An Omission-Tolerant Cryptographic Checksum

Francisco Corella and Karen Lewison

Pomcor

Abstract. A cryptographic checksum is a small data item that is computed from a data structure and can be used to prevent undetected intentional modification by making it difficult for an adversary to construct a different data structure that has the same checksum. We broaden the concept of a checksum to include the concept of a data item intended to prevent certain modifications while tolerating others. An omission-tolerant checksum is computed from a data structure that represents a set and does not change when elements are omitted from the set, while making it difficult for an adversary to modify the set in any other way without invalidating the checksum.

We use the root label of a typed hash tree to implement an omission-tolerant checksum. A typed hash tree is a variation on a Merkle tree, where each node has a type and a label. The root label of a typed hash tree does not change when a subtree is pruned from the tree. The same is true for a Merkle tree, but in a Merkle tree this a "bug" to be mitigated, while in a typed has tree it is a "feature" that makes it possible to use the label of the root node as an omission-tolerant checksum for a set of key-value pairs. To do so, we encode each key-value pair as the type and label of a leaf node of an incomplete typed hash tree without internal-node labels, then serialize the tree to obtain a bit-string encoding of the set. To compute the checksum on the encoding we deserialize the bit-string, compute the missing internal nodes, and output the label of the root node as the checksum.

We use Boneh and Shoup's system parameterization and attack game methodology to prove that, given a set of key-value pairs, an efficient adversary has a negligible probability of producing a set of key-value pairs, other than a subset of the given one, that has the same checksum.

1 Introduction

A *checksum* is a small data item that is computed from a data structure and can be used to verify its integrity, i.e. to verify that the data structure has not been accidentally or intentionally modified. We are primarily concerned with *cryptographic checksums* that are used to prevent undetected intentional modification by making it difficult for an adversary to construct a different data structure that has the same checksum. An example of a cryptographic checksum is a cryptographic hash computed on a bit-string encoding of a data structure. A cryptographic checksum can be verified by comparing it to an original checksum supplied by the originator of the data structure, or by verifying a signature on the checksum by the originator.

In this paper we broaden the concept of a checksum to include the concept of a data item intended to prevent certain modifications while tolerating others. An *omission-tolerant checksum* is computed from a data structure that represents a set and does not change when elements are omitted from the set, i.e. when the set is replaced with a subset, while making it difficult for an adversary to modify the set in any other way without invalidating the checksum.

More broadly, given a binary relation defined among the elements of a set, a *relaxed-integrity checksum* with respect to the relation is a data item that is computed from a data structure representing an original element of the set and does not change when the original element is replaced with a different element that is related to the original one, while it is difficult for an adversary to find an unrelated element that has the same checksum. Omission-tolerant integrity is the special case of relaxed integrity where the binary relation is the subset relation.

We use a *typed hash tree* to represent a set of key-value pairs with omission-tolerant integrity protection provided by the root label of the tree, i.e. by the label of its root node. In a typed hash tree each node has a type and a label, and the label of an internal node is a cryptographic hash of an encoding of the types and labels of its children called the prelabel of the node. A distinguished type is assigned to all the internal nodes and possibly to some of the leaf nodes. The key and the value of a key-value pair are encoded as bit strings and represented by the type and the label of a leaf node that has an undistinguished type. The same key may be associated with multiple values, and thus a typed hash tree can be used to represent a set of unstructured elements as a special case of a set of key-value pairs where there is only one key, associated with all the unstructured elements.

The key-value pairs in a subtree can be omitted without modifying the root label of the tree by "pruning" the subtree, causing its root node to become a leaf node whose type is the distinguished type; we refer to such a node as a dangling node; this can be used to implement selective disclosure by pruning different subtrees when presenting the tree to different verifiers. When the tree is used for selective disclosure, a dangling node with a random label may be used to make it infeasible to tell whether a subtree rooted at that node has been pruned or the node was originally dangling.

If a typed hash tree Y is derived from a typed hash tree X by pruning one or more subtrees, then X and Y have the same root label. Conversely, we formally prove that if X and Y have the same root label, then either Y is isomorphic to a pruned derivative of X, or the label of an internal node of Y is equal to the label of a dangling node of X, or an internal node of Y has the same label as an internal node of X but a different prelabel.

From this formal result it follows that if a typed hash tree is used to represent a set of key-value pairs and the hash function used to construct the tree is collision and preimage resistant, then the root label of the tree can serve as an omission-tolerant checksum of that set. This is easy to see informally, but a formal proof faces the well known difficulty of formalizing the concepts of collision

and preimage resistance for a keyless hash function. We solve this difficulty using the novel concept of system parameterization proposed by Boneh and Shoup in [2], and the attack game methodology that they use.

A further difficulty arises from the fact that some of the algorithms involved in the attack games are *hybrid non-deterministic algorithms*, which make two kinds of non-deterministic choices: probabilistic choices, and arbitrary, non-probabilistic ones. The output of such an algorithm is a random variable that has a well-defined distribution over a probability space only if the probabilistic choices that define the space are made after the non-probabilistic ones and after any probabilistic ones not included in the definition of the space. We solve this difficulty in the asymptotic security setting by letting only some aspects of a typed hash tree be dependent on the security parameter.

An omission-tolerant checksum has many possible applications. A particularly important one is a method for implementing selective disclosure of data items bound to a public key by a public key certificate, either by including the omission-tolerant checksum in a traditional X.509 certificate and presenting the data with any omissions separately, or by defining a new certificate format that includes the data, with omissions as desired at presentation time, and the signature computed on the omission-tolerant checksum. Cryptographic credentials that feature selective disclosure of attributes, such as anonymous credentials [3] or U-Prove tokens [5], have been proposed before; but they are more difficult to use than public key certificates and have failed to be widely deployed. Public key certificates with selective disclosure would be much easier to deploy for a wide range of privacy-preserving applications, such as TLS client authentication. Details of such applications are left for future work.

1.1 Prior work and present contributions

Hash trees were first proposed by Merkle [11]. In a Merkle tree, each internal node is labeled by a hash of the concatenation of the labels of its children, and each leaf node is labeled by a hash of a data block. In both kinds of trees, a subtree can be pruned without modifying the root label of the tree. But in the case of a Merkle tree this is a "bug" to be mitigated if the tree is to provide integrity protection, while in the case of a typed hash tree it is a "feature" that allows the root label of the tree to be used as an omission-tolerant checksum. The root label of a Merkle tree is a checksum of a collection of data blocks whose hashes label the leaf nodes. The root label of a typed hash tree is a checksum of a collection of key-value pairs, the key and value of each pair being represented by the type and the label of a leaf node. Pruning a subtree causes data blocks in the case of a Merkle tree, or key-value pairs in the case of typed hash tree, to be removed from the checksummed collection without modifying the checksum. But in the case of a Merkle tree, it also causes an extraneous data block to be added to the collecion, viz. the concatenation of the labels of the children of the root of the subtree. In the case of a typed hash tree, nothing is added to the collection, because the type and label of a leaf node do not represent a key-value pair when the type of the leaf node is the distinguished type.

In Certificate Transparency [8], the subtree pruning "bug" of a Merkle hash tree is mitigated by prepending a byte to the data that is hashed into the label of a node, the value of the byte being 0x00 for leaf nodes and 0x01 for internal nodes. This byte is akin to a type in a typed hash tree, but the intended effect of the byte is to alter the value of the root label of the tree when a subtree is pruned. So this kind of type cannot be used for implementing omission-tolerant integrity protection.

The Secure Electronic Transactions (SET) protocol [10,13], uses a "dual signature" to disclose different information about an online credit card transaction to the merchant and to the acquiring bank. The dual signature uses a cascade of hashes that can be mapped to a very special case of a typed hash tree. Neither a broadly applicable concept of an omission-tolerant checksum nor a proof of security are provided.

We originally proposed the concept of omission-tolerant integrity protection in a white paper concerned with a specific application [9]. NIST has subsequently but independently proposed the equivalent concept of "integrity protection with erasure capability" [6]. In the method that NIST uses to implement the concept, data items, referred to as blocks, are arranged in a two-dimensional matrix, and integrity is verified by the hashes of the rows and columns of the matrix. Zeroing out one of the blocks alters the hashes of the row and column of the block, but the hashes of the other rows and columns can still be used to verify the integrity of the other blocks. A carefully designed algorithm assigns a sequence of blocks to matrix positions excluding the diagonal, in a manner that ensures that two consecutive blocks can be zeroized without losing integrity protection for other blocks. An advantage of this method over ours is that a data block can be altered in any way, not just deleted by zeroing it, without losing integrity protection for other blocks. A disadvantage is that, when a block is zeroized or otherwise altered, multiple hashes have to be individually checked to verify the integrity of the remaining blocks. Origin authentication by digital signature would thus requires all the row and column hashes to be individually signed. It is interesting that the collection of row and column hashes provides integrity protection but does not constitute an omission-tolerant checksum, since two of the hashes in the collection change when a block is zeroized or altered. This shows that the concepts of omission-tolerant checksum and omission-tolerant integrity protection are not equivalent.

US patent 6,802,002 [4] describes "structured certificates", which are public key certificates containing nested "certificate folders" than can be open or closed, where closing a folder means replacing it with a cryptographic hash of the folder. This is essentially equivalent to binding the public key to the root label of a typed hash tree, a folder being equivalent to a subtree, and closing a folder being equivalent to pruning the corresponding subtree. Thus the certificates described in the patent provide selective disclosure of certified information. However neither a broadly applicable concept of an omission-tolerant checksum nor a proof of security are provided.

The contributions made by this paper include:

1. The concepts of omission-tolerant checksum and omission-tolerant integrity protection, and the broader concept of relaxed integrity protection with respect to a binary relation.
2. The concept of a typed hash tree.
3. A fundamental theorem relating the structures of typed hash trees having the same root label (Theorem 1).
4. A method for using a typed hash tree to represent a set of key-value pairs (or, as a special case, a set of unstructured elements all associated as values with the same key), serializing the tree to obtain a bit-string encoding of the set, and using the root label of the tree as an omission-tolerant cryptographic checksum of the encoding.
5. A formal proof in an asymptotic security setting that given a set of key-value pairs, an efficient adversary has a negligible probability of producing a set of key-value pairs, other than a subset of the given one, that has the same root-label checksum (Theorem 3).

2 Preliminaries

2.1 Algorithms

Many definitions of the concept of algorithm have been proposed [14]. We shall not use a particular one or propose our own. We think of an algorithm in a practical way as an abstraction of the concrete concept of a computer program, and all algorithms described in the paper can be readily implemented in a modern programming language.

An algorithm may be passed any input or inputs, but may stop without output if it receives an unexpected input, or for other reasons. Stopping without output is an abstraction of the programming concept of throwing an exception. When we say that an algorithm *takes as input* data of some particular form we mean that it determines whether the input is of that form before performing any other computation, and stops without output if it is not of that form. An algorithm may also have parameters, which may be viewed as being hardcoded into the program that implements the algorithm or as additional inputs.

An algorithm may be deterministic or non-deterministic. Non-deterministic algorithms may make probabilistic choices or arbitrary, non-probabilistic choices. Examples of algorithms that make non-probabilistic choices, in a variety of contexts, include: ASN.1 encoding with Basic Encoding Rules (BER), which allow more than one way of encoding a data item into a byte string; the algorithm for generating the DSA domain parameters p and q specified in [12, Appendix A.1.1.2], which chooses an arbitrary domain parameter seed at step 5; and the non-deterministic polynomial time algorithms used in the definition of the complexity class NP. Examples of algorithms that make probabilistic choices, again in a variety of contexts, include: the Miller-Rabin primality testing algorithm; the OAEP padding algorithm for RSA encryption; randomized signature schemes such as DSA or ECDSA; and Boneh & Shoup's concept of a system parameterization algorithm [2, Definition 2.9] further discussed below in Section 5.1.

Some algorithms make both kinds of choices. For example, the the algorithm for generating the DSA domain parameters p and q specified in [12, Appendix A.1.1.2] chooses an arbitrary domain parameter seed at step 5, then uses the Rabin-Miller primality test, which makes probabilistic choices. We refer to such algorithms as *hybrid algorithms*, and to an algorithm that only makes probabilistic choices as a *probabilistic algorithm*.

The output of a probabilistic algorithm is a random variable with a probability distribution that only depends on its inputs. We will have the need to view the output of a hybrid algorithm as a random variable whose probability distribution depends on the non-probabilistic choices and some of the probabilistic choices, and make statements about that random variable that are universally quantified over the inputs and the other choices. However such probability distribution is only well defined if the choices that define the probability space are made after the other choices, and we will be careful to ensure that this is indeed the case.

When describing specific algorithms, for the sake of brevity and as is done elsewhere, e.g., in [1], we shall sometimes blur the distinction between algorithmic variables, which take different values at different stages of an execution of an algorithm, and particular values of those variables.

2.2 Games

We use attack games to define security properties as in [2].

A game has a challenger and an adversary, which are algorithms, and parameters that are available to the challenger and the adversary. The challenger may be a hybrid algorithm, but the adversary may not make any non-probabilistic choices, and is therefore either deterministic or probabilistic.

Executing the game means invoking the challenger algorithm, with the adversary algorithm provided as an input to the challenger algorithm. As its last step, the challenger usually invokes the adversary algorithm, providing it with inputs, and obtaining any outputs produced by the adversary. In exceptional cases, the challenger may stop without invoking the adversary. The game protocol specifies the steps taken by the challenger, the inputs that the challenger provides to the adversary, whether the adversary is deemed to have won or lost if it is not invoked by the challenger, and winning conditions on any outputs produced by the adversary that determine if the adversary wins the game, in the usual case where it is invoked by the challenger. After running the adversary, the challenger produces a boolean output indicating whether the adversary has won the game. It also relays any outputs produced by the adversary as part of its own outputs, and may produce other outputs that it has computed itself.

2.3 Notations and conventions

We write $\{0,1\}^*$ for the set of all bit strings, $\{0,1\}^n$ for the set of bit strings of length n, $\{0,1\}^{<n}$ for the set of bit strings less than n, and $\mathbf{Z}_{\geq 1}$ for the set of the positive integers.

We write $f : S \mapsto S'$ to state that f is a function with domain S and codomain S'.

As in [1], we use the term "fact" to refer to a theorem whose proof is left to the reader.

2.4 Hash functions

It is difficult to provide formal definitions for the security properties of hash functions, especially for collision resistance [7, end of §5.1.1], [2, preamble of §8.1]. In an asymptotic security setting, the difficulty comes from the absence of randomness in the inputs and algorithms of hash functions, and is usually avoided by defining security properties for *keyed hash functions* that take a random key as an additional input, rather than for *keyless* hash functions such as those of the SHA-2 family. As further discussed below in Section 5.1, we overcome this difficulty using the system parameterization methodology of [2], which introduces randomness without adding a key argument to the hash function.

For our purposes we need hash functions with domain $\{0,1\}^{<a}$ and codomain $\{0,1\}^b$ with a much greater than b and we only consider such hash functions. The hash functions of the SHA-2 family satisfy this requirement, with b equal to 224, 256, 384 and 512 for SHA-224, SHA-256, SHA-384 and SHA-512 respectively, and a being equal to 2^{64} for SHA-224 and SHA-256, and 2^{128} for SHA-384 and SHA-512, as determined by the number of bits used to specify the length of the input in the padding. This requirement motivates the following definition of a *simple hashing scheme*. A definition of a *hashing scheme with system parameterization* will be provided in Section 5.

Definition 1 (Simple hashing scheme). *A simple hashing scheme is a triple (a, b, h), where a and b and positive integers such that $a > b$ and h is a function with domain $\{0,1\}^{<a}$ and codomain $\{0,1\}^b$.*

A hash function should be hard to invert and have security properties such as collision and preimage resistance. The following broad definition does not capture such requirements, but formal definitions of collision and preimage resistance are provided below in Section 5 using attack games.

Definition 2 (Hash function). *A function h is a hash function if there exist positive integers a and b such that (a, b, h) is a simple hashing scheme.*

2.5 Simple encoding schemes

We will need the following simple notion of an encoding scheme. A more complex definition of an encoding scheme with system parameterization will be provided in Section 5.

Definition 3 (Simple encoding scheme). *A* simple encoding scheme *for the elements of a set S is a triple $\mathcal{E} = (S, E, D)$, where*

1. *S is the set of data items to be encoded.*
2. *E is an algorithm, possibly non-deterministic, called the* encoding algorithm *of the scheme, that takes as input an element x of S and outputs a bit string y called an* encoding of *x.*
3. *D is a deterministic algorithm, called the* decoding algorithm *of the scheme, that takes as input a bit string y and is such that:*
   (a) *If y is an output of E on input x, then D outputs x on input y.*
   (b) *If y is not an output of E, then D stops without output.*

3 Typed hash trees

A typed hash tree is an ordered tree augmented by assignments of types and labels to its nodes, where the label of each internal node is a hash of a prelabel consisting of the types and labels of its children. The formal definition is in four stages, which define the concepts of an ordered tree, an ordered tree augmented with types and labels, a prelabel, and finally a typed hash tree.

There are many equivalent ways of defining an ordered tree. Here we define it as a set of nodes together with a function mapping each node to the sequence of its children.

Definition 4 (Ordered tree). *An ordered tree is a pair $(\mathcal{N}, \mathcal{C})$ where*

1. *$\mathcal{N}$ is a finite set whose elements are called the* nodes *of the tree.*
2. *$\mathcal{C}$ is a function that maps each node $N \in \mathcal{N}$ to a sequence without repetitions, the nodes in the sequence being called the* children *of N, nodes without children being called* leaf nodes *and nodes with children being called* internal nodes*, such that:*
   (a) *Every node N is a child of at most one other node, which is called the parent of N.*
   (b) *The parent-child relation is a directed acyclic graph.*
   (c) *Exactly one node, called the* root *of the tree, has no parent.*

In an ordered tree augmented with types and labels, the types are bit strings of fixed length m, while the labels of some nodes are of variable length less then 2^m and the labels of some other nodes are of fixed length b. These complications are due to the fact that b is the bit length of the output of the hash function, which will grow with the security parameter in the asymptotic security setting of Section 5 while m will be constant.

Definition 5 (Ordered tree augmented with types and labels). *An* ordered tree augmented with types and labels *or, more briefly, an* augmented ordered tree*, is a tuple $(\mathcal{N}, \mathcal{C}, m, d, b, \mathcal{T}, \mathcal{L})$ that augments an ordered tree $(\mathcal{N}, \mathcal{C})$ with the following components:*

1. *A positive integer m, large enough that the number of children of any node can be binary-encoded as m bits (i.e. greater than the base-2 logarithm of the degree of the tree).*

2. *A bit string d of length m, known as the* distinguished type*.*
3. *A positive integer b.*
4. *A function $\mathcal{T} : \mathcal{N} \mapsto \{0,1\}^*$ that maps each node $N \in \mathcal{N}$ to a bit string of length m called the* type *of N, such that every internal node and zero or more leaf nodes have type d, nodes having type d being called* distinguished nodes *and distinguished leaf nodes, if any, being called* dangling nodes*.*
5. *A function $\mathcal{L} : \mathcal{N} \mapsto \{0,1\}^*$ that maps each node $N \in \mathcal{N}$ to a bit string called the* label *of N, such that distinguished nodes have labels of length b and undistinguished nodes have labels of length less than 2^m.*

In the following definition, m determines the length of the length indicator of certain labels, in addition to determining the length of types, and the length of the binary encoding of the number of children of a node (for tree serialization). We let the same parameter m play these three roles for the sake of simplicity. There would be no difficulty in using three different parameters, other than additional complication.

Definition 6 (Prelabel). *Let $T = (\mathcal{N}, \mathcal{C}, m, d, b, \mathcal{T}, \mathcal{L})$ be an augmented ordered tree, and N a node of T. The* prelabel *of N is the bit string obtained by concatenating, for each node N' in the sequence $\mathcal{C}(N)$ of the children of N:*

1. *The type of N', of fixed length m; and*
2. *Either*
   (a) *The label of N', of length b, if the type of N' is d, or*
   (b) *The label of N', of length less than 2^m, preceded by the m-bit binary representation of its length, if the type of N' is other than d.*

In the following definition, the tuple $(\mathcal{N}, \mathcal{C}, m, d, a, b, h, \mathcal{T}, \mathcal{L})$ is a mathematical definition of a typed hash tree X, not a data structure representing X. Implementations of algorithms that create and manipulate typed hash trees may use any data structure suitable to the programming language in which the algorithm is implemented.

Definition 7 (Typed hash tree). *A typed hash tree is a tuple $X = (\mathcal{N}, \mathcal{C}, m, d, a, b, h, \mathcal{T}, \mathcal{L})$ where $(\mathcal{N}, \mathcal{C}, m, d, b, \mathcal{T}, \mathcal{L})$ is an augmented ordered tree and (a, b, h) a simple hashing scheme, such that for every internal node $N \in \mathcal{N}$ the prelabel p of N is a bit string of length less than a and the label of N is $h(p)$.*

We say that two typed hash trees having the same parameters m, d, a, b and h are *of the same kind* and define the *kind of a typed hash tree* accordingly.

Definition 8 (Kind of a typed hash tree). *A* kind of typed hash tree *is a tuple (m, d, a, b, h) where m is a positive integer, d is a bit string of length m, and (a, b, h) is a simple hashing scheme. A typed hash tree $(\mathcal{N}, \mathcal{C}, m, d, a, b, h, \mathcal{T}, \mathcal{L})$ is said to be* of kind *(m, d, a, b, h).*

We need a notion of isomorphism for typed hash trees of a particular kind.

Definition 9 (Isomorphism of typed hash trees). *Let (m, d, a, b, h) be a kind of typed hash tree. We say that two typed hash trees of kind (m, d, a, b, h) are* isomorphic *if there exists a bijection between their sets of nodes such that corresponding nodes have same type, same label, and sequences of corresponding children.*

The *subtree* of a typed hash tree $X = (\mathcal{N}, \mathcal{C}, m, d, a, b, h, \mathcal{T}, \mathcal{L})$ rooted at a node N of X is the typed hash tree $X' = (\mathcal{N}', \mathcal{C}', m, d, a, b, h, \mathcal{T}', \mathcal{L}')$, where $\mathcal{N}'$ is the set of descendants of N (including N itself), and $\mathcal{C}'$, $\mathcal{T}'$ and $\mathcal{L}'$ are the restrictions of the functions $\mathcal{C}$, $\mathcal{T}$ and $\mathcal{L}$ to the domain $\mathcal{N}'$. *Pruning* X' means removing from X the descendants of N other than N itself, and thus results in the typed hash tree $X'' = (\mathcal{N}'', \mathcal{C}'', m, d, a, b, h, \mathcal{T}'', \mathcal{L}'')$, where $\mathcal{N}'' = (\mathcal{N} \setminus \mathcal{N}') \cup \{N\}$, $\mathcal{C}'' = (\mathcal{C} \setminus \mathcal{C}') \cup \{(N, \emptyset)\}$, and $\mathcal{T}''$ and $\mathcal{L}''$ are the restrictions of the functions $\mathcal{T}$ and $\mathcal{L}$ to the domain $\mathcal{N}''$. A *pruned derivative* of X is the result of pruning zero or more subtrees from X. If X'' is the result of pruning X' from X, we also say that X is the result of *grafting* X' onto X''.

The labels of the internal nodes of a typed hash tree can be computed from the types and labels of the leaf nodes, hence they can be omitted when the tree is serialized. Algorithm 1 below attempts to compute the internal-node labels of a typed hash tree where they are missing either because the tree is under construction, or because they have been omitted. In the former case it may fail and stop without output if it encounters a prelabel that is too long. The details of the algorithm are worth noting because a trace of the algorithm is used in the proof of Theorem 1.

We need a formal notion of an *incomplete typed hash tree without internal-node labels* to characterize the input to the label computation algorithm. A similar notion of and *incomplete typed hash tree without distinguished-node labels* will be needed later for other purposes. We provide both definitions together.

Definition 10 (Incomplete typed hash tree without internal-node or distinguished-node labels). *An* incomplete typed hash tree without internal-node labels *(resp. without* distinguished-node labels*) is a tuple $(\mathcal{N}, \mathcal{C}, m, d, b, \mathcal{T}, \mathcal{L})$ (resp. $(\mathcal{N}, \mathcal{C}, m, d, \mathcal{T}, \mathcal{L})$) where $\mathcal{N}$, $\mathcal{C}$, m, d, b, $\mathcal{T}$, and $\mathcal{L}$ (resp. $\mathcal{N}$, $\mathcal{C}$, m, d, $\mathcal{T}$ and $\mathcal{L}$) are as in Definition 5, except that $\mathcal{L}$ is a partial function that only assigns labels to leaf nodes (resp. distinguished nodes).*

We also need definitions of a *kind of incomplete typed hash tree without internal-node labels* and a *kind of incomplete typed hash tree without distinguished-node labels*, and corresponding definitions of isomorphism.

Definition 11 (Kinds and isomorphism of incomplete typed hash trees). *A* kind of typed hash tree without internal-node labels *(resp. without distinguished-node labels) is a tuple (m, d, b) (resp. (m, d) where m and b are positive integers (resp. m is a positive integer) and d is a bit string of length m. An incomplete typed hash tree $(\mathcal{N}, \mathcal{C}, m, d, b, \mathcal{T}, \mathcal{L})$ resp. $(\mathcal{N}, \mathcal{C}, m, d, \mathcal{T}, \mathcal{L})$ without internal-node labels (resp. undistinguished node labels) is said to be of kind (m, d, b). (resp.*

(m, d)). *Two incomplete typed hash trees without internal-node labels (resp. without distinguished-node labels) of the same kind are* isomorphic *if there exists a bijection between their sets of nodes such that corresponding nodes have the same type, the same set of labels (comprising zero or one labels), and sequences of corresponding children.*

We will make use of the following fact.

Fact 1. *Let X and Y be incomplete typed hash trees without internal-node labels of kind (m, d, b), and let X' and Y' be incomplete typed hash trees without distinguished-node labels derived from X and Y by removing the labels of the internal nodes. Then if X and Y are isomorphic as incomplete typed hash trees without internal-node labels of kind (m, d, b), X' and Y' are isomorphic as incomplete typed hash trees without distinguished-node labels of kind (m, d).*

Algorithm 1 (Label computation). *Let (m, d, a, b, h) be a kind of typed hash tree. The* (m, d, a, b, h)-label computation algorithm*, takes as input an incomplete typed hash tree X without internal-node labels of kind m, d, b, maintains a state consisting of a stack of type-label pairs and a node list, and performs the following steps:*

1. *Initialize the stack to be empty, and the node list to the list of all the nodes in the tree in depth-first post-order.*
2. *While the node list is not empty, perform a step, said to be "triggered" by the first node N in the list, as follows:*
   (a) *If N has no children, push a type-label pair with its type and its label onto the stack.*
   (b) *If N has $n > 0$ children, the type-label pairs of those children can be found at the top of stack. Use them to assemble the prelabel p. Stop without output if the bit length of p is not less than a. Otherwise compute $l = h(p)$, augment X by assigning l as the label of N, pop the top n entries from the stack, and push the type-label pair (d, l) onto the stack.*
   (c) *Remove N from the node list.*
3. *Output X, as augmented.*

Algorithm 1 outputs a typed hash tree of kind (m, d, a, b, h) unless it encounters a prelabel that is too long.

The omission-tolerant integrity protection provided by the root label of a typed hash tree is based on the following fundamental theorem about the structure of typed hash trees that have the same root label.

Theorem 1 (Fundamental theorem: typed hash trees with same root label). *Let X and Y be two typed hash trees of the same kind. If X and Y have the same root label, then either (i) Y is isomorphic to a pruned derivative of X, or (ii) the label of an internal node of Y is equal to the label of a dangling node of X, or (iii) an internal node of Y has the same label as an internal node of X but a different prelabel.*

Proof. Let X and Y be two typed hash trees of kind $\mathcal{K} = (m, d, a, b, h)$ that have the same root label. To prove that Y satisfies one of the conditions (i), (ii) or (iii) of the theorem we consider the computation of the internal labels of the nodes of Y, X, and various pruned derivatives of X using Algorithm 1. Strictly speaking, Algorithm 1 takes as input an incomplete typed hash tree without internal-node labels. For our present purposes we let it take as input a complete typed hash tree, ignoring and recomputing its internal-node labels.

Recall that Algorithm 1 maintains a state consisting of a stack of type-label pairs and a node list. We define the *label computation trace (LCT)* of a typed hash tree as the sequence of states in an execution of the algorithm when given the tree as input, except that the node list contains entries corresponding to the nodes instead of the nodes themselves, each entry being a tuple consisting of the type, the number of children, and the label of the node, except that the label is omitted if the number of children is zero; thus the entry corresponding to an internal node N is a pair (d, n_N), where d is the distinguished type and n_N is the number of children of N. We still refer to nodes, rather than node entries, as "triggering" the steps of the algorithm.

The stack is empty in the initial state and never becomes empty again. In the final state it contains a single type-label pair, where the type is d and the label is the root label. The node list contains entries for all the nodes in the initial state, and is empty in the final state. It is easy to see that the LCT is entirely determined by its initial state, and that two typed hash trees of kind $\mathcal{K}$ with the same LCT are isomorphic.

Let τ be the LCT of Y. Since X and Y have the same root label, the last state of τ is also the last state of the LCT of X. Let σ be the earliest state of τ that is also a state of the LCT of a pruned derivative of X and let X' be a minimally pruned such derivative, i.e. let X' be a pruned derivative of X that has σ as a state of its LCT and is not a pruned derivative of another pruned derivative of X that also has σ as a state of its LCT. Let τ' be the LCT of X'.

If the stack is empty in state σ, σ is the initial state of both τ and τ', hence τ and τ' are identical and Y and X' are isomorphic. Therefore Y satisfies condition (i) of the theorem.

If the stack is not empty in state σ, let (t, l) be the type-label pair at the top of the stack. Let N be the node of Y and N' the node of X' that trigger the steps leading to state σ in the label computations of Y and X' respectively. N and N' both have type t and label l.

Type t must be d, because otherwise N and N' would be leaf nodes of Y and X', and the state preceding σ would be the same in τ and τ', contradicting the definition of σ as the earliest state of τ that is also a state of the LCT of a pruned derivative of X (such as X'). Therefore each of N and N' must be either an internal node or a dangling node.

N' cannot be an internal node of X' if N is a dangling node of Y. Otherwise let X'' be the pruned derivative X'' of X obtained by further pruning from X' the subtree rooted at N'. N' would be a dangling node of X'' triggering a step leading to σ in the label computation of X''. Since a label computation proceeds

in depth-first post order, the LCTs of X'', X and Y would be identical from state σ onward. But then, N' and N being leaf nodes of X'' and Y, the states preceding σ would be the same in X'' and Y, contradicting the definition of σ as the earliest state of τ that is also a state of the LCT of a pruned derivative of X (such as X'').

This leaves two possibilities. N may be an internal node of Y while N' is a dangling node of X', or both may be internal nodes.

Consider first the case where N is an internal node of Y and N' is a dangling node of X'. N' must also be a dangling node of X, for otherwise the tree X''' derived from X' by grafting the subtree of X rooted at N' would be a pruned derivative of X having σ as a state of its LCT, and X' would be a pruned derivative of X''' distinct from X''', contradicting the minimality of X'. Therefore N' is a dangling node of X having the same label l as the internal node N of Y, hence Y satisfies condition (ii) of the theorem.

Now consider the case where N and N' are internal nodes of Y and X'. Let n_N be the number of children of N in Y and $n_{N'}$ the number of children of N' in X'. Let ρ_N and $\rho_{N'}$ be the states that precede σ in the LCTs of Y and X' respectively, with N triggering the transition from ρ_N to σ in the LCT of Y and N' the transition from $\rho_{N'}$ to σ in the LCT of X'.

By the definition of σ, ρ_N and $\rho_{N'}$ must be different. This implies that the sequences of type-label pairs of the children of N and N' must be different, since the stack of σ is derived by popping the type-label pairs of the children of N from ρ_N before pushing (t, l) as well as by popping the type-label pairs of the children of N' from $\rho_{N'}$ before pushing (t, l), and the node list of σ is derived by removing the entry (d, n_N) from the front the node list of ρ_N as well as by removing the entry $(d, n_{N'})$ from the front of the node list of $\rho_{N'}$; so if the sequences of type-label pairs of the children of N and N' were the same, both the states and the node lists of ρ_N and $\rho_{N'}$ would be the same, and thus ρ_N and $\rho_{N'}$ would be the same.

But if the sequences of type-label pairs of the children of N and N' are different, so must be the prelabels of N in Y and N' in X'. Hence N is an internal node of Y having the same label l as the internal node N' of X' but a different prelabel. But it follows from the fact that N' is an internal node of a pruned derivative of X that N' is also an internal node of X and has the same prelabel in X as in X'. Thus an internal node of Y has the same label as an internal node of X but a different prelabel, and Y satisfies condition (iii) of the theorem. □

4 An omission-tolerant checksum

4.1 Overview

The root label of a typed hash tree can be used as an omission-tolerant checkum of a bit string encoding of a finite set of key-value pairs, as follows. A nondeterministic algorithm is used to produce an incomplete typed hash tree without distinguished-node labels that represents the set of key-value pairs. The

representation algorithm is free to choose the form of the tree, and may choose to include dangling nodes. If the tree has dangling nodes, random labels are assigned to them to obtain an incomplete typed hash tree without internal-node labels. That tree is then serialized to produce the bit string encoding of the set of key-value pairs. A checksum computation algorithm applies to the bit string encoding, deserializes it to obtain an incomplete typed hash tree without internal-node labels, uses the label computation algorithm to compute the labels of the internal nodes, and outputs the label of the root node as the checksum.

In this section we define the representation algorithm, a derepresentation algorithm, and the the serialization and deserialization algorithms. The encoding and checksum algorithms are defined in Section 5 in an asymptotic security setting.

4.2 Representation of a set of key-value pairs

When a typed hash tree is used to represent a set of key-value pairs, the key and the value of each pair are represented as the type and the label of an undistinguished leaf node. But in order to represent a key as a type and a value as a label, the key and the value must be encoded as bit strings. This motivates the following definitions.

Definition 12 (Space of key-value pairs). *A* space of key-value pairs *is a pair (K, V), where K is a finite non-empty set of elements called keys and $V = (V_k)_{k \in K}$ is a family of finite non-empty sets, the elements of V_k being the values that can be associated with key k. In the context of such a space, a key-value pair is an element of the cartesian product $\prod_{k \in K} V_k$, and a set of key-value pairs is an element of the powerset of the cartesian product, which we write* $\mathrm{Sets}_{K,V} = \mathbf{P}(\prod_{k \in K} V_k)$.

Definition 13 (Representation framework for sets of key-value pairs). *A* representation framework for sets of key-value pairs *is a tuple $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ where*

1. *(m, d) is a kind of incomplete typed hash tree without distinguished-node labels.*
2. *(K, V) is a space of key value pairs.*
3. *KE and KD are the encoding and decoding algorithms of a simple encoding scheme $(K, \mathrm{KE}, \mathrm{KD})$ such that, for every $k \in K$, on input k, KE outputs a bit string of length m other than d.*
4. *$\mathrm{VE} = (\mathrm{VE}_k)_{k \in K}$ and $\mathrm{VD} = (\mathrm{VD}_k)_{k \in K}$ are families of algorithms such that, for all $k \in K$, $(V_k, \mathrm{VE}_k, \mathrm{VD}_k)$ is a simple encoding scheme such that, for all $v \in V_k$, VE_k outputs a bit string of length less than 2^m on input v.*

We can now define the representation algorithm, using a representation framework $\mathcal{F}$ as a parameter.

Algorithm 2 ($\mathcal{F}$-representation of a set of key-value pairs). *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs. The $\mathcal{F}$-representation algorithm takes as input an element x of $\mathrm{Sets}_{K,V}$ and outputs an incomplete typed hash tree without distinguished-node labels of kind (m, d), constructed by the following steps:*

1. *Construct an ordered tree X having a number of leaf nodes at least equal to the cardinality $|x|$ of x.*
2. *For each key-value pair $(k, v) \in x$, run* KE *on input k and* VE_k *on input v and assign their outputs to the type and label of a leaf node that has not been assigned a type yet.*
3. *For each node N that has not been assigned a type yet (including all internal nodes and possibly some of the leaf nodes), assign d as the type of N.*
4. *Output X, which is now an incomplete typed hash tree without distinguished-node labels of kind (m, d).*

The above $\mathcal{F}$-representation algorithm is non-deterministic, as it makes non-probabilistic choices in step 1, then calls KE and VE, which may make probabilistic and/or non-probabilistic choices. It has a deterministic inverse, which we call the *$\mathcal{F}$-derepresentation algorithm.*

Algorithm 3 ($\mathcal{F}$-derepresentation). *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs. The $\mathcal{F}$-derepresentation algorithm takes as input an incomplete typed hash tree without distinguished-node labels X of kind (m, d) and performs the following steps to construct a set $x \in \mathrm{Sets}_{K,V}$.*

1. *Let the initial value of x be $\emptyset$.*
2. *Traverse X, enumerating its nodes in depth-first post-order. (The traversal order is immaterial, but one must be specified if the algorithm is to be deterministic.) For each undistinguished node N do the following, where l is the label of N:*
   (a) *Run* KD *on input t and stop without output if* KD *stops without output, or else let $k \in K$ be the output of* KD.
   (b) *Run* VD_k *on input l and stop without output if* VD_k *stops without output, or else let $v \in V_k$ be the output of* VD_k.
   (c) *Stop without output if (k, v) is already in x, or else add (k, v) to x.*
3. *Output x.*

Fact 2. *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, let R and D be the $\mathcal{F}$-representation and derepresentation algorithms, and let X be an incomplete typed hash tree without distinguished-node labels of kind (m, d). Then:*

1. *If X is an output of R on input $x \in \mathrm{Sets}_{K,V}$, then D outputs x on input X.*
2. *If X is not an output of R on any input, then D stops without output.*

Fact 3. *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, let D be $\mathcal{F}$-derepresentation algorithm, and let X and Y be incomplete typed hash trees without distinguished-node labels of kind (m, d). If D outputs x on input X and Y is isomorphic to X, then D outputs x on input Y.*

Fact 4. *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, and let R and D be the $\mathcal{F}$-representation and derepresentation algorithms. Let x be an element of $\mathrm{Sets}_{K,V}$, X an incomplete typed hash tree of kind (m, d) output by R on input x, and Y an incomplete typed hash tree of kind (m, d) isomorphic to X. Then D outputs x on input Y.*

Fact 5. *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, let X and Y be typed hash trees of kind (m, d, a, b, h), let X' and Y' be the incomplete typed hash trees obtained by removing the distinguished-node labels from X and Y, and let x and y be outputs of the $\mathcal{F}$-derepresentation algorithm on inputs X' and Y' respectively. If Y is a pruned derivative of X, then $y \subseteq x$.*

Algorithm 4 ((m, d, b)-serialization of an incomplete typed hash tree without internal node labels). *Let (m, d, b) be a kind of incomplete typed hash tree without internal-node labels. The* (m, d, b)-serialization algorithm *takes as input an incomplete typed hash tree X without internal-node labels of kind (m, d, b) and outputs a bit string computed by the following steps:*

1. *Initialize a bit string s to have length 0.*
2. *Traverse X in depth-first post-order and, for each node N, append to s a node description string consisting of:*
   (a) *The type of N, which is an m-bit string.*
   (b) *The m-bit binary representation of the number of children of N.*
   (c) *If N is a dangling node (i.e. if the type of N is d and the number of children is 0), the label of N, which is a bit string of length b.*
   (d) *If N is an undistinguished leaf node, the label of N preceded by the m-bit representation of its bitlength.*
3. *Output s.*

Algorithm 5 ((m, d, b)-deserialization of an incomplete typed hash tree without internal node labels). *Let (m, d, b) be a kind of incomplete typed hash tree without internal-node labels. The* (m, d, b)-deserialization algorithm *takes as input a bit string s and performs the following steps:*

1. *Attempt to parse s as a concatenation of node description strings and construct a sequence r of corresponding node description entries, each having components specifying a type, a number of children, and a label if the number of children is not 0, as follows:*
   (a) *Initialize r to be empty.*
   (b) *While s is not empty do the following:*
      i. *Create a node description entry e.*

   ii. *Stop without output if the length of s is less than m; otherwise let the bit string comprising the first m bits of s be the type component of e and remove the bits from s.*
   iii. *Stop without output if the length of s is less than m; otherwise parse the first m bits of s as a nonnegative integer n, let n be number-of-children component of e, and remove the bits from s.*
   iv. *If $n = 0$, stop without output if the length of s is less than m; otherwise parse the first m bits of s as a nonnegative integer l and remove those bits from s; then let the bit string comprising the first l bits of s be the label component of e, stopping without output if the length of s is less than l.*
   v. *Append e to r.*
2. *Attempt to construct an incomplete typed hash tree without internal node labels X of kind (m, d, b) as follows:*
   (a) *Initialize to empty a stack that will be used to store nodes of X temporarily as they are created by the algorithm.*
   (b) *Initialize the set of nodes of X as being empty.*
   (c) *While the sequence r is non-empty do the following:*
      i. *If the number-of-children component of the first entry of r is 0, create a node, add it to the set of nodes of X, assign to it the type and the label specified by the entry, and push it onto the stack.*
      ii. *If the number-of-children component of the first entry of r is $n > 0$:*
         A. *Stop without output if the type specified by the entry is not d.*
         B. *Create a node N, add it to the set of nodes of X, and assign to it the type specified by the entry.*
         C. *Stop without output if there are fewer than n nodes in the stack; otherwise assemble a sequence of the top n nodes in the stack and assign it to N as the sequence of its children.*
         D. *Pop the top n nodes off the stack and push N onto the stack.*
         E. *Remove the first entry from r.*
   (d) *Stop without output if the stack does not contain a single node.*
3. *Output X.*

Fact 6. *Let (m, d, b) be a kind of incomplete typed hash tree without internal-node labels, let S and D be the (m, d, b)-serialization and deserialization algorithms, and let s a bit string. Then:*

1. *If s is an output of S on input X, where X is an incomplete typed hash tree without internal-node labels of kind (m, d, b), then D outputs an incomplete typed hash tree Y without internal-node labels of kind (m, d, b) isomorphic to X on input s.*
2. *If s is not an output of S on any input, then D stops without output.*

The serialization algorithm is deterministic, but the deserialization algorithm is non-deterministic because it does not specify what the nodes that it creates are. (It could be made deterministic by specifying that the nodes are, for example, integers denoting positions in a total order such as depth-first post order; but

that is unnecessary.) But it is deterministic up to isomorphism in the following sense:

Fact 7. *Let (m, d, b) be a kind of incomplete typed hash trees without internal-node labels, and s a bit string. If X and Y are outputs of the (m, d, b)-deserialization algorithm on input s, then X and Y are isomorphic as incomplete typed hash trees without internal-node labels of kind (m, d, b).*

5 Asymptotic security

5.1 Boneh and Shoup's system parameterization

In an asymptotic security setting, a cryptographic scheme is said to be secure if an adversary has a *negligible probability* of breaching a specified security property. This requires a *security parameter*, so that a "negligible probability" can be defined as a probability that is asymptotically negligible as a function of the security parameter. It also requires a probability space, which in a traditional asymptotic security setting is defined by randomness inherent in the scheme. In [2, §2.4.2], Boneh and Shoup define a novel asymptotic security setting that uses both a security parameter λ and an additional *system parameter* Λ, output by a *system parameterization algorithm* P on input λ. The algorithm P is probabilistic, and thus provides a probability space independent of any randomness inherent in the scheme. This is essential if the scheme itself has no randomness.

In particular, randomized system parameterization makes it possible to provide a formal definition of collision resistance for a keyless hash function. Collision resistance in an asymptotic security setting is traditionally defined for keyed rather than keyless hash functions, and introduce a probability space by letting the key be random. Boneh and Shoup define a keyless hash function as an algorithm that takes as inputs λ and Λ in addition to the message m. They say on one hand that this is "really the same" as the keyed hash function approach, where the random key can be viewed as a system parameter; but they also suggest, on the other hand, that each choice of system parameter can be viewed as describing a different hash function. We adopt the latter view by referring to the algorithm that takes as inputs λ and Λ as a *hashing scheme* rather than a hash function, and using the hashing scheme to define a family of ordinary keyless hash functions parameterized by the security and system parameters.

We find the idea of a family of hash functions indexed by security and system parameters intuitive and close to actual practice. After all, SHA-2 is a family of hash functions defined for several security parameters, whose design includes various constants that could have been chosen at random. More generally, one could think of the randomness of a system parameterization algorithm taken as input by a cryptographic scheme as the randomness of the process of technological evolution resulting in the design of the scheme.

Besides the above-mentioned change in terminology, we simplify Boneh and Shoup's treatment of hash functions by defining the domain and codomain of a hash functions in terms of bit lengths, as is done in actual practice, and we

provide a formal definition of preimage resistance, which is not provided in [2]. We also adapt and simplify Boneh and Shoup's complex definition of an *efficient algorithm*, which they allow to not terminate [2, Definition 2.8]. We say that an algorithm is efficient if it terminates and its run time is bounded by a polynomial function of the security parameter, independently of the length of other inputs; we can do this because, as explained above in Section 2.1, our algorithms stop without output when given unexpected inputs, and may thus stop without output when given inputs that are too long.

Definition 14 (Efficient algorithm). *We say that an algorithm A that takes as its first input a security parameter λ is* efficient *if there exists a polynomial p such that, for all $\lambda \in \mathbf{Z}_{\geq 1}$, the running time of A until it stops with or without output is less than $p(\lambda)$.*

Boneh and Shoup define a system parameter Λ as a bit string that is efficiently generated by a probabilistic algorithm P when given as input a security parameter λ and has a length bounded by a polynomial function of λ . They use the notation $\mathrm{Supp}(P(\lambda))$ to refer to the support of the distribution of Λ, i.e. to the range of possible values that P may output on input λ. We must require that it be possible to determine efficiently if $\Lambda \in \mathrm{Supp}(P(\lambda))$, so that an efficient algorithm that takes λ and Λ as inputs can stop without output if Λ is not in its expected range, in accordance with our definition in Section 2.1 of what it means to take as an input data of some particular form. Hence the following definition.

Definition 15 (System parameterization). *A system parameterization P is an efficient probabilistic algorithm that takes as input a* security parameter *$\lambda \in \mathbf{Z}_{\geq 1}$ and outputs a bit string Λ called a* system parameter*, such that*

1. *there exists a polynomial p such that the length of Λ is less than $p(\lambda)$ for all $\lambda \in \mathbf{Z}_{\geq 1}$, and*
2. *there exists an efficient deterministic algorithm that takes as inputs $\lambda \in \mathbf{Z}_{\geq 1}$ and $\Lambda \in \{0,1\}^*$ and outputs a boolean value indicating whether $\Lambda \in \mathrm{Supp}(P(\lambda))$.*

Definition 16 (Hashing scheme with system parameterization). *A* hashing scheme *with system parameterization P is a tuple $\mathcal{H} = (P, A, B, H)$ where:*

1. *P be a system parameterization.*
2. *A and B are indexed families of positive integers A_λ, B_λ, with $\lambda \in \mathbf{Z}_{\geq 1}$. ‘*
3. *H is an efficient deterministic algorithm that takes as inputs $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\lambda))$ and a bit string $x \in \{0,1\}^{<a}$, and outputs a bit string $y \in \{0,1\}^b$, where $a = A_\lambda$ and $b = B_\lambda$.*

Definition 17 (Family of hash functions defined by a hashing scheme with system parameterization). *A hashing scheme $\mathcal{H} = (P, A, B, H)$ with system parameterization P defines an indexed family of hash functions $\mathcal{H}_{\lambda,\Lambda}$, where each $\mathcal{H}_{\lambda,\Lambda}$ has domain $\{0,1\}^{<a}$ with $a = A_\lambda$ and codomain $\{0,1\}^b$ with $b = B_\lambda$, and maps each $x \in \{0,1\}^{<a}$ to the output of H on inputs λ, Λ and x.*

(In the above definition the notation $\mathcal{H}$ is used to refer both to the hashing scheme and to the family of hash functions that it defines, but this abuse of notation should create no confusion.)

The following attack game is used to define collision resistance for a hashing scheme.

Game 1 (Attack on collision resistance).

Game parameters:

1. *A hashing scheme with system parameterization* $\mathcal{H} = (P, A, B, H)$.
2. *A security parameter* $\lambda \in \mathbf{Z}_{\geq 1}$.

Game protocol:

1. *The challenger runs* P *on input* λ*, and* P *outputs* Λ.
2. *The challenger runs the adversary on input* λ *and* Λ.
3. *Let* $a = A_\lambda$ *and* $b = B_\lambda$. *The adversary wins the game if it outputs a collision of the hash function* $h = \mathcal{H}_{\lambda,\Lambda}$*, i.e. a pair of distinct elements* (u, u') *of* $\{0,1\}^{<a}$ *such that* $h(u) = h(u')$.

Definition 18 (Collision resistance of a hashing scheme with system parameterization). *A hashing scheme* $\mathcal{H} = (P, A, B, H)$ *with system parameterization* P *is collision resistant if for every efficient adversary* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *wins Game 1 with parameters* $\mathcal{H}$ *and* λ *is a negligible function of* λ.

The following attack game is used to define preimage resistance for a hashing scheme.

Game 2 (Preimage resistance). *[Attack on preimage resistance]*

Game parameters:

1. *A hashing scheme with system parameterization* $\mathcal{H} = (P, A, B, H)$.
2. *A security parameter* $\lambda \in \mathbf{Z}_{\geq 1}$.

Game protocol:

1. *The challenger runs* P *on input* λ*, and* P *outputs* Λ.
2. *Let* $b = B_\lambda$. *The challenger chooses an element* v *of* $\{0,1\}^b$ *at random with uniform distribution.*
3. *The challenger runs the adversary on inputs* λ*,* Λ *and* v.
4. *Let* $a = A_\lambda$. *The adversary wins the game if it outputs an element* u *of* $\{0,1\}^{<a}$ *such that* $h(u) = v$.

Definition 19 (Preimage resistance). *A hashing scheme* $\mathcal{H} = (P, A, B, H)$ *with system parameterization* P *is preimage resistant if for every efficient adversary* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *wins Game 2 is a negligible function of* λ.

5.2 Encoding schemes with system parameterization

Definition 20 (Encoding scheme with system parameterization). *An* encoding scheme with system parameterization *is a tuple* $\mathcal{E} = (P, S, E, D)$ *where*

1. P *is a system parameterization.*
2. S *is the set of data items to be encoded.*
3. E *is an efficient algorithm, possibly non-deterministic, called the* encoding algorithm *of the scheme, that takes as inputs* $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\lambda))$, *and an element* x *of* S *and outputs a bit string* y *called an* E-encoding of x for security and system parameters (λ, Λ).
4. D *is an efficient deterministic algorithm, called the* decoding algorithm *of the scheme, that takes as inputs* $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\lambda))$ *and a bit string* y *and*
   (a) *outputs an element* x *of* S *if* y *is an output of* E *on inputs* λ, Λ *and* x, *or*
   (b) *stops without output otherwise.*

Definition 21 (Checksum scheme with system parameterization). *A* checksum scheme with system parameterization *is a pair* (P, C) *where*

1. P *is a system parameterization.*
2. C *is an efficient deterministic algorithm that takes as inputs* $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\lambda))$, *and a bit string, and outputs a bit string or stops without output.*

A checksum scheme with system parameterization (P, C) may be used in conjunction with an encoding scheme (P, S, E, D) if both have the same system parameterization P. Then, if E outputs y on inputs λ, Λ and an element x of S, and C outputs z on inputs λ, Λ and y, z is said to be an (E, C)*-checksum of* x *for security and system parameters* (λ, Λ).

As a warm-up for the proof of Theorem 3, we now prove that a hashing scheme with system parameterization provides full integrity protection for an encoding scheme with system parameterization, in the special case where the encoding algorithm does not make any non-probabilistic choices, i.e. in the case where it is deterministic or probabilistic.

Fact 8. *If* (P, A, B, H) *is a hashing scheme with system parameterization, then* (P, H) *is a checksum scheme with system parameterization.*

Game 3 (Attack on full integrity protection).

Game parameters:
1. *An encoding scheme with system parameterization* $\mathcal{E} = (P, S, E, D)$ *where the encoding algorithm* E *is deterministic or probabilistic.*
2. *A checksum scheme with system parameterization* $\mathcal{C} = (P, C)$.
3. *An element* x *of* S.
4. *A security parameter* $\lambda \in \mathbf{Z}_{\geq 1}$.

Game protocol:
1. *The challenger runs* P *on input* λ, *and* P *outputs* Λ.

2. *The challenger runs E on inputs λ, Λ and x, and E outputs a bit string y.*
3. *The challenger runs C on inputs λ, Λ and y, and either C stops without output or it outputs a bit string z.*
4. *If C stops without output, the challenger stops without invoking the adversary and the adversary is deemed to have lost the game.*
5. *The challenger runs the adversary on inputs λ, Λ and y. (The adversary may run C on inputs λ, Λ and y to obtain z.)*
6. *The adversary wins the game if it outputs a bit string y' satisfying the following winning conditions:*
   (a) *C outputs the same bit string z on inputs λ, Λ and y' as on inputs λ, Λ and y.*
   (b) *y' is an E-encoding of an element x' of S for security and system parameters (λ, Λ), computable by running D on inputs λ, Λ and y'.*
   (c) *$x' \neq x$.*

Definition 22 (Full integrity protection with system parameterization). *Let $\mathcal{E} = (P, S, E, D)$ be an encoding scheme with system parameterization where E is deterministic or probabilistic. A checksum scheme $\mathcal{C} = (P, C)$ is said to provide* full integrity protection *for $\mathcal{E}$ if for every $x \in S$ and every efficient adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins Game 3 with parameters $\mathcal{E}$, $\mathcal{C}$, x and $\lambda \in \mathbf{Z}_{\geq 1}$ is a negligible function of λ.*

Theorem 2 (Integrity protection provided by a collison-resistant hashing scheme). *Let $\mathcal{E} = (P, S, E, D)$ be an encoding scheme with system parameterization where E is deterministic or probabilistic. If $\mathcal{H} = (P, A, B, H)$ is a collision-resistant hashing scheme, then the checksum scheme $\mathcal{C} = (P, H)$ provides full integrity protection for $\mathcal{E}$.*

Proof. Let $\mathcal{E}$, $\mathcal{H}$ and $\mathcal{C}$ be as stated. Reasoning by contradiction, assume that $\mathcal{C}$ does not provide integrity protection for $\mathcal{E}$. This means that there exists an element x of S and an efficient adversary $\mathcal{A}$ of Game 3 such that the probability $p(\lambda)$ that $\mathcal{A}$ wins Game 3 with parameters $\mathcal{E}$, $\mathcal{C}$, x and $\lambda \in \mathbf{Z}_{\geq 1}$ is not a negligible function of λ. Let $\bar{x}$ and $\bar{\mathcal{A}}$ be particular selections of such an x and $\mathcal{A}$.

For every $\lambda \in \mathbf{Z}_{\geq 1}$, consider an execution of Game 1 with parameters $\mathcal{H} = (P, A, B, H)$ and λ. The challenger of Game 1 runs P on input λ, and P outputs Λ; then the challenger runs the adversary on inputs λ and Λ. Consider a particular adversary $\mathcal{A}_{\mathrm{CR}}$ of Game 1 defined as follows. $\mathcal{A}_{\mathrm{CR}}$ plays Game 3 with parameters $\mathcal{E}$, $\mathcal{C}$, x and λ, instructing the challenger to use $\bar{\mathcal{A}}$ as the adversary, and modifying the challenger so that it uses the system parameter Λ produced by the challenger of Game 1 instead of running P to generate Λ. This modification makes no difference to $\bar{mathcal}A$, which still has a probability $p(\lambda)$ of winning Game 3. $\mathcal{A}_{\mathrm{CR}}$ further modifies the challenger of Game 3 to output the pair (y, y') whenever the adversary of Game 3 wins the game, and $\mathcal{A}_{\mathrm{CR}}$ relays such pair as its own input in Game 1; so with probability $p(\lambda)$, $\mathcal{A}_{\mathrm{CR}}$ outputs a pair (y, y') where

1. y is an output of E on inputs λ, Λ and $\bar{x}$, by construction;
2. H outputs the same bit string z on inputs λ, Λ and y as on inputs λ, Λ and y', by the first winning condition of Game 3, i.e. $h(y) = z = h(y')$ with $h = \mathcal{H}_{lL}$;
3. D outputs an element x' of S on inputs λ, Λ and y', by the second winning condition of Game 3; and
4. $x' \neq \bar{x}$ by the third winning condition of Game 3.

Since y is an output of E on inputs λ, Λ and $\bar{x}$, $\bar{x}$ is an output or D on inputs λ, Λ and y. Hence since x' is an output of D on inputs λ, Λ and y', D is deterministic, and $x' \neq \bar{x}$, we have that $y' \neq y$. And since $h(y) = h(y')$, (y, y) is a collision of h. Therefore $\mathcal{A}_{\mathrm{CR}}$ wins Game 1.

Furthermore $\mathcal{A}_{\mathrm{CR}}$ is an efficient algorithm, because the algorithms E and H used by the modified challenger of Game 3 and $\bar{\mathcal{A}}$ are efficient. Thus $\mathcal{A}_{\mathrm{CR}}$ is an efficient adversary that wins Game 1 with non-negligible probability ($p(\lambda)$, contradicting the hypothesis that $\mathcal{H}$ is collision resistant. □

5.3 Omission-tolerant integrity protection in the asymptotic security setting

In Game 3 the probability that the adversary wins the game would not be well defined if the encoding algorithm were allowed to make non-probabilistic choices, because those choices would be made after the probabilistic choices made by P for the computation of Λ. Since omission-tolerant integrity protection is achieved by representing data as a typed hash tree whose construction involves non-probabilistic choices, we must now find a way of allowing the encoding algorithm to make such choices.

We do that by splitting the encoding algorithm into two phases, a first phase E_{I} that is allowed to make non-probabilistic choices but does not take as inputs security and system parameters, and a second phase E_{II} that takes as inputs security and system parameters but is only allowed to make probabilistic choices. The output E_{I} need not be a bit string, so we refer to E_{I} as the *representation phase* while we refer to E_{II} as the *encoding phase*. Since the output of E_{I} does not depend of the security parameter, we refer to it as a *fixed representation* of the input.

Definition 23 (Two-phase split). *A* two-phase split of an encoding algorithm with system parameterization *is a tuple* $(P, S, E_{\mathrm{I}}, E_{\mathrm{II}})$ *where*

1. P *is a system parameterization.*
2. S *is a set of data items to be encoded.*
3. E_{I} *is an algorithm, possibly exhibiting hybrid non-determinism, that takes as input an element* x *of* S *and outputs a data structure* X *known as a* fixed representation *of* x*.*
4. E_{II} *is a deterministic or probabilistic algorithm that takes as inputs* $\lambda \in \mathbf{Z}_{\geq 1}$*,* $\Lambda \in \mathrm{Supp}(P(\lambda))$*, and a fixed representation* X *of an element* x *of* S *and outputs a bit string* y *known as an* encoding *of* x *for security and system parameters* (λ, Λ)*.*

Algorithm 6 (Two-phase encoding). *Let $\mathcal{S} = (P, S, E_{\mathrm{I}}, E_{\mathrm{II}})$ be a two-phase split of an encoding algorithm with system parameterization. The* two-phase encoding algorithm with system parameterization *defined by $\mathcal{S}$ takes as inputs $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\lambda))$, and an element x of S and performs the following steps:*

1. *Run E_{I} on input x and let X be the output.*
2. *Run E_{II} on inputs λ, Λ, and X and let y be the output.*
3. *Output y.*

The following attack game is used in the definition of omission-tolerant integrity protection.

Game 4 (Attack on omission-tolerant integrity protection).

Game parameters:

1. *An encoding scheme with system parameterization $\mathcal{E} = (P, S, E, D)$ where the elements of S are sets and E is a two-phase encoding algorithm defined by a two-phase split $(P, S, E_{\mathrm{I}}, E_{\mathrm{II}})$.*
2. *A checksum scheme with system parameterization $\mathcal{C} = (P, C)$.*
3. *An element x of S.*
4. *A fixed representation X of x, i.e. an output X of E_{I} on input x.*
5. *A security parameter $\lambda \in \mathbf{Z}_{\geq 1}$.*

Game protocol:

1. *The challenger runs P on input λ, and P outputs Λ.*
2. *The challenger runs E_{II} on inputs λ, Λ and X, and E_{II} outputs a bit string y.*
3. *The challenger runs C on inputs λ, Λ and y, and either C stops without output or it outputs a bit string z.*
4. *If C stops without output, the challenger stops without invoking the adversary and the adversary is deemed to have lost the game.*
5. *The challenger runs the adversary on inputs λ, Λ and y. (The adversary can compute Z and z from y as needed.)*
6. *The adversary wins the game if it outputs a bit string y' satisfying the following winning conditions:*
   (a) *C outputs the same bit string z on inputs λ, Λ and y' as on inputs λ, Λ and y.*
   (b) *y' is an E-encoding of an element x' of S for security and system parameters (λ, Λ), computable by running D on inputs λ, Λ and y'.*
   (c) *$x' \not\subseteq x$.*

Definition 24 (Omission-tolerant integrity protection with system parameterization). *Let $\mathcal{E} = (P, S, E, D)$ be an encoding scheme with system parameterization where the elements of S are sets and E is a two-phase encoding algorithm defined by a two-phase split $(P, S, E_{\mathrm{I}}, E_{\mathrm{II}})$. A checksum scheme with system parameterization $\mathcal{C} = (P, C)$ is said to provide* omission-tolerant integrity protection *for $\mathcal{E}$ if for every $x \in S$, every fixed representation X of x, and every efficient adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins Game 4 with parameters $\mathcal{E}$, $\mathcal{C}$, x, X and $\lambda \in \mathbf{Z}_{\geq 1}$ is a negligible function of λ. An (E, C)-checksum is then said to be an* omission-tolerant checksum.

Algorithm 7 (Assign-and-serialize). *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, and $\mathcal{H} = (P, A, B, H)$ a hashing scheme with system parameterization. The* $(\mathcal{F}, \mathcal{H})$-assign-and-serialize algorithm *takes as inputs $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\lambda)$ and an incomplete typed hash tree X without internal-node labels of kind (b, m, d) with $b = B_\lambda$ and performs the following steps:*

1. *Assign random labels to the dangling nodes of X, if any, each assigned label being uniformly distributed over $\{0,1\}^b$, to obtain an incomplete typed hash tree U without internal-node labels of kind (m, d, b).*
2. *Run the (m, d, b)-serialization algorithm on input U to obtain a bit string y.*
3. *Output y.*

Algorithm 8 (Encoding of a set of key-value pairs with system parameterization). *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, and $\mathcal{H} = (P, A, B, H)$ a hashing scheme with system parameterization. The* $(\mathcal{F}, \mathcal{H})$-encoding algorithm for sets of key-value pairs with system parameterization *is the two-phase encoding algorithm defined by the two-phase split $(P, S, E_{\mathrm{I}}, E_{\mathrm{II}})$ where E_{I} is the $\mathcal{F}$-representation algorithm and E_{II} is the $(\mathcal{F}, \mathcal{H})$-assign-and-serialize algorithm.*

Algorithm 9 (Decoding of a set of key-value pairs with system parameterization). *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, and $\mathcal{H} = (P, A, B, H)$ a hashing scheme with system parameterization. The* $(\mathcal{F}, \mathcal{H})$-decoding algorithm for sets of key-value pairs with system parameterization *takes as input $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\Lambda))$ and a bit string y and performs the following steps:*

1. *Run the (m, d, b)-deserialization algorithm on input y with $b = B_\lambda$ and let U be the output if one is produced, or stop without output otherwise. If U is produced, it is an incomplete typed hash tree without internal-node labels of kind (m, d, b).*
2. *Remove the labels of the dangling nodes of U, if any, obtaining an incomlete typed hash tree X without distinguished-node labels of kind (m, d).*
3. *Run the $\mathcal{F}$-derepresentation algorithm on input X and let x be the output if one is produced, or stop without output otherwise.*
4. *Output x.*

Algorithm 10 (Checksum of a set of key-value pairs with system parameterization). *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, and $\mathcal{H} = (P, A, B, H)$ a hashing scheme with system parameterization. The* $(\mathcal{F}, \mathcal{H})$-checksum algorithm for sets of key-value pairs with system parameterization *takes as input $\lambda \in \mathbf{Z}_{\geq 1}$, $\Lambda \in \mathrm{Supp}(P(\Lambda))$ and a bit string y and performs the following steps:*

1. *Run the (m, d, b)-deserialization algorithm on input y, where $b = B_\lambda$, and let Y be the output if one is produced, or stop without output otherwise.*

2. *Run the (m, d, a, b, h)-internal node label computation algorithm on input Y, where $a = A_\lambda$, $b = B_\lambda$ and $h = \mathcal{H}_{\lambda,\Lambda}$, and let Z be the ouput if one is produced, or stop without output otherwise.*
3. *Output the root label z of Z.*

Fact 9. *The* $(\mathcal{F}, \mathcal{H})$-encoding, decoding and checksum algorithms for sets of key-value pairs with system parameterization *are efficient algorithms.*

Theorem 3. *Let $\mathcal{F} = (m, d, K, V, \mathrm{KE}, \mathrm{KD}, \mathrm{VE}, \mathrm{VD})$ be a representation framework for sets of key-value pairs, and $\mathcal{H} = (P, A, B, H)$ a hashing scheme with system parameterization that is collision and preimage resistant. Let $\mathcal{E} = (P, S, E, D)$ be the encoding scheme with system parameterization where $S = \mathrm{Sets}_{K,V}$, and E and D are the $(\mathcal{F}, \mathcal{H})$-encoding and decoding algorithms with system parameterization. Let $\mathcal{C} = (P, C)$ be the checksum scheme with system parameterization where C is the $(\mathcal{F}, \mathcal{H})$-checksum algorithm for sets of key-value pairs with system parameterization. Then $\mathcal{C}$ provides omission-tolerant integrity protection for $\mathcal{E}$.*

Proof. Let $\mathcal{F}$, $\mathcal{H}$, $\mathcal{E}$ and $\mathcal{C}$ be as stated and let E_I and E_II be the phases of the two-phase encoding algorithm E. Reasoning by contradiction, assume that $\mathcal{C}$ does not provide omission-tolerant integrity protection for $\mathcal{E}$. This means that there exists an element x of S, a fixed representation X of x output by E_I, and an efficient adversary $\mathcal{A}$, such that the function p that maps $\lambda \in \mathbf{Z}_{\geq 1}$ to the probability that $\mathcal{A}$ wins Game 4 with parameters $\mathcal{E}$, $\mathcal{C}$, x, X and λ is not negligible.

Consider an execution of Game 4 with the stated parameters where $\mathcal{A}$ wins the game. X is an incomplete typed hash tree without distinguished-node labels of kind (m, d), where the type-label pairs of leaf nodes with undistinguished types are the key-value pairs of x.

The challenger runs P on input λ, and P outputs Λ. Let $a = A_\lambda$, $b = B_\lambda$, and $h = \mathcal{H}_{\lambda,\Lambda}$.

Then the challenger runs E_II, which is the $(\mathcal{F}, \mathcal{H})$-assign-and-serialize algorithm, on inputs λ, Λ and X. Let $N_1, \ldots N_j$, $j \geq 0$ be the dangling nodes of X, if any. For each i, $1 \leq i \leq j$, E_II chooses a random element of $\{0,1\}^b$ with uniform distribution and assigns it to the dangling node N_i. This results in an incomplete typed hash tree U without internal-node labels. Then E_II runs the (m, d, b)-serialization algorithm on input U and outputs the bit string y that it produces.

Then the challenger runs C, which is the $(\mathcal{F}, \mathcal{H})$-checksum algorithm for sets of key-value pairs, on inputs λ, Λ and y. C runs the (m, d, b)-deserialization algorithm, which outputs an incomplete typed hash tree Y without internal-node labels of kind (m, d, b). Then C runs the (m, d, a, b, h)-label computation algorithm on input Y, which adds internal-node labels to Y and outputs a complete typed hash tree Z. Then C outputs the root label z of Z.

Then the challenger runs the adversary on inputs λ, Λ and the adversary wins the game by outputting a bit string y' that satisfies the winning conditions.

One of the winning conditions is that D outputs an element x' of S on inputs λ, Λ and y'. D computes x' in three stages. It runs the (m, d, b)-deserialization al-

gorithm on input y' to obtain an incomplete typed hash tree U' without internal-node labels of kind (m, d, b). Then it removes the dangling-node labels from U' to obtain an incomplete typed hash tree X' without distinguished-node labels of kind (m, d). Then it runs the $\mathcal{F}$-derepresentation algorithm on input X' to obtain x'.

Another winning conditions is that C outputs z on inputs λ, Λ and y'. To do so, C runs the (m, d, b)-deserialization algorithm and obtains an incomplete typed hash tree Y' without internal-node labels of kind (m, d, b). Then C runs the (m, d, a, b, h)-label computation algorithm on input Y', obtaining a complete typed hash tree Z' whose root label is z.

Z and Z' are typed hash trees of same kind (m, d, a, b, h) that have the same root label z. Therefore, by Theorem 1, either (i) Z' is isomorphic to a pruned derivative of Z, or (ii) the label of an internal node of Z' is equal to the label of a dangling node of Z, or (iii) an internal node of Z' has the same label as an internal node of Z but a different prelabel.

But (i) can be ruled out by the following argument:

1. U and Y are isomorphic as incomplete typed hash trees without internal-node labels of kind (m, d, b) by Fact 6, and so are U' and Y' by Fact 7.
2. Since U is derived from X by assigning labels to the internal nodes, X is derived from U by removing those labels. Let V be derived from Y and V' from Y' by removing the labels of the internal nodes. Then X, V, X' and V' are incomplete typed hash trees without distinguished node labels derived respectively from U, Y, U' and Y' by removing the labels of the internal nodes. Therefore, by Fact 1, since (U, Y) and (U', Y') are pairs of isomorphic incomplete typed hash trees without internal-node labels of kind (m, d, b), (X, V) and (X', V') are isomorphic incomplete typed hash trees without distinguished-node labels of kind (m, d).
3. Since X and V are isomorphic, from the fact that the $\mathcal{F}$-representation algorithm outputs X on input x it follows by Fact 4 that the $\mathcal{F}$-derepresentation algorithm outputs x on input V.
4. Since X' and V' are isomorphic, from the fact that the $\mathcal{F}$-derepresentation algorithm outputs x' on input X' it follows by Fact 3 that it also outputs x' on input V'.
5. Now suppose that (i) holds, i.e. that Z' is isomorphic to a pruned derivative of Z. Since Z is output by the m, d, a, b, h-label computation algorithm on input Y, Y is derived from Z by removing the internal-node labels, and therefore V is derived from Z by removing the distinguished-node labels. Similarly V' is derived from Z' by removing the distinguished-node labels. Since x and x' are output by the $\mathcal{F}$-derepresentation algorithm on inputs V and V' respectively, it follows by Fact 5 that x' is a subset of x. But this is ruled out by the contradiction hypothesis.

Hence either an internal node of Z' has the same label as an internal node of Z but a different prelabel, or the label of an internal node of Z' is equal to the label of one of the dangling nodes of Z, which are also the dangling nodes of U and the dangling nodes N_i, $1 \leq i \leq j$ of X.

Let $\mathbf{E}$ be the event that $\mathcal{A}$ wins Game 4 with parameters $\mathcal{E}$, $\mathcal{C}$, x, X and λ, whose probability is $p(\lambda)$. Let p'_0 be the function that maps $\lambda \in \mathbf{Z}_{\geq 1}$ to the probability that $\mathbf{E}$ occurs and furthermore an internal node of Z' has the same label as an internal node of Z but a different prelabel. Similarlhy, for $1 \leq i \leq j$, let $p'_i(\lambda)$ be the probability that $\mathbf{E}$ occurs, and furthermore the label of an internal node of Z' is equal to the label of N_i. Since one or more of the further eventualities must occur when event $\mathbf{E}$ occurs, we have

$$p(\lambda) \leq \sum_{i=0}^{j} p'_i(\lambda) \tag{1}$$

The fact that the function p is not negligible means that there exists $c \in \mathbf{Z}_{\geq 1}$ such that, for all $\lambda \in \mathbf{Z}_{\geq 1}$ there exists $\lambda' > \lambda$ such that $p(\lambda') \geq \frac{1}{\lambda'^c}$; or that, for such a c, the set

$$L = \{\lambda \in \mathbf{Z}_{\geq 1} | p(\lambda) \geq \frac{1}{\lambda^c}\}$$

is infinite. Define

$$L_i = \{\lambda \in \mathbf{Z}_{\geq 1} | p'_i(\lambda) \geq \frac{1}{(j+1)\lambda^c}\}$$

for $0 \leq i \leq j$. From (1) it follows that, since L is infinite, at least one of the L_i is infinite. And if L_i is infinite, so is

$$L'_i = \{\lambda \in \mathbf{Z}_{\geq 1} | p'_i(\lambda) \geq \frac{1}{\lambda^{c'}}\}$$

for $c' = c + 1$, which implies that p'_i is not negligible.

If p'_0 is non-negligible, consider an execution of Game 1, used in the definition of collision resistance, with parameters (P, A, B, H) and λ, played by a collision resistance challenger $\mathcal{C}_{\mathrm{CR}}$ and a collision resistance adversary $\mathcal{A}_{\mathrm{CR}}$. $\mathcal{C}_{\mathrm{CR}}$ runs P on λ, P outputs Λ, and $\mathcal{C}_{\mathrm{CR}}$ invokes $\mathcal{A}_{\mathrm{CR}}$ on inputs λ and Λ. $\mathcal{A}_{\mathrm{CR}}$ executes Game 4 with parameters $\mathcal{E}$, $\mathcal{C}$, x, X and λ, invoking the challenger algorithm and specifying the adversary algorithm to be run by providing it an input to the challenger.

The adversary of Game 4 that $\mathcal{A}_{\mathrm{CR}}$ specifies is the particular adversary $\mathcal{A}$ that has a probability $p(\lambda)$ of winning the game, modified as needed to ensure that it outputs Z and Z' if it wins the game. $\mathcal{A}_{\mathrm{CR}}$ also modifies the challenger of Game 4 so that it takes Λ as an input instead of obtaining it by running P on input λ; then it passes the Λ of Game 1 as input to the challenger of Game 4. These modifications do not change the probability $p'_0(\lambda)$ that $\mathcal{A}$ wins the game and an internal node of Z' has the same label as an internal node of Z but a different prelabel.

If $\mathcal{A}$ wins the game, which happens with probability $p(\lambda)$, the challenger of Game 4 relays Z and Z' to $\mathcal{A}_{\mathrm{CR}}$, which looks for nodes N_0 of Z and N'_0 of Z' that have the same label l but different prelabels u and u'. If it finds such nodes, which happens with probability $p'_0(\lambda)$, it outputs (u, u') as a collision of h, thus winning Game 1.

Furthermore, $\mathcal{A}_{\mathrm{CR}}$ is efficient according to Definition 14, for the following reasons:

1. The algorithms run by the unmodified challenger of Game 4, the unmodified challenger itself, and the unmodified adversary $\mathcal{A}$ are efficient.
2. Z and Z' are of size polynomial in λ because they are constructed by efficient algorithms.
3. The challenger of Game 4 and the adversary $\mathcal{A}$ remain efficient when modified to output Z and Z' respectively because Z and Z' are of size polynomial in λ.
4. Finding N_0 and N_0' takes polynomial time because Z and Z' are of polynomial size.
5. Outputting u and u' takes polynomial time because u and u' are of polynomial size.

Thus $\mathcal{A}_{\mathrm{CR}}$ is an efficient adversary that wins Game 1 with non-negligible probability $p_0'(\lambda)$, contradicting the assumption that (P, A, B, H) is collision-resistant.

If p_i' is non-negligible for some i in the range $1 \leq i \leq j$, consider an execution of Game 2, used in the definition of preimage resistance, with parameters (P, A, B, H) and λ, played by a preimage resistance challenger $\mathcal{C}_{\mathrm{PR}}$ and a collision resistance adversary $\mathcal{A}_{\mathrm{PR}}$. $\mathcal{C}_{\mathrm{PR}}$ runs P on λ and P outputs Λ. Then $\mathcal{C}_{\mathrm{PR}}$ chooses an element v of $\{0,1\}^b$ at random with uniform distribution, and invokes $\mathcal{A}_{\mathrm{PR}}$ on inputs λ, Λ and v. $\mathcal{A}_{\mathrm{PR}}$ executes Game 4 with parameters $\mathcal{E}$, $\mathcal{C}$, x, X and λ, invoking the challenger algorithm and specifying the adversary algorithm to be run by providing it an input to the challenger.

As above, $\mathcal{A}_{\mathrm{PR}}$ specifies $\mathcal{A}$ as the adversary of Game 4, modifying it so that it outputs Z and Z' if it wins the game, modifying the challenger of Game 4 so that it takes Λ as an input instead of obtaining it by running P on input λ, and passing the Λ of Game 1 as input to the challenger of Game 4. Furthermore $\mathcal{A}_{\mathrm{PR}}$ modifies the parameter $\mathcal{E}$ of Game 4 so that its second phase E_{II} assigns v as the label of N_i, instead of generating a fresh random label. Given that v is an element of $\{0,1\}^b$ randomly generated with uniform distribution, none of these modifications change the probability $p_i'(\lambda)$ that $\mathcal{A}$ wins the game and the label of an internal node of Z' is equal to the label of N_i.

The challenger of Game 4 relays Z and Z' to $\mathcal{A}_{\mathrm{PR}}$, which traverses Z' looking for an internal node whose label coincides with the label v of the i-th dangling node of Z. If it can find such a node, which happens with probability $p_i'(\lambda)$, it wins Game 2 by outputting its prelabel. Like $\mathcal{A}_{\mathrm{CR}}$ above, $\mathcal{A}_{\mathrm{PR}}$ is an efficient algorithm because the data structures that it manipulates are of size polynomial in λ and the algorithms that it invokes are themselves efficient. Thus $\mathcal{A}_{\mathrm{PR}}$ is an efficient adversary that wins Game 2 with non-negligible probability $p_i'(\lambda)$, contradicting the assumption that (P, A, B, H) is preimage resistant.

Having reached a contradiction in both cases, we can conclude that $\mathcal{E}$ does provide omission-tolerant integrity protection. □

6 Conclusion

We have introduced the concepts of an omission-tolerant checksum and a typed hash tree, and described a method for using a typed hash tree to represent a set of key-value pairs or, as a special case, a set of unstructured elements, serializing the tree to obtain a bit-string encoding of the set, and using the root label of the tree as an omission-tolerant cryptographic checksum of the encoding. Using Boneh & Shoup's system parameterization and attack game methodology, we have proved that, given a set of key-value pairs, an efficient adversary has a negligible probability of producing a set of key-value pairs other than a subset of the given one that has the same root-label checksum.

An omission-tolerant checksum has many possible applications. A particularly important one is a method for implementing selective disclosure of data items bound to a public key by a public key certificate such as, e.g., a TLS client certificate. We leave the details of this application for future work.

References

1. Alfred J. Menezes and Paul C. Van Oorschot and Scott A. Vanstone and R. L. Rivest: Handbook of Applied Cryptography (1997), Chapters available online at http://cacr.uwaterloo.ca/hac/
2. Boneh, D., Shoup, V.: A Graduate Course in Applied Cryptography, Version 0.4. September 2017. https://crypto.stanford.edu/~dabo/cryptobook/
3. Camenisch, J., Lysyanskaya, A.: Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In: Theory and Application of Cryptographic Techniques, EUROCRYPT (2001)
4. Corella, F.: Method and apparatus for providing field confidentiality in digital certificates (October 2004), US Patent 6,802,002
5. Corporation, M.: U-Prove Home Page, http://www.microsoft.com/u-prove
6. D. Richard Kuhn: A Data Structure for Integrity Protection with Erasure Capability, NIST White Paper (DRAFT), Computer Security Division, Information Technology Laboratory. May 31, 2018. https://csrc.nist.gov/publications/detail/white-paper/2018/05/31/data-structure-for-integrity-protection-with-erasure-capability/draft
7. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. Chapman & Hall/CRC, 2nd edn. (2014)
8. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency, IETF RFC 6962. https://tools.ietf.org/html/rfc6962
9. Lewison, K., Corella, F.: Rich Credentials for Remote Identity Proofing, Pomcor technical report, revised July 10, 2017; first version October 15, 2016. https://pomcor.com/techreports/RichCredentials.pdf
10. Mastercard, VISA: SET Secure Electronic Transaction Specification, Version 1.0, May 1997. http://www.maithean.com/docs/set_bk1.pdf, http://www.maithean.com/docs/set_bk2.pdf, http://www.maithean.com/docs/set_bk3.pdf
11. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology — CRYPTO 1987. Springer, Lecture Notes in Computer Science, vol 293

12. NIST: Digital Signature Standard (DSS) (July 2013), FIPS PUB 186-4, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
13. Stallings, W.: The SET Standard & E-Commerce, November 1, 2000. http://www.drdobbs.com/the-set-standard-e-commerce/184404309#
14. Wikipedia: Algorithm characterizations, https://en.wikipedia.org/wiki/Algorithm_characterizations. Accessed October 30, 2018