overline{\cup_{i=1}^{v} S_i}]$ and $x'[\overline{\cup_{i=1}^{v} S_i}]$ are strings x and x', respectively, with segments $S_i, i = 1, \cdots, v$ removed.

Intuitively, $\mathsf{Diff}_v[x, x']$ is the minimal total size of v segments that an adversary can modify in order to change x to x'. For example, let $v = 2, n = 11, x = 00000000000$, and $x' = 10100000100$, and assume x' is the corrupted version of string x. We note the minimal size of *two* segments in x that one can modify in order to change x to x' is 4: $S_1 = \{0, 1, 2\}$, $S_2 = \{8\}$ and $\mathsf{Diff}_2[x, x'] = 4$. Generally, we say $S_i \subset \{0, \cdots, n-1\}, i = 1, \cdots, v$ achieve $\mathsf{Diff}_v[x, x']$, if $\sum_{i=1}^{v} |S_i| = \mathsf{Diff}_v[x, x']$ and $x[\overline{\cup_{i=1}^{v} S_i}] = x'[\overline{\cup_{i=1}^{v} S_i}]$. Note $\mathsf{Diff}_v[x, x']$ can always be computed in time $O(n^{v-1})$ by searching for the rightmost element of segment S_i and verifying if $x[\overline{\cup_{i=1}^{v} S_i}] = x'[\overline{\cup_{i=1}^{v} S_i}]$. On the other hand, $\mathsf{Diff}_v[x, x']$ is mainly required in the definition of the security experiment below but need not be calculated in our corruption-localization algorithms. So we do not require an efficient algorithm for computing $\mathsf{Diff}_v[x, x']$.

We then define a *hash scheme* as a pair $\mathrm{HS} = (\mathrm{CLH}, \mathrm{LOC})$, where CLH is an algorithm that, on input an n-block string x (and, implicitly, a security parameter) returns a string tag, and LOC is an algorithm that, on input an n-block string x' and a string tag, returns a set of indices $T \subseteq \{0, \cdots, n-1\}$. Similarly, we define a *keyed hash scheme* as a pair $(\mathrm{CLKH}, \mathrm{KLOC})$, where CLKH is an algorithm that, on input an n-block string x (and, implicitly, security parameter λ), a λ-bit string k, returns a string tag, and KLOC is an algorithm that, on input an n-block string x', a λ-bit string k, and a string tag, returns a set of indices $T \subseteq \{0, \ldots, n-1\}$.

We now formally define the corruption-localization properties of hash schemes and keyed hash schemes, using three additional parameters: v, the number of corrupted segments, β the upper bound on the number of corrupted blocks in the largest corruption segment, and α the lower bound on the ratio of the number of blocks T that is the output of the localizing algorithm to $\mathsf{Diff}_v[x, x']$.

Definition 1. Let $\mathrm{HS} = (\mathrm{CLH}, \mathrm{LOC})$ be a hash scheme and $\mathrm{KHS} = (\mathrm{CLKH}, \mathrm{KLOC})$ be a keyed hash scheme.

For any $t, \epsilon, \alpha, \beta, v \geq 0$, the hash scheme HS is said $(t, \epsilon, \alpha, \beta, v)$-*corruption-localizing* if for any algorithm A running in time t and returning corruption segments of size $\leq \beta$, the probability that experiment $\mathsf{HExp}^{\mathrm{HS},A,hash}(\alpha, v)$ (defined below) returns 1 is at most ϵ.

For any $t, q, \epsilon, \alpha, \beta, v \geq 0$, the keyed hash scheme KHS is said $(t, q, \epsilon, \alpha, \beta, v)$-*corruption-localizing* if for any oracle algorithm A running in time t, making at most q oracle queries, and returning corruption segments of size $\leq \beta$, the probability that experiment $\mathsf{KExp}^{\mathrm{KHS},A,keyh}(\alpha, v)$ (defined below) returns 1 is at most ϵ.

$\mathsf{HExp}^{\mathrm{HS},A,hash}(\alpha, v)$
1. $(x, x') \leftarrow A(\alpha, v)$
2. $tag \leftarrow \mathrm{CLH}(x)$
3. $T \leftarrow \mathrm{LOC}(v, x', tag)$
4. if $x[\overline{T}] \neq x'[\overline{T}]$ then **return:** 1
5. if $|T| > \alpha \cdot \mathsf{Diff}_v[x, x']$ then **return:** 1 else **return:** 0.

$\mathsf{KExp}^{\mathrm{KHS},A,keyh}(\alpha, v)$
1. $k \leftarrow \{0,1\}^{\lambda}$
2. $(x, x') \leftarrow A^{\mathrm{CLKH}_k(\cdot)}(\alpha, v)$
3. $tag \leftarrow \mathrm{CLKH}_k(x)$
4. $T \leftarrow \mathrm{KLOC}(k, v, x', tag)$
5. if $x[\overline{T}] \neq x'[\overline{T}]$ then **return:** 1
6. if $|T| > \alpha \cdot \mathsf{Diff}_v[x, x']$ then **return:** 1 else **return:** 0.

In both above experiments, the adversary is successful if it either prevents *effective localization* (i.e., one of the modified blocks is not included in T), or forces the scheme to exceed the expected localization factor (i.e., $|T| > \alpha \cdot \mathsf{Diff}_v[x, x']$).

Corruption-Localizing Hashing: metrics of interest. We use the following three main metrics of interest to evaluate and compare corruption-localizing hash schemes and keyed hash schemes.

First, the parameter α in the above definition is called *localization factor*. Note that a collision-resistant hash function implies a trivial corruption-localizing hash scheme with localization factor at least $\alpha = n/v$. This is by simply defining the algorithm Loc to return all blocks $\{0, \ldots, n-1\}$, where n is the length of the input to the hash function CLH. (This is scheme Trivial_1 in Figure 1.) Clearly, we target better schemes with localization factor $o(n/v)$ or even constant.

A second metric of interest is the output length of the hash function, also called *tag length*. Note that a corruption-localizing hash scheme with localization factor 1 and efficient localizer running time can be simply constructed as follows: the tag is obtained by calculating the hash of each block in the input message individually (if a block is not small such as a long line); the localizer returns the indices where the hashes differ. (This is scheme Trivial_2 in Figure 1.) Clearly, such a scheme is not interesting since the tag length is linear in n. Instead, we target schemes where the tag length is logarithmic or poly-logarithmic in n.

A third metric of interest is the localizer's *running time* as a function of n, where n is the length of the input to the function CLH (or CLKH). Our schemes only slightly decrease the efficiency of the atomic collision-resistant hash function used.

3 A Corruption-Localizing Hashing Scheme

In this section we design a corruption-localizing hash scheme based on any collision-resistant hash function. Our scheme can be instantiated so that it localizes up to v corruptions in an n-block file, while satisfying a non-trivial localization factor, very efficient storage complexity and only slightly super-linear runtime complexity. For instance, when v is constant (as a function of n), it has localization factor $O(n^c)$, for some $c < 1$, and $O(\log n)$ storage complexity, or localization factor $O(n^d)$, for any $0 < d < 1$, and $O(\log^2 n)$ storage complexity. (See Theorem 1 and related remarks for formal and detailed statements.) In the rest of the section, we start with an informal description and a concrete example for the scheme, and then conclude with the formal description and a sketch of proof of its properties.

AN INFORMAL DESCRIPTION. At a very high level, our hash algorithm goes as follows. A collection of block segments from the n-block file x are joined to create several segment lists, and the collision resistant hash function h_λ is applied to compute a hash tag for each segment list. The localizer, on input a file x' with up to v corruptions, computes a hash tag on input the same segment lists from file x', and eliminates all segment lists for which the obtained tag matches the tag returned by the hash algorithm. The remaining blocks are returned as the area localizing the v corruptions. The hard part in the above high level description is choosing block segments and segment lists in such a way to achieve desired values for the localization, storage and running time metrics.

Here, our approach can be considered a non-trivial extension of the scheme from [8] that provides non-trivial localization for a single corruption (i.e., $v = 1$). We start by briefly recalling the mentioned scheme, and, in particular, by highlighting some of the properties that will be useful to describe our scheme.

A single-corruption scheme. The scheme in [8] follows the above paradigm in the case $v = 1$ and localizes any single corrupted segment S (of up to $n/4$ blocks) with localization factor 2, using $O(\log n)$ storage and running in $O(n \log n)$ time. There, $n = 2^w$ for some positive integer w. Now, assume that S satisfies $2^{w-i_0-1} < |S| \leq 2^{w-i_0}$, and let $i = i_0 - 1$. The n-block file x is split into 2^i consecutive segments, each containing 2^{w-i} blocks. Then, the 2^i segments are grouped into 2 segment lists such that the ℓ-th segment is assigned to segment list $\ell \mathrm{mod} 2$. Thus, each of the 2 segment lists contains 2^{i-1} segments. So far, the idea is that if, for some i, one of the 2 segment lists contains the entire corruption, then the localization is restricted to the segment list containing the entire corruption. However, it may happen that the corruption lies in one intersection of the two segment lists, in which case the above 2 tests do not help. To take care of this situation, the same process is repeated for a cyclic shift by 2^{w-i-1} blocks of file x. Then, the corruption will intersect at most 3 out of 4 segment lists, and the remaining one can be considered "corruption-free". This already provides some localization, but further hash tags are needed to achieve an interesting localization factor. In particular, because the corruption size and thus the value i_0 are not known, the above process is repeated for $i = 1, \ldots, w - 1$ from the hash algorithm, and until such i_0 is found from the localizer.

Our multiple-corruption scheme. The natural approach of using the same scheme for $v \geq 2$ fails because an attacker can carefully place 2 corruptions so that one intersects both segment lists generated from file x and the other one intersects both segment lists generated from the cyclic shift of file x. This is simple to realize for any specific i, and can be realized so that the intersections happen for all $i = 1, \ldots, w$, by enforcing the intersections when $i = 1$. We avoid this problem by increasing the number of segment lists. Specifically, we write $n = z^w$ for some positive integers z, w satisfying $z > v$ (where parameter z has to be carefully chosen), and repeat the same process by using z segment lists rather than 2, for all $i = 1, \ldots, w$.

However, not any value for z would work: because each corrupted segment can intersect up to 3 segment lists (2 generated from file x and 1 from the cyclic shift of x, or viceversa), it turns out that, for instance, choosing $z \leq 3v/2$ would still allow for one (less obvious) placement of the v corruptions by the attacker so that no segment lists can be considered "corruption-free". Moreover, choosing any $z > 3v/2$ may result in a less desirable localization factor. We deal with these problems by increasing the number of cyclic shifts, denoted as y, of the original file x: more precisely, we repeat the process for each file obtained by shifting x by n/y blocks.

We can show that these two modifications suffice to maintain efficiency in storage and time complexity, to achieve effective localization (or else the collision-resistance of the original hash function is contradicted) and to achieve a non-trivial localization factor. To prove the latter claim, we show that: (1) over all cyclic shifts, v corruptions intersect with at most $\leq v(y + 1)$ segment lists in total; (2) hence, there exists one cyclic shift of x, for which these v corruptions intersect at most $\lfloor v(y + 1)/y \rfloor$ segment

lists; (3) for each $i = 1, \ldots, w-1$, the set T_i of blocks that have not been declared "corruption-free" satisfies $|T_i| \leq n \cdot \nu^i$ for some $\nu < 1$ and $0 \leq i \leq i_0$, where i_0 is such that $|S_a| \leq z^{w-i_0}/y$ for all corrupted segments S_a and $|S_a| > z^{w-i_0-1}/y$ for some corrupted segment S_a. Here, we note that fact (3) is proved using facts (1) and (2) and implies that the final output T_{w-1} from the localizer is a "good enough" localization of the v corruptions.

A CONCRETE EXAMPLE. We discuss (and depict in Figure 2) a concrete example of our scheme, starting with a file $x = x[0] \cdots x[63]$, containing $n = z^w = 4^3 = 64$ blocks, with the parameter settings $z = 4, w = 3$. Our scheme consists of tag algorithm CLH_1 (see left side of Figure 2) and localization algorithm LOC_1 (see right side of Figure 2).

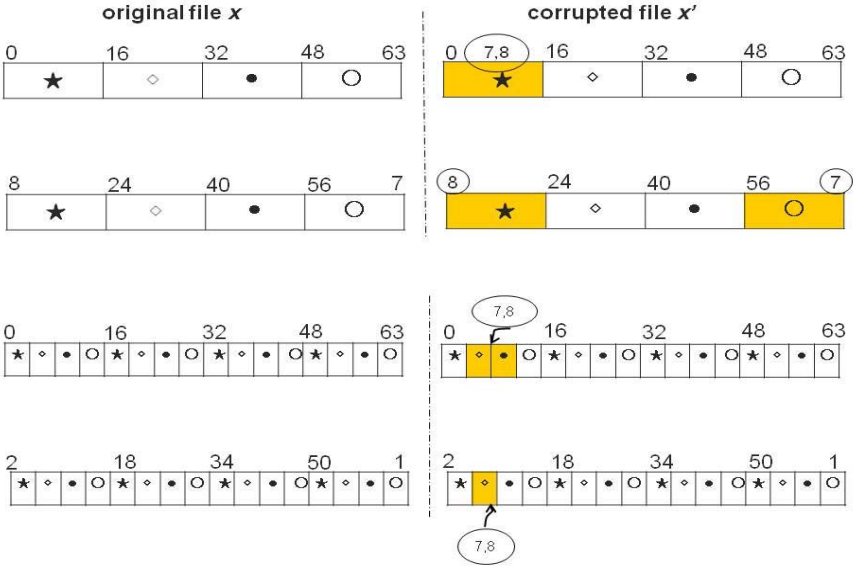

Fig. 2. The HS scheme for $n = z^w = 64, z = 4, w = 3, y = 2$

Hash Algorithm. The algorithm CLH_1 consists of $w - 1 = 2$ stages and can be considered as a sequence of computations of hash tags based on the following equations, for different values of ℓ, i:

$$tag_{\ell,i,0} = h_\lambda(\star), tag_{\ell,i,1} = h_\lambda(\diamond), tag_{\ell,i,2} = h_\lambda(\bullet), tag_{\ell,i,3} = h_\lambda(\circ), \qquad (1)$$

where $\star, \diamond, \bullet, \circ$ are 4 classes of segments, that are differently obtained from x at each application of these equations.

Stage one. x is split into $z^1 = 4$ segments of equal size $n/z^1 = z^{w-1} = 16$ (row 1 in the figure). That is, $(0, \cdots, 63) = \star || \diamond || \bullet || \circ$, and the equations in (1) are applied for $\ell = 0, i = 1$. Now, set parameter y as $= 2$. Next, left cyclic shift x by $1/y$ segment size (see row 2). That is, shift $z^{w-1}/y = 8$ blocks. The result is $\mathcal{L}^{z^{w-1}/y}(x) = \mathcal{L}^8(x) = (8, 9, \cdots, 63, 0, \cdots, 7)$. Again split $\mathcal{L}^8(x)$ into $z^1 = 4$ blocks $\star || \diamond || \bullet || \circ$ and apply the equations in (1) for $\ell = 1, i = 1$. In this example, $y = 2$. If $y \geq 3$, we need to

further consider $\mathcal{L}^{\ell 2^{w-i}/y}(x)$ for $\ell \leq y-1$ similarly. In this scenario, cases $\ell = 0, 1$ are similarly as above.

Stage two. Here, x is split into $z^2 = 16$ segments of each size $n/z^2 = 64/16 = 4$ (see row 3). Then assign all segments into 4 classes $\star, \diamond, \bullet$ and $\circ$. $\star$ contains segments $0, 4, 8, ..., 48$; $\diamond$ contains segments $1, 5, 9, ..., 49$; $\bullet$ contains segments $2, 6, 10, ..., 50$; $\circ$ contains segments $3, 7, 11, ..., 51$. Then we apply the equations in (1) for $\ell = 0, i = 2$. Next, as in Stage one, we cyclicly shift x by $z^{w-2}/y = 4/2 = 2$ blocks (see row 4). That is, we compute $\mathcal{L}^{z^{w-2}/y}(x) = \mathcal{L}^2(x) = (2, 3, \cdots, 63, 0, 1)$. We similarly classify $\mathcal{L}^2(x)$ into classes $\star, \diamond, \bullet$ and $\circ$ and apply the equations in (1) for $\ell = 1, i = 2$.

Localization Algorithm. Suppose x is corrupted in a file x' by changing blocks 7, 8. We compute a set $T \subseteq \{0, \cdots, 63\}$ that contains $7, 8$ but $|T|$ is not large. There are two stages. Initially, set $T_0 = \{0, \cdots, 63\}$.

Stage one. Similarly as for x, split x' into $\star || \diamond || \bullet || \circ$ and compute ${tag'}_{0,1,j}, j = 0, \cdots, 3$. Then since ${tag'}_{0,1,j} = tag_{0,1,j}, j = 1, 2, 3$, it follows that $\diamond, \bullet, \circ$ are all uncorrupted (see row 1); otherwise, h_λ is not collision-resistant. Then we can update $T_0 = T_0 \backslash \{16, \cdots, 63\} = \{0, \cdots, 15\}$. By verifying ${tag'}_{0,1,0} \neq tag_{0,1,0}$, we know $\star$ contains a corruption. Then we consider a shift $\mathcal{L}^8(x')$ of x', i.e., $(8, \cdots, 63, 0, \cdots, 7)$ (see row 2). Let $T_1 = T_0$. Compute ${tag'}_{1,1,j}, j = 0, \cdots, 3$. Since ${tag'}_{1,1,j} = tag_{1,1,j}$ for $j = 1, 2$, then $T_1 = T_1 \backslash \{24, \cdots, 55\} = \{0, \cdots, 15\}$ remains unchanged.

Stage two. Consider row 3 in Figure 2. Split x' into $z^2 = 16$ segments. Set $T_2 = T_1$. Compute ${tag'}_{0,2,j}, j = 0, \cdots, 3$. Since ${tag'}_{0,2,j} = tag_{0,2,j}$, we can update $T_2 = T_2 - \{0, \cdots, 3\} - \{16, \cdots, 19\} - \{32, \cdots, 25\} - \{48, \cdots, 51\} = \{4, \cdots, 15\}$. Similarly, from ${tag'}_{0,2,3} = tag_{0,2,3}$, we can update T_2 to $T_2 = \{4, \cdots, 11\}$. Next, consider a shift $\mathcal{L}^2(x')$ of x' (see row 4 in Figure 2). Compute ${tag'}_{1,2,j}, j = 0, \cdots, 3$. Since ${tag'}_{1,2,j} = tag_{1,2,j}$ for $j = 0, 2, 3$, we can update T_2 by removing indices not in $\diamond$. The result is $T_2 = \{4, \cdots, 11\} - \{2, \cdots, 5\} - \{10, \cdots, 17\} = \{6, 7, 8, 9\}$. So the localization factor here is $\alpha = 2$.

FORMAL DESCRIPTION AND PROOFS. Our formal presentation (in Fig. 3) is a generalization of the above concrete example, where the classes $\star || \diamond || \bullet || \circ$ are replaced by symbol $S_{\ell,i,j}$. The scheme's properties are formally described in the following theorem.

Theorem 1. *Let z, y, v, λ, w be positive integers such that $y \mid z$, $v < yz(y+1)^{-1}$, $n = z^w$, and let $\beta = n/2y$. Assume $\mathcal{H} = \{H_\lambda\}_{\lambda \in \mathcal{N}}$ is a (t, ϵ)-collision-resistant family of hash functions from $\{0,1\}^{p_1(\lambda)} \to \{0,1\}^\sigma$. Then there exists a $(t', \epsilon', \beta, v)$-corruption-localizing hash scheme* HS*, where $\epsilon' = \epsilon$ and $t' = t + O(t_n(H) \cdot yz \log_z n)$, where $t_n(H)$ is the running time of functions from H_λ on inputs of length n. Moreover,* HS *has localization factor $\alpha = \lfloor v(y+1)/y \rfloor^{-1} zy \cdot n^{\log_z \lfloor v(y+1)/y \rfloor}$, tag length $\tau = 3 \log n + \sigma zy \log_z n + |desc(H)|$, and runtime complexity $\rho = O(t_n(H) \cdot zy \log_z n)$, where $|desc(H)|$ is an upper bound on the description size of functions from H_λ.*

Remarks and parameter instantiations. The condition $n = z^w$ is for simplicity only and can be removed by a standard padding. When v is constant, we can always choose constants z, y such that $v < yz(y+1)^{-1}$. It follows that in this setting it always holds that $\alpha = O(n^c)$, for *some constant* $c < 1$. So our scheme does provide a non-trivial localization (in terms of file size n): α sublinear, τ logarithmic and ρ almost linear. Moreover, by setting $y = v + 1$ and $z = \log n$, we have $\alpha = v^{-1}(v+1) \log n \times$

The algorithm CLH_1**:** On input x, $|x| = n$, and parameters (z, y), do the following:
- Randomly choose h_λ from H_λ
- For $i = 1, \ldots, w-1$, and $\ell = 0, \ldots, y-1$,
  set $s = \ell \cdot z^{w-i}/y$ and compute $x_{\ell,i} = \mathcal{L}^s(x)$
  split $x_{\ell,i}$ into segments $B_{\ell,i,0} \| \cdots \| B_{\ell,i,z^i-1}$ of equal length
  for $j = 0, \ldots, z-1$,
    compute segment list $S_{\ell,i,j} = (B_{\ell,i,j} \| B_{\ell,i,j+z} \| \cdots \| B_{\ell,i,j+z^i-z})$
    compute $tag_{\ell,i,j} = h_\lambda(S_{\ell,i,j})$
- Output: $tag = \{tag_{\ell,i,j} \mid \ell \in \{0, \ldots, y-1\}, i \in \{1, \ldots, w-1\}, j \in \{0, \ldots, z-1\}\} \cup \{n, z, y, desc(h_\lambda)\}$.

The algorithm LOC_1**:** On input x', $|x'| = n$, tag, and parameters (z, y), do the following:
- Let $tag = \{tag_{\ell,i,j} \mid \ell \in \{0, \ldots, y-1\}, i \in \{1, \ldots, w-1\}, j \in \{0, \ldots, z-1\}\} \cup \{n, z, y, desc(h_\lambda)\}$.
- Set $T_0 = \{0, \ldots, n-1\}$.
- For $i = 1, \ldots, w-1$,
  set $T_i = T_{i-1}$
  for $\ell = 0, \ldots, y-1$, and $j = 0, \ldots, z-1$,
    compute $B'_{\ell,i,j}, S'_{\ell,i,j}$ from x' as done for $B_{\ell,i,j}, S_{\ell,i,j}$ from x in CLH_1
    let $I_{\ell,i,j}$ be the set of indices for $B'_{\ell,i,j}$
      i.e., $I_{\ell,i,j} = \{\ell \cdot z^{w-i} y^{-1} + j \cdot z^{w-i}, \ldots, \ell \cdot z^{w-i} y^{-1} + j \cdot z^{w-i} + z^{w-i} - 1\}$
    if $h_\lambda(S'_{\ell,i,j}) = tag_{\ell,i,j}$ then update $T_i = T_i \setminus \cup_{t=0}^{z^{i-1}-1} I_{\ell,i,j+zt}$.
- Output: T_{w-1}.

Fig. 3. The Corruption-Localizing Hash Scheme HS

$n^{\log\log^{-1} n \times \log v}$. By simple calculation, we have that for *any* $0 < c < 1$, $\alpha = O(n^c)$, $\tau = O(\log^2 n)$ and $\rho = O(n \log^2 n)$. That is, for *any* $0 < c < 1$, HS localizes any v corruptions up to a sub-linear factor $O(n^c)$ with only poly-logarithmic tag length and slightly super-linear running time, where v can be up to $c' \log n$, for $c' < c$. Finally, by setting $y = z = 2$ and $v = 1$, we obtain $\alpha = 4$, $\tau = (3 + 4\sigma) \log n$ and $\rho = 4n\sigma \log n$; i.e., HS localizes a single corruption up to a small constant factor with logarithmic tag length and slightly super-linear running time. Note that one scheme in [8] considered this special case and has a result essentially matching ours.

Proof idea of Theorem 1. As ρ and τ can be checked by calculation, and effective localization can be seen to directly follow from the collision-intractability of the original hash function, here we only focus on justifying the localization factor α. Obviously, T_i is related to the size of each corrupted segment S_a. Let i_0 be such that each $|S_a| \leq nz^{-i_0}/y$ but some $|S_a| \geq nz^{-i_0-1}/y$. If we are able to show that $|T_i| \leq n \cdot \nu^i$ for some $\nu < 1$ and all $0 \leq i \leq i_0$, then we have that $|T_{w-1}| \leq |T_{i_0}| \leq n \cdot \nu^{i_0}$ and thus

$$|T_{w-1}| \leq nz^{-i_0-1}/y \cdot zy(z\nu)^{i_0} \leq zy \sum_a |S_a| \cdot z^{i_0 \log_z(z\nu)} \leq zy \sum_a |S_a| \cdot n^{\log_z(z\nu)},$$

which is a sub-linear factor in n since $z\nu < z$. So we need to show an upper bound of $|T_i|$ can decrease with i by some factor $\nu < 1$ for $i \leq i_0$. We demonstrate the technical idea for this using the example in Fig. 2. Here, the corrupted segment is $S_1 = \{7, 8\}$.

Then, it holds that $i_0 = 2$. Consider row 1 and 2 in Fig. 2. Since S_1 has a size 2 and segment size is z^{w-1}, the event that S_1 is intersecting with two neighboring segments can occur in at most one of x and $\mathcal{L}^8(x)$. In our example, in $\mathcal{L}^8(x)$, S_1 intersects with two segments $\{\star, \circ\}$. So in x and $\mathcal{L}^8(x)$, there are at most 3 segments in total intersecting with S_1 (in general, these are at most $v(y+1)$). So one of x and $\mathcal{L}^8(x)$ contains at most $\lfloor 3/2 \rfloor = 1$ corrupted segments (in general, these are $\lfloor v(y+1)/y \rfloor$). In our example, x contains 1 corrupted segment. So $|T_1| = n/z = 16$ (in general, $|T_1| = \lfloor v(y+1)/y \rfloor \cdot n/z$). Now we only consider Stage two (row 3 and 4 in Fig. 2). Again, since S_1 has size 2 and segment size is $z^{w-2} = 4$, the event that S_1 is intersecting with two neighboring segments can occur in at most one of x and $\mathcal{L}^2(x)$. The remaining part in this stage is to follow the idea in stage one. We obtain that $|T_2| = 4$ (in general, $T_2 = \lfloor v(y+1)/y \rfloor \cdot |T_1|/z = (\lfloor v(y+1)/y \rfloor / z)^2 \cdot n$, where $\lfloor v(y+1)/y \rfloor / z < 1$ by assumption). The formal proof carefully generalizes the idea in this description.

4 A Corruption-Localizing Keyed Hashing Scheme

In this section we propose a corruption-localizing *keyed* hash scheme starting from any collision-resistant *keyed* hash function. Our scheme improves the previous (not keyed) scheme on the localization factor for an arbitrary number of corruptions, and on the range of the number of corruptions for which it provides non-trivial localization. In particular, for a constant number of corruptions, it provides essentially optimal (up to a constant factor) localization, at the expense of small storage complexity and only a small increase in running time. (See Theorem 2 and related remarks for the formal statement.) In the rest of the section, we start with an informal description, then give a concrete example, the formal description and a sketch of proof of its properties.

AN INFORMAL DESCRIPTION. By using keyed hash functions in our previous scheme, we do obtain a corruption-localizing keyed hash scheme. The following construction, however, makes a more intelligent use of the randomness in the key resulting in significant improvements both on the localization factor and on the range for the number of corruptions, with only a slightly worse performance in storage and time complexity.

At a very high level, our keyed hash algorithm goes as the hash algorithm of scheme HS, with the following differences. The new algorithm uses the secret key shared with the localizer (and unknown to the attacker) as an input to a pseudo-random function that generates pseudo-random values. These latter values are used as colours associated with each block segment of each cyclic shift of file x (including the file x itself). Then, segment lists are created so that each segment list contains all block segments of a given colour. In other words, the generation of segment lists from the block segments is done (pseudo-)randomly and in a way that it can be done by both the hash algorithm and the localizer, but not by the attacker (as the key is unknown to the attacker and the hash tags are further encrypted using a different portion of the key).

The reason for this pseudo-random generation of segment lists is that the deterministic generation done in scheme HS allowed the attacker to place the corruptions in a way to maximize the number of intersections with segment lists. This resulted in a localization factor still polynomial in n (even though the polynomial could be made as small as desired at moderate losses in terms of storage and time complexity). Instead, the

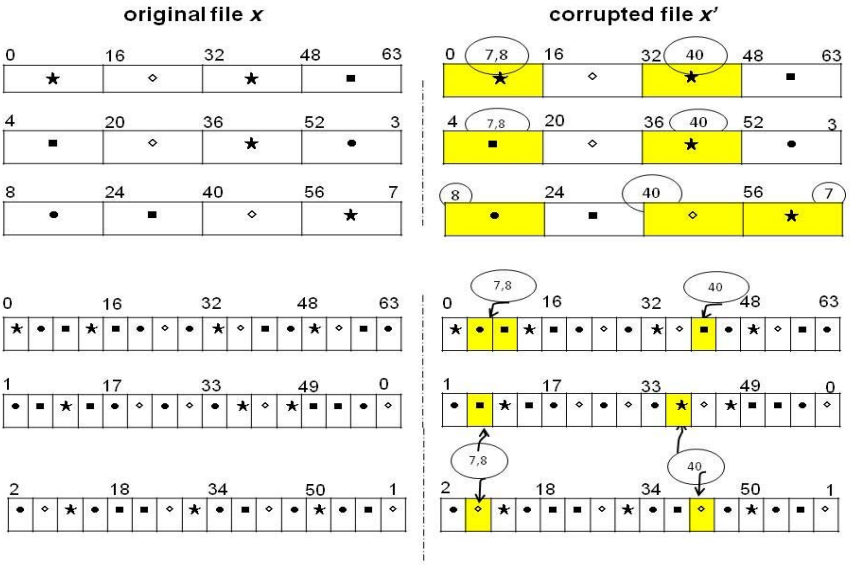

Fig. 4. The CLKH scheme for $n = z^w = 64, z = 4, w = 3$. (Note: $\mathcal{L}^{12}(x)$ and $\mathcal{L}^3(x)$ are not shown in the figure.)

pseudo-random generation of the segment lists makes it much harder for the attacker to place corruptions so to intersect a large number of segment lists, and is crucial to achieve constant localization factor (except with negligible probability).

A CONCRETE EXAMPLE. In Fig. 4 we illustrate an example for scheme KHS analogous to the one in the previous section for scheme HS. We again use file $x = x[0] \cdots x[63]$, but we now consider $v = 3$ and $n = (v+1)^w = 4^3 = 64$ and $w = 3$. As before, segments are somehow assigned to classes $\star, \bullet, \diamond, \circ$, and analogues of the equations in (1) are used to compute hash tags, the differences being here that the hash functions used are keyed functions, the assignment of the segments to the classes is probabilistic, and the tags are further encrypted using a key available to the localizer. Specifically, scheme HS can be regarded as assigning the classes to the segments *periodically* while the current scheme assigns a class to each segment *randomly* (see left part of Fig. 4). Now, let x' be the corrupted version of x, where blocks $7, 8, 40$ are changed. The localization algorithm (see right part of Fig. 4) returns $T_2 = \{6, 7, 8, 9, 40\}$, thus resulting in a localization factor $\alpha = 5/3 = 1.67$.

FORMAL DESCRIPTION. The formal presentation of our keyed hash scheme can be found in Fig. 5. The properties of this scheme are shown in the following theorem.

Theorem 2. *Let λ, v, w be positive integers such that $v \geq 2$ and $n = (v+1)^w$, and define $\beta = n/2(v+1)$, and δ a function negligible in λ. Assume $\mathcal{H} = \{H_\lambda\}_{\lambda \in \mathcal{N}}$ is a (t_h, ϵ_h)-collision-resistant family of keyed hash functions from $\{0,1\}^\lambda \times \{0,1\}^{p_1(\lambda)} \to \{0,1\}^\sigma$ and $\mathcal{F} = \{f_k\}_{|k| \in \mathcal{N}}$ is a (t_f, ϵ_f)-pseudo-random family of functions. Then the scheme in Fig. 5 is a $(t', \epsilon', \beta, v)$-corruption-localizing keyed hash scheme* KHS*, where $\epsilon' \leq \epsilon_h + \epsilon_f + \delta$ and $t' \leq t_f + t_h + O(t_n(H) \cdot (v+1)^2 \log_{v+1} n)$, where $t_n(H)$ is*

The algorithm CLKH**:** On input $k, x, |x| = n$, do the following:
- Randomly choose h_λ from H_λ
- Write k as $k = k_1|k_2|k_3$, randomly choose nonces μ_1, μ_2, and let psr_1, psr_2 be sufficiently long number of pseudo-random bits obtained as $psr_i = f_{k_i}(\mu_i)$, for $i = 1, 2$;
- For $i = 1, \ldots, w-1$, and $\ell = 0, \ldots, v$,
  compute $x_{\ell,i}$ and $B_{\ell,i,0}, \ldots, B_{\ell,i,(v+1)^i-1}$ as done in CLH_1 (in the case of $x = y = v+1$)
  for $z = 1, \ldots, \lambda$,
    for each $j = 0, \ldots, (v+1)^i - 1$
      randomly choose *colour* $c_{\ell,i,j,z} \in \{C_0, \ldots, C_v\}$ and assign it to $B_{\ell,i,j}$, (using fresh pseudorandom bits from psr_1)
    for $c \in \{C_0, \ldots, C_v\}$,
      let $S_{\ell,i,c,z}$ be the set of segments $B_{\ell,i,j}$ ($j \in \{0, \cdots, (v+1)^i - 1\}$) with assigned color $c_{\ell,i,j,z} = c$
      compute $tag_{\ell,i,c,z} = h_\lambda(k_3; S_{\ell,i,c,z}) \oplus psr_2$
- Output: $tag = \{n, s, \mu_1, \mu_2, desc(h_\lambda), tag_{\ell,i,c,z} \mid \ell \in \{0, \ldots, v\}, i \in \{1, \ldots, w-1\}, c \in \{C_0, \ldots, C_v\}, z \in \{1, \ldots, \lambda\}\}$.

The algorithm KLOC**:** On input k, v, x', tag, where $k = k_1|k_2|k_3$, and $tag = \{n, s, \mu_1, \mu_2, desc(h_\lambda), tag_{\ell,i,c} \mid \ell \in \{0, \ldots, v\}, i \in \{1, \ldots, w-1\}, c \in \{C_0, \ldots, C_v\}, z \in \{1, \ldots, \lambda\}\}$, do the following:
- Set $T_0 = \{0, \ldots, n-1\}$ and compute psr_1, psr_2 as in CLKH;
- For $i = 1, \ldots, w-1$,
  set $T_i = T_{i-1}$
  for $\ell = 0, \ldots, v$, $c = C_0, \ldots, C_v$, and $z = 1, \ldots, \lambda$,
    compute $S'_{\ell,i,c,z}$ from x' as done for $S_{\ell,i,c,z}$ from x in CLKH above
    let $I_{\ell,i,c,z}$ be the set of indices from all segments in $S'_{\ell,i,c,z}$
    if $psr_2 \oplus h_\lambda(k_3; S'_{\ell,i,c,z}) = tag_{\ell,i,c,z}$ then set $T_i = T_i \setminus I_{\ell,i,c,z}$
- Output: T_{w-1}.

Fig. 5. The Corruption-Localizing Keyed Hash Scheme KHS

the running time of any keyed hash function from H_λ on inputs of n blocks. Moreover, KHS *has localization factor $\alpha = (v+1)^2 v$, storage complexity $\tau = O(\log n + \sigma(v+1)^2 \lambda \log_{v+1} n + |desc(H)|)$, and runtime complexity $\rho = O(t_f + t_n(H) \cdot v^2 \lambda \log_{v+1} n)$, where $|desc(H)|$ is an upper bound on the description size of any hash function from H_λ and $\lambda = O(\log^{1+\epsilon} n)$ for any $\epsilon > 0$.*

Remarks and proof idea. We note that if $v = O(1)$, scheme KHS can localize v corruptions with a constant localization factor and polylogarithmic (in n) storage complexity. We also note that an active adversary could observe which blocks are being re-sent and then infer the coloring and build more efficient attacks. However, the honest parties share a key and can thus encrypt their communication and pad it to the upper bound on the localization factor so to not even release how many blocks are being resent.

Now we outline the proof idea for Theorem 2. As ρ and τ can be checked by calculation, we only need to consider localization factor α. Obviously, T_i is related to the size of each corrupted segment S_a. Let i_0 be such that each $|S_a| \leq nz^{-i_0-1}$ but some

$|S_a| \geq n(v+1)^{-i_0-2}$. If we are able to show that $|T_i| \leq vn \cdot (v+1)^{-i}$ for $0 \leq i \leq i_0$, then $|T_{w-1}| \leq |T_{i_0}| \leq vn(v+1)^{-i_0} \leq n(v+1)^{-i_0-2} \cdot v(v+1)^2 \leq (v+1)^2 v \sum_a |S_a|$, constant localization factor $(v+1)^2 v$. So we focus on proving $|T_i| \leq vn \cdot (v+1)^{-i}$ for $i \leq i_0$. Instead of a rigorous proof, we demonstrate the technical idea using the example in Figure 4, where the corrupted segments are $S_1 = \{7, 8\}$ and $S_2 = \{40\}$. Consider Row one and Row two in Figure 4. As in the proof idea for the HS scheme, one of $\mathcal{L}^{4i}(x)$ for $i = 0, 1, 2, 3$ has at most $\lfloor v(y+1)/y \rfloor = \lfloor 2(v+2)/(v+1) \rfloor = 2$ corrupted segments. In our example, x has 2 corrupted segments SB_1, SB_3 (see Row one). If there is coloring z such that SB_1, SB_3 are assigned to the same color and SB_2, SB_0 are assigned to other color(s), then SB_2 and SB_4 are uncorrupted and can be removed from T_1. This occurs with probability $1/4 \cdot (3/4)^2$. Since we have λ coloring experiments, this event won't occur only with negligible probability. In our Row one, SB_1, SB_3 are assigned to color $\star$; while SB_2 is assigned to color $\diamond$ and SB_4 is assigned to color $\circ$. Therefore, $|T_1| \leq 2(v+1)^{w-1} \leq vn \cdot (v+1)^{-1}$. So it holds for $i = 1$. In iteration two, x is divided into $(v+1)^2 = 16$ segments. Again similar to the proof idea in HS scheme, there is i such that $\mathcal{L}^i(x)$ has at most $\lfloor v(y+1)/y \rfloor = \lfloor 2(v+2)/(v+1) \rfloor = 2$ corrupted segments. In our example, $\mathcal{L}^1(x)$ in row 5 has this property. T_1 intersects with $\mathcal{L}^1(x)$ at most $2(v+1)+2 = 10$ segments. In our example, it is 10 segments exactly. By our assumption, among these 10 segments, two are corrupted and the remaining are uncorrupted. In our example, SB_2 and SB_{10} are corrupted. If in some experiment we can color these two with one color and the remaining 8 to other colors, then $T_2 \subseteq SB_2 \cup SB_{10}$ and thus $|T_2| \leq 2(v+1)^{w-2} \leq vn \cdot (v+1)^{-2}$. The conclusion holds again. Such a coloring occurs with probability $1/4 \cdot (3/4)^8$. Since there are λ colorings, this desired coloring does not occur with exponentially small probability only. The formal proof of the theorem carefully generalizes the idea in this description.

Acknowledgements. Jiang's work was mainly done at U. of Calgary supported by Informatics Circle of Research Excellence and is now supported by National 863 High Tech Plan (No. 2006AA01Z428), NSFC (No. 60673075) and UESTC Young Faculty Plans.

References

1. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
2. Blaze, M.: A Cryptographic File System for UNIX. In: Proc. of 1993 ACM Conference on Computer and Communications and Security (1993)
3. Blum, M., Kannan, S.: Designing Programs That Check Their Work. In: Proc. of the 1989 ACM Symposium on Theory on Computing (1989)
4. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the Correctness of Memories. In: Proc. of the 1995 IEEE Symposium on Foundations on Computer Science (1995)
5. Cattaneo, G., Catuogno, L., Del Sorbo, A., Persiano, G.: The Design and Implementation of a Cryptographic File System for UNIX. In: Proc. of 2001 USENIX Annual Technical Conference (2001)
6. Damgård, I.B.: Collision free hash functions and public key signature schemes. In: Price, W.L., Chaum, D. (eds.) EUROCRYPT 1987. LNCS, vol. 304, pp. 203–216. Springer, Heidelberg (1988)

7. Di Crescenzo, G., Ghosh, A., Talpade, R.: Towards a Theory of Intrusion Detection. In: de Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 267–286. Springer, Heidelberg (2005)
8. Di Crescenzo, G., Vakil, F.: Cryptographic hashing for Virus Localization. In: Proc. of the 2006 ACM CCS Workshop on Rapid Malcode (2006)
9. Du, D., Hwang, F.: Combinatorial Group Testing and its Applications. World Scientific Publishing Company, Singapore (2000)
10. Ghosh, A., Swaminatha, T.: Software security and privacy risks in mobile e-commerce. Communications of the ACM 44(2), 51–57 (2001)
11. Goldreich, O., Goldwasser, S., Micali, S.: How to Construct Random Functions. Journal of the ACM 33(4) (1986)
12. Kim, G., Spafford, E.: The design and implementation of tripwire: a file system integrity checker. In: Proc. of 1994 ACM Conference on Computer and Communications Security (1994)
13. Merkle, R.: A Certified Digital Signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435. Springer, Heidelberg (1990)
14. NIST. Secure hash standard. Federal Information Processing Standard, FIPS-180-1 (April 1995)
15. NIST. Secure Hash Signature Standard (SHS) (FIPS PUB 180-2). United States of America, Federal Information Processing Standard (FIPS) 180-2, August 1 (2002)
16. NIST, Cryptographic Hash Algorithm Competition, `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`
17. Oprea, A., Reiter, M., Yang, K.: Space-Efficient Block Storage Integrity. In: Proc. of 2005 Network and Distributed System Security Symposium (2005)
18. Rivest, R.: The MD5 Message-Digest Algorithm. Request for Comments (RFC 1320). Internet Activities Board, Internet Privacy Task Force (April 1992)
19. Russell, A.: Necessary and Sufficient Conditions for Collision-Free Hashing. Journal of Cryptology 8(2) (1995)
20. Skoudis, E.: MALWARE: Fighting Malicious Code. Prentice-Hall, Englewood Cliffs (2004)
21. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Reading (2005)
22. Stalling, W., Brown, L.: Computer Security: Theory and Practice. Prentice-Hall, Englewood Cliffs (2007)
23. Sivathanu, G., Wright, C., Zadok, E.: Ensuring Data Integrity in Storage: Techniques and Applications. In: Proc. of the 2005 ACM International Workshop on Storage Security and Survivability (2005)
24. 1st NIST Cryptographic Hash Functions Workshop, `http://www.csrc.nist.gov/pki/HashWorkshop/2005/program.htm`