4

Security Approaches for Information-Centric Networking

Walter Wong and Maurício Ferreira Magalhães
University of Campinas
Brazil

1. Introduction

The increasing demand for highly scalable infrastructure for efficient content distribution has stimulated the research on new architectures and communication paradigms, where the focus is on the efficient content delivery without explicit indication of the resource location. One of these paradigms is known as information-centric networking (ICN) and its main focus is on data retrieval regardless of the source at the network level. This scenario usually happens when content providers (e.g. Warner Bros, BBC News) produce information (movies, audios, news in a Web page, etc.) and hire delivery systems such as Akamai[1] to deliver their content to the customers. In this model, there is a decoupling between content generation from the server storing the content itself (the actual machine serving the content for clients). Originally, servers used to generate and deliver data to the clients, however, nowadays data may be generated in specialized locations and placed in strategic servers in the network to speed up the content delivery to content consumers.

From the security perspective, the decoupling of data production and hosting opens new challenges for content authentication. The first issue regards the trust establishment for content authentication and a second one is the time decoupling between data consumption and production. Previously, data was generated in servers and the authentication of the hosting server resulted into an *implicit* data authentication because the content producer is the same as the content server. Nowadays, a common scenario is the separation between content generation and delivery, breaking the previous trust relationship established between the serving host and the content. Servers are deployed by content delivery companies to deliver data according to a contract, thus, there might not be a correlation between serving host and the data itself. The second issue regards the time decoupling between data consumption and production, which is a direct consequence of content production and hosting separation. Content providers produce content (e.g. news feeds) that may not be synchronously consumed, i.e., BBC News web-site produces news every 5 minutes, but clients access the data after some period of time. As a consequence, content providers and consumers are *decoupled in time* and *synchronization*, and there might not be any interaction between clients and servers to ensure the content authenticity[2]. Some threats such as fake and unauthorized content publication or content data blocks corruption may appear, requiring a new security model focused on the content itself rather than securing the connection.

[1] http://www.akamai.com
[2] Sometimes the original content provider is not online to provide authentication data.

In this paper, we present two hash tree techniques to provide content authentication based on the content rather than the communication channel to provide content authentication in information-centric networks. The authentication model uses *skewed hash trees* (SHT) and *composite hash trees* (CHT) to provide amortized content authentication and integrity for a set of data blocks with one single digital signature. Moreover, the security model is independent of the underlying transport protocol, allowing it to verify the content with the original content owner, regardless of the storage or mirror where it was retrieved. The SHT mechanism allows for secure content caching in the network, enabling data verification by intermediate devices at low processing costs. The CHT mechanism allows for parallel authentication over HTTP, enabling parallel content download in the Internet. As a proof-of-concept, we implemented a prototype with the SHT and CHT libraries and evaluated in these two scenarios, outlining the main experimental results.

The organization of this paper is as follows. Section 2 presents the background information about Merkle Trees. Section 3 presents the SHT and CHT techniques for content authentication in information-centric networks. Section 4 describes the SHT and CHT implementations in the secure caching and parallel authentication scenarios. Finally, Section 5 summarizes the paper.

2. Background

The Merkle Tree (MT) (Merkle, 1989) is a *balanced binary tree* structure containing summary information of a large piece of data or a message set. The data structure was originally proposed in the late 70's as an alternative to provide compact representation of public keys and the main idea is to apply a cryptographic hash over a set of messages and use these hash values as input for a balanced tree. Each parent node contains the hash of the concatenation of the hash values stored in the children's nodes and it goes recursively until reaching the top of the tree. This value is known as *root hash* and it represents the *fingerprint* over a set of messages. Each data block has a list of hash values called *authentication path* (AP) that allows users to verify the integrity by computing the path from the leaves towards the *root hash*, and comparing it with the securely retrieved *root hash* value.

Some applications use MTs to provide efficient content authentication in different scenarios, such as HTTP (Bayardo & Sorensen, 2005) and P2P networks (Tamassia & Triandopoulos, 2007). These applications create a MT over a set of data blocks and append an AP on each block to allow data verification in the receiver side. However, the construction of a MT requires a balanced binary tree, demanding a number of data blocks that are multiple of power of two, otherwise the original MT algorithms will not work due to the unbalance in the tree. There are two simple solutions to tackle this issue: (1) fit the number of blocks to be a power of two; (2) pad with zeros. The first one is restrictive because it intervenes with the application's requirements, e.g., maximum transmission unit, and the second one results in additional computation overhead.

Fig. 1 illustrates the problem with the naive zero padding. The worst case happens when a user has a number of blocks that fits in a balanced tree plus one, requiring a binary tree that is the double of the size. As the height of a MT grows, the number of required zero leaves increases proportionally, resulting in $2^{(H-1)} - 1$ zero leaves, when the number of blocks is equal to $N/2 + 1$, and it requires a tree with height $H + 1$ to hold the hash information of all data blocks. Hence, the number of hash function calls in the zero padding scheme is the same as in the balanced tree since the zero leaves are computed as a regular leaf. Thus, the total number of hash function calls is the sum of all hash functions calls over the N data blocks,

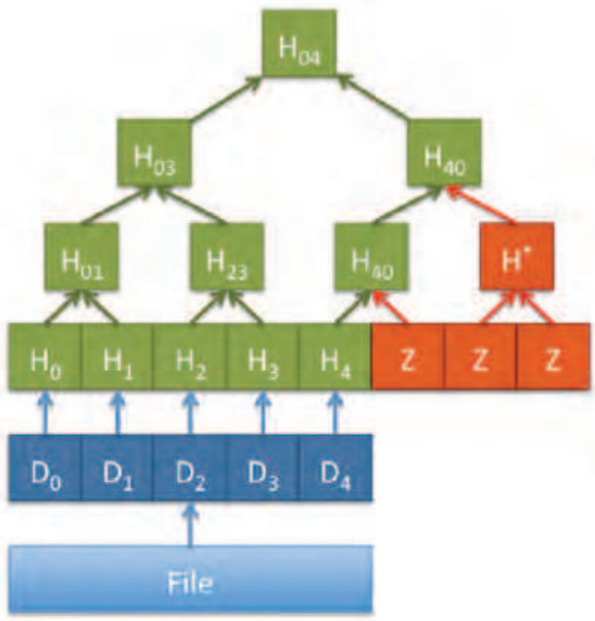


Fig. 1. Naive zero-padding in the Merkle Tree.

plus in the intermediate and top nodes. Consequently, we have:

$$\sum_{i=0}^{H} 2^i = 2^{H+1} - 1 = 2N - 1 \tag{1}$$

Therefore, a MT with N leaves (where $N = 2^H$) requires $2N - 1$ hash function calls to generate the *root hash*, regardless of the number of empty leaves in the tree. In order to tackle this limitation, we propose two mechanisms based on hash trees for information-centric data authentication, called *skewed hash tree* and *composite hash tree* that will be presented in the next section.

3. Security design

In this section we present two hash tree techniques, the skewed hash tree and the composite hash tree, that provide content authentication based solely on the content. These two techniques transfer the trust placed on the root hash to the data blocks through strong cryptographic hash functions, allowing for efficient and trusted content authentication. We start describing the skewed hash tree and then we describe the composite hash tree.

3.1 Definitions

In order to better describe the hash tree data structure and the verification procedures associated to it, we start with some definitions used through the text to ease the comprehension of the proposed mechanism.

- **Block.** A block or data block is a fragment of a larger file and is considered as the *smallest* unity of data used as input of the skewed hash tree algorithms.

- **Leaf.** A leaf is the bottom node of a binary tree. It contains the cryptographic hash value of a data block.
- **Balanced leaf.** A balanced leaf is a leaf of a balanced binary tree. Even though they are leaves, they may have some skewed leaves appended, but they are called balanced leaves to identify the lowest level of a balanced tree. These leaves can be handled using regular Merkle tree algorithms.
- **Skewed leaf.** A skewed leaf is the leaf that is appended under a balanced leaf. It needs special handling in order to generate a coherent root hash value that can be used in the verification process.
- **Height.** The height h is the total height of the entire skewed hash tree, which is the height of a balanced tree if there is no skewed leaf, or the balanced tree plus one if there are skewed leaves.
- **Hash Tree (HT).** A binary hash tree is a complete binary tree with height h and 2^h leaves. Each leaf stores a cryptographic hash value of over a data block and each internal node stores the hash of the concatenation of its children's node;
- **Root Hash (RH).** The *Root Hash* is the hash value in the top of an intermediate hash tree, representing the signature over a set of data blocks. The RH algorithmically binds together all data blocks, and any change in any data block will result in a different signature;
- **Composite Root Hash (CH).** The *Composite Root Hash* is the hash value in the top of a composite hash tree used to authenticate the incoming *Authentication Data Blocks*. The CH can be digitally signed to provide both content authentication and integrity regardless of the number of data blocks;
- **Authentication Data Block (AD).** The *Authentication Data Block* contains intermediate RH values of the hash trees used in the composition. It is used to authenticate the smaller trees and data blocks as they arrive in the receiver side;
- **Authentication Path (AP).** The *Authentication Path* is the list of hash values needed to authenticate a specific data block. The AP hash value in a given height h is the sibling hash in the hash tree towards the root hash. The main difference between AP and AD is that the first one is used to authenticate one data block and the second one is used to authenticate the RH of intermediate hash trees.

3.2 Skewed hash tree

In this section, we present the *skewed hash tree* (SHT), a variant of the original Merkle Tree that supports random size file verification with the minimum overhead associated with each data block. The SHT introduces an easy yet powerful algorithms to leverage the file partitioning procedure, allowing applications to freely divide the data blocks according to their requirements. The proposed mechanism is useful for applications that require: (i) low verification overhead; (ii) content-based or connection-less verification; (iii) random order verification; (iv) random size file authentication.

The SHT extends the original Merkle Tree algorithms to allow data authentication in cases where the number of chunks (data fragments) is not multiple of power of two. In order to achieve this requirement, we separate the hash tree into two parts: one balanced tree and a second one with the *skewed leaves*. A skewed leaf is a leaf that is going to be appended under a balanced leaf and it has a special handling in the algorithm. The balanced tree is created over a partitioned content and later the skewed leaves are added under the balanced tree, creating

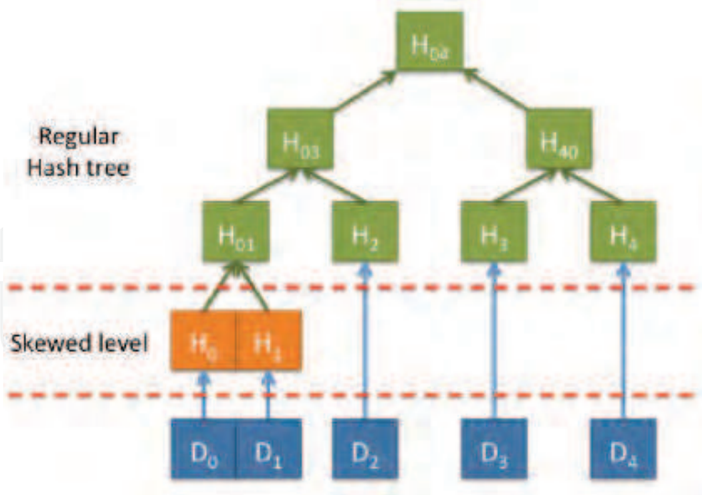


Fig. 2. Skewed Hash Tree proposal.

one extra height in the skewed hash tree. The advantage of splitting the tree in balanced tree and skewed leaves is to maintain the compatibility with the original Merkle tree algorithms for the balanced tree while handling correctly the skewed leaves.

Fig. 2 illustrates an example of skewed hash tree, where the balanced tree comprehends the leaves with hash values H_{01}, H_2, H_3 and H_4 and the skewed leaves contain the hash values H_0 and H_1. The SHT construction starts with the computation of the smallest tree height that can hold all data blocks minus one[3], which in this case is $h = 2$ and results in four balanced leaves. Next, the mechanism computes the number of balanced leaves that will receive the skewed leaves in order to hold all data blocks. Finally, it computes the root hash over the data set.

In order to differentiate the skewed leaves from the balanced ones, the skewed leaves are inserted at the height $h = -1$, indicating that they are appended leaves and they should be handled as a special case when using regular Merkle tree algorithms.

The algorithm to calculate the root hash starts in the first leaf of the balanced tree, in this case, H_{01}. The first step of the algorithm is to check whether it has skewed leaves appended in that leaf or not. In the example, the leaf H_{01} has appended the skewed leaves H_0 and H_1, thus the algorithm must compute first these two leaves and later the algorithm returns again to the balanced tree. The balanced tree algorithm now goes to the second leaf H_2. It checks whether there are appended leaves or not and treats the skewed leaves. From leaf H_2 onward, there is no more skewed leaves, thus, the balanced Merkle tree algorithms can work normally.

3.2.1 SHT construction

The skewed hash tree computation is divided into three phases: *root hash generation*, *AP generation* and *data blocks verification*. The first phase generates the public signature of a target file, the second phase generates the AP for each data block and the third phase authenticates each data block. In the following algorithms, we use the *stack* data structure to ease the algorithm description and understanding. The decision to use a stack is because it can hold

[3] The motivation to reduce the tree height in one is to avoid empty leaves, for example, if we choose a tree of height $h = 3$ for this example, we would have 5 data blocks and three empty blocks.

the last two values in the top of the stack, easing the comparison process of the last two values. Also, we consider that the stack has the *pop* and *push(element)* primitives, where pop removes the top element of the stack and push adds an element in the top of the stack.

The number of skewed leaves in a skewed hash tree with height h is the number of current leaves in the hash tree minus the number of data blocks of a balanced hash tree with height $h-1$, multiplied by two[4]. Therefore:

$$num\_skewed\_leaves = 2 * (N - 2^{balanced\_tree\_height}) \quad (2)$$

where the *balanced_tree_height* is height of the balanced tree. The number of balanced leaves with appended skewed leaves is:

$$num\_balanced\_leaves = N - 2^{balanced\_tree\_height} \quad (3)$$

Fig. 3 presents a comparison between the number of hash function calls in the MT and SHT.

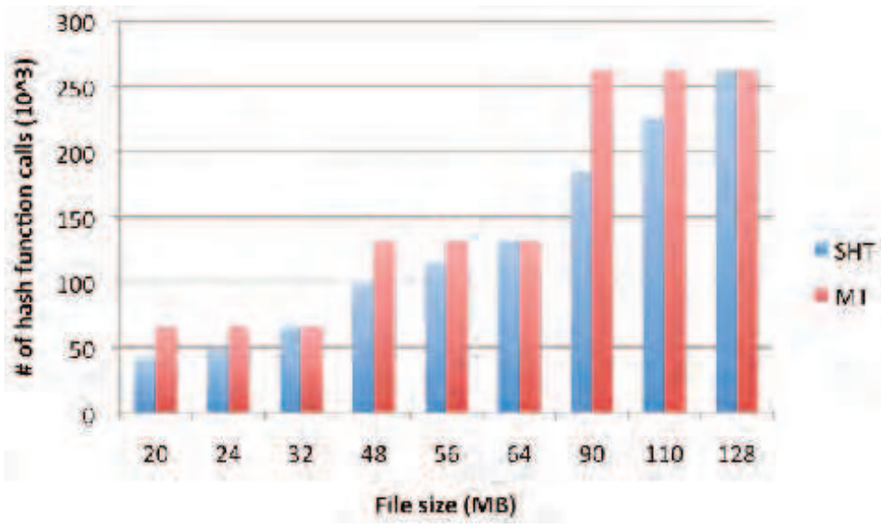


Fig. 3. Comparison between the number of hash function calls in Merkle trees and Skewed hash trees.

Note that MT has a constant overhead per tree height while SHT adapts to the current number of data blocks. The main reason why MT has a constant processing overhead is due to the computation of the empty leaves in order to reach to the *root hash*. On the other hand, SHT just computes the leaves with data blocks, skipping the empty ones. Thus, the worst case for SHT is to have the same computational overhead as regular MT.

3.2.2 SHT algorithms

There are three algorithms associated to SHT: *skewed_treehash*, *skewed_ap* and *skewed_verify*. The skewed_treehash computes the root hash of a skewed hash tree; the skewed_ap computes the authentication path for each data block; and skewed_verify checks whether a data block is consistent with a given root hash or not. We are going to describe each one in detail.

[4] The next height of a binary tree has two times the number of leaves of the previous height.

Algorithm 1 SHT treehash algorithm

```
Input: File, max_height, num_skewed_leaves
Output: Root hash
skewed_count = 0; height = 0;
while height <= max_height do
  if top 2 values have equal height then
    h_R ← pop()
    h_L ← pop()
    height = h_L.height
    h_x ← hash(h_L || h_R)
    stack.push(h_x, height + 1)
  else
    if read_data NOT EOF then
      data = read_data(file)
      if skewed_count < num_skewed_leaves then
        stack.push(hash(data), height=-1)
        skewed_count = skewed_count + 1
      else
        stack.push(hash(data), height=0)
      end if
    end if
  end if
  height ← stack[0].height
end while
Return stack[0]
```

Alg. 1 describes the root hash generation procedure in a skewed hash tree, which is based on the original *treehash* Merkle tree algorithm.

The algorithm receives as input a file, the block size, the maximum height of the tree (which is calculated dividing the file size by data block size and verifying the smallest height of a balanced tree that can hold that number of leaves) and the number of skewed leaves computed with Eq. 2.

The second phase corresponds to the AP generation for each data block and is divided into two steps: (1) initial stack filling and (2) AP generation. The first step uses the skewed treehash algorithm to store all hash values of the leftmost and rightmost leaves ($h_L \leftarrow pop()$ and $h_R \leftarrow pop()$ in Alg. 1) in the S_h and AP_h stacks respectively. The S_h stack contains the hash value to be used in the next AP generation and the AP_h stack contains the AP value at the height h and it contains the authentication path of the first block. These stacks are used as temporary variables to store the previous hash computed hash values to be used in the next AP computation.

The second step uses the pre-filled S_h and AP_h stacks to output each AP in sequence with one tree traversal. Alg. 2 describes the skewed hash tree traversal algorithm. The algorithm receives as input the file, the number of balanced leaves with appended skewed leaves and the height of the balanced tree and outputs the AP for each data block in sequence.

The third phase comprehends the data block verification procedure, described in Alg. 3, where the receiver gets the data block with its corresponding AP and the block index. We assume

Algorithm 2 SHT authentication path generation

Input: File, num_balanced_leaves, H
Output: Data blocks with *Authentication Paths*
leaf = 0, skewed_count = 0
if leaf < $2^H - 1$ **then**
  if skewed_count < num_balanced_leaves **then**
    $data_0$ = read_block(); $data_1$ = read_block()
    Output $data_0$, hash($data_1$), AP; Output $data_1$, hash($data_0$), AP
    skewed_count = skewed_count + 1
  else
    data = read_block()
    Output data, AP
  end if
  for $h = 0$ to H **do**
    if (leaf + 1) mod 2^h == 0 **then**
      $AP_h = Stack_h$
      startnode = (leaf + 1 + 2^h) XOR 2^h
      $Stack_h$ = skewed_tree_hash(startnode, h)
    end if
  end for
  leaf = leaf + 1
end if

Algorithm 3 SHT verification

Input: Root Hash, block index, data block, AP
Output: True or False
pos = index
digest = hash(data_block)
for each AP_i value in AP **do**
  if (pos % 2 == 0) **then**
    digest = hash(digest || AP_i)
  else
    digest = hash(AP_i || digest)
    $pos = \lfloor pos/2 \rfloor$
  end if
end for
if (digest == *Root Hash*) **then**
  Return **True**
else
  Return **False**
end if

the root hash was previously transferred to the receiver in a secure way, for example, using the security plane model. The algorithm starts reading the data block's AP and appends each hash value in the correct side to reach the root hash.

3.3 Composite hash tree

The Composite Hash Tree (CHT)(Wong et al., 2010a;b) is a data structure created over a set of data blocks belonging to a complete file. The main idea is to create a set of small binary hash trees of fixed height over a set of data blocks and recursively construct other binary hash tree over the previous hash trees in the first level, until reaching one single hash tree in the top level. The motivation for this approach is the high overhead present in the Merkle tree and also skewed hash tree, because the latter one is mainly based on the original Merkle tree algorithms. In these approaches, each data block has a list of cryptographic hash values (authentication path) that is the same length of the hash tree. Therefore, each authentication path has $log_2 N$ values and the sum of all authentication overhead grows $N * log_2 N$, where N is the number of blocks. Thus, for large files, this overhead might be considerable, especially in scenarios using low processing devices such as mobile phones.

In order to attack the authentication overhead problem, we propose CHT as an alternative to both Merkle and skewed hash trees for authentication purposes with low overhead. The proposed mechanism also provides signature amortization, allowing one piece of content to be authenticated with one digital signature regardless of the number of data blocks, requiring on average $O(N)$ fingerprints to authenticate N data blocks that are components of the original content for small composing Merkle tree with height h. A CHT(α, h) is a composite hash tree using smaller Merkle trees of height h (MT(h)) whose root hash values are aggregated in blocks of α elements. Fig. 4 illustrates an example of CHT(1, 2) using internal hash tree value $h = 1$ and intermediate RH aggregation of two blocks ($\alpha = 2$). In this example, a file is divided in eight data blocks (D_0 to D_7) and an intermediate hash tree of height $h = 1$ is constructed using the cryptographic hash of the data blocks as input (H_0 and H_1), resulting in an intermediate *Root Hash* (H_{01}). This intermediate RH is used as the verification information for the data blocks D_0 and D_1, which later on will be aggregated in *Authentication Data Blocks*.

The CHT has two configuration parameters: aggregation index (α) and internal hash tree height (h). The α parameter is used to define the aggregation level of the intermediate RH values in the binary hash tree in α values. The internal hash tree height (h) defines the height of the internal hash trees used in the composition. These two parameters allow for the customization of the tree behavior, for instance, the initial verification ordering and the verification overhead, according to the application requirements. Higher h values provide smaller authentication hierarchy, meaning that data and authentication blocks have low interdependency at the cost of higher authentication overhead per data block. On the other hand, small h values results in low authentication overhead, but longer data block authentication hierarchies (thus, higher dependency between data and authentication blocks).

In this example of Fig. 4, intermediate RH values in the first level (H_{01} and H_{23}) are aggregated together in blocks of two ($\alpha = 2$), resulting in the *Authentication Data Blocks* with hash values $H_{01}||H_{23}$ and $H_{45}||H_{67}$, where $||$ represents the concatenation operation. In the second level, the *Authentication Data Blocks* are considered as input data blocks. Hence, the CHT applies the cryptographic hash over the ADs, resulting in the hash values H_{03} and H_{47} and another intermediate hash tree of height $h = 1$ is constructed over these two data blocks, resulting in the *Composite Root Hash* that will be used in the verification procedure. In the case of larger files, this procedure is applied recursively until reaching the *Composite Root Hash*.

In order to provide data verification, each data chunk carries a list of hash values represented by the AP used to verify with the CH. The AP for each data block is the sibling hash value in the hash tree, for instance, in the example described in Fig. 4, the AP for the D_0 is H_1, since

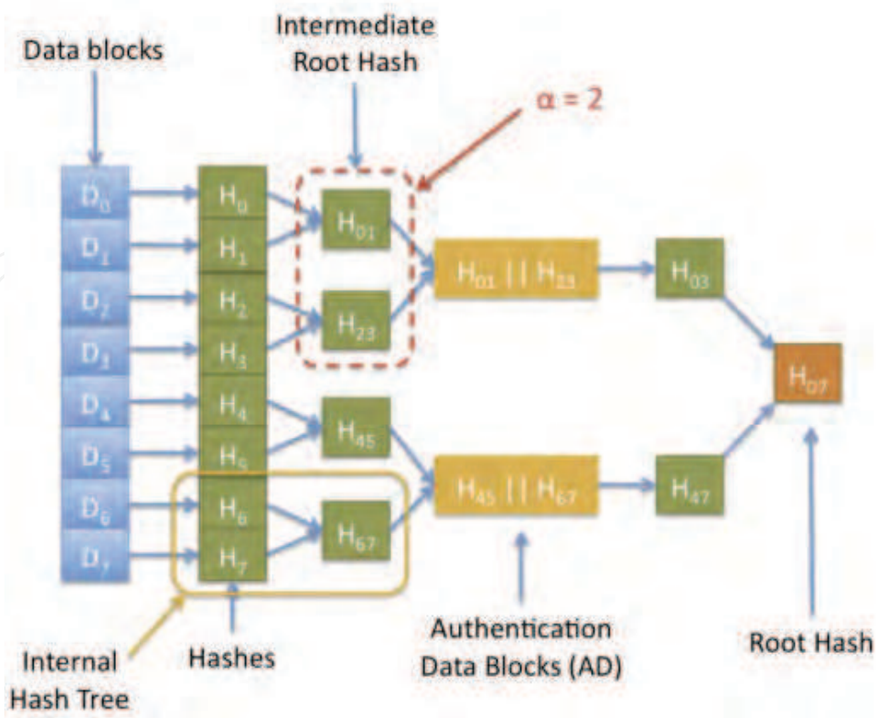


Fig. 4. (a) Composite Hash Tree with internal HT of height $h = 1$ and $\alpha = 2$.

this value is the sibling value of H_0. For larger hash trees, the AP is composed of the sibling hash value at each height towards the RH[5]. Therefore, the overhead per data chunk is defined by the height of the internal hash tree. In this approach, the CHT maintains just one hash value needed to authenticate a target data block, discarding the repeated values of the regular Merkle Tree. On the other hand, this mechanism introduces an authentication hierarchy between data and authentication blocks, requiring that some blocks to be authenticated prior to the data blocks authentication.

The α index reduces the authentication hierarchy needed to authenticate all data blocks in an order of α elements. Thus, the index reduces $\log_\alpha N$ authentication levels, where N is the number of partitioned data blocks.

Fig. 5 illustrates an example of authentication hierarchy using a sliding window for a CHT($\alpha = 2, h = 1$). The figure has two columns, the first one indicates the received data blocks in the receiver side and the second column shows the next blocks window to be downloaded. As authentication blocks arrive, the next blocks to be downloaded *slides* to the next set of data blocks that can be downloaded with the arrival of the new set of *Root Hashes*. For example, after the receiver authenticates the AD_0 containing the hash values $H_{01}||\ H_{23}$, the user can start downloading data blocks D_0, D_1, D_2 and D_3, in any sequence.

The same procedure is taken when the AD with concatenated hash values $H_{45}||H_{67}$ is received in the destination, allowing the download and authentication of data blocks D_4, D_5, D_6, D_7 in any sequence.

[5] Recalling that the AP length is the height of the Merkle Tree, thus, this is the motivation to use really small Merkle trees.

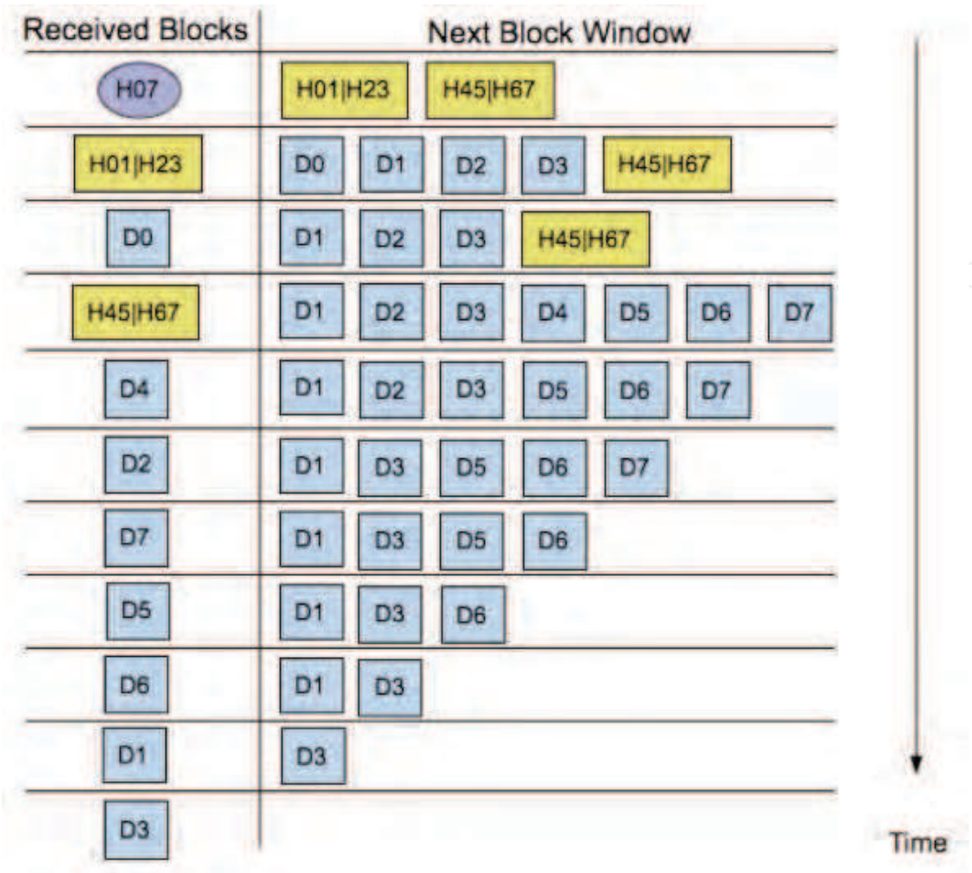


Fig. 5. Authentication Window for CHT.

CHT overhead complexity

The CHT overhead has two components associated, the *Authentication Path* (O_{AP}) overhead of each data and *Authentication Data Block* (O_{AD}) overhead, which are the aggregated *Root Hash* values of the intermediate Merkle Trees. Thus, the total overhead is:

$$O_T = O_{AP} + O_{AD} \tag{4}$$

The O_{AP} is the sum of the product between the number of data blocks on each height by the size of the AP, which is defined by the height of the Merkle Tree used in the CHT. From the CHT construction examples above (Figs. 4), we can notice that the factor $2^h\alpha$ repeats recursively i times to create the CHT over the data blocks. Note that the last MT created over the data blocks does not follow the pattern because they are the data blocks on which the composite hash tree is being created over. These data blocks add 2^h leaves, thus we need to add it separately in the formula to compute the overhead. Therefore, the O_{AP} formula is the product of the i recursions plus 2^h leaves over the data blocks plus the AP length (which is the same as h).

$$O_{AP} = \sum_{i=0}^{H'} (2^h\alpha)^i * 2^h * (AP\ length = h) \tag{5}$$

where H' represents the CHT height minus 1[6]. The H' is the number of data blocks N minus the MT(2^h) over the data blocks that do not repeat. Therefore:

$$H' = \lceil \log_{(2^h \alpha)} (N/2^h) \rceil \tag{6}$$

The O_{AD} is similar to the AP overhead formula and computes the sum of the product of the intermediate *Root Hash* values that are aggregated into α hash values, excluding the MT(2^h) over the data blocks since it starts from the first level. Hence:

$$O_{AD} = \sum_{i=1}^{H'} (2^h \alpha)^i * (AP\ length = h) \tag{7}$$

In order to calculate the overhead complexity with the input, we first calculate the total number of data blocks of a CHT(α, h). From 6, we have that:

$$N = (2^h \alpha)^{H'} * 2^h \tag{8}$$

From 5 and substituting with 8, we have:

$$O_{AP} = \sum_{i=0}^{H'} (2^h \alpha)^i * 2^h * h \approx (2^h \alpha)^{H'} * 2^h * h = N * h \tag{9}$$

Therefore the O_{AP} in the CHT is $N * h$ and grows $O(N)$ when $N >> h$ and h is a constant that does not change with the input size. The maximum value for h in a binary tree is $\log_2 N$, reducing the CHT to a regular Merkle Tree with overhead complexity of $O(N \log_2 N)$.

The *Authentication Data Block* overhead has similar proof to the previous one. Thus, substituting in 7, we have:

$$O_{AD} = \sum_{i=1}^{H'} (2^h \alpha)^i * h \approx (2^h \alpha)^{H'} * h = (N * h)/2^h \tag{10}$$

Therefore, the O_{AD} in the CHT is $N * h/2^h$ and grows $O(N)$ when $N >> h$ and h is a constant parameter that does not change with the input size. The total CHT overhead (O_T) is:

$$O_T = N * h + (N * h)/2^h = O(N) \tag{11}$$

Table 1 compares the overhead of a regular Merkle Tree and a CHT(1, 2):

4. Application scenarios

In this section we present two application scenarios for the SHT and CHT. For the first evaluation scenario, we apply the SHT mechanism in the secure content caching mechanism. The SHT allows for content authentication prior to the caching procedure, preventing the unnecessary caching of bogus content. In the second evaluation scenario, we apply the CHT mechanism in the parallel authentication over HTTP scenario.

[6] It is not considered the *Root Hash* in the height computation (thus, $H' = H - 1$).

# of blocks	Merkle Tree	Composite Hash Tree (1,2)	Overhead Reduction(%)
8	24	12	50.00
32	160	48	70.00
128	896	192	78.57
512	4,608	768	83.34
2,048	22,528	3,072	86.36
8,192	106,496	12,288	88.46
32,768	491,520	49,152	90.00
131,072	2,228,224	196,608	91.18
524,288	9,961,472	786,432	92.10

Table 1. Merkle Tree vs. Composite Hash Tree Overhead Comparison

4.1 Secure caching

The fast growth of the user generated content have put pressure on the Internet infrastructure, requiring higher bandwidth capacity and lower latency to connect content providers and users. However, the infrastructure deployment speed has not followed the bandwidth usage mainly due to the lack of incentives to upgrade the infrastructure that interconnects the ISPs, problem known as the *middle mile problem* (Leighton, 2009). In order to reduce the pressure on the infrastructure and also the inter-ISP traffic, ISPs have deployed Web caches (Rodriguez et al., 2001) and content routers (Wong et al., 2011), to reduce the redundant traffic going through their networks. The placement of the caches close to the consumers improves the overall user experience and also temporarily reduces the pressure on the middle mile.

Despite the fact that content caches can be introduced in the network to improve the network efficiency, there is no explicit mechanism to authenticate the cached content, for example, check whether a piece of content is a *malware* or not. As a consequence, caches can be filled with bogus content, reducing the overall cache hit ratio. According to (Reis et al., 2008), 1.3% of the total Web pages downloaded in the Internet are changed during the transfer from the server to the clients, without any explicit knowledge of the receiver. The lack of external security parameters prevents intermediate devices to authenticate content, mainly because current security protocols are end-to-end.

In this section, we present an authentication scheme based on SHT to provide content authentication in network-level caches. The authentication mechanism based on SHT allows for data verification prior to the caching event, preventing the caching of polluted content on *content routers* (CR) (Wong et al., 2011). The CRs are able to route and cache pieces of content in their internal memory for some amount of time. Therefore, these devices have limited storage and they need to optimize the caching capacity and verification mechanisms must be fast enough to be in line speed. The caching mechanism uses high-level content identifiers, resulting in location-independent identifiers to represent content in the Internet and also content self-certification. Some benefits of the secure caching network include improved traffic efficiency by saving the amount of traffic in the network, opportunistic multi-source content retrieval by redirecting requests to nearby caches and security embedded in the content, allowing for authentication directly with the original provider through a security plane.

4.2 Network caching design

This section presents the in-network caching architecture, outlining the main design goals and discussing project decisions regarding content identification, forwarding and authentication.

4.2.1 Design goals

The in-networking caching architecture aims at the following design goals:

- **Protocol independence.** The in-network caching mechanism must be independent of any specific protocol (Arianfar et al., 2010), for instance, peer-to-peer protocols or HTTP.
- **Multi-source content retrieval.** The forwarding mechanism should support multi-source content retrieval from multiple caches on the path towards the original provider.
- **Cache-based forwarding.** The delivery mechanism forwards data requests towards other in-network caches that may have the content, thus, avoiding any lookup process and incurring into a minimum latency towards the original content provider.
- **Content authenticity.** Clients should be able to verify the content integrity despite retrieving data chunks from multiple sources.
- **Provenance.** Data must be always authenticated with the original source or providers, regardless from which mirror (e.g., network cache, peer) that it was retrieved from.

4.2.2 Content router

The *Content Router* (CR) is a network element that acts as a regular router and also provides content routing mechanisms. The main idea is that CRs inspect a CR header in all in-transit data and store some of them with a certain caching probability. Thus, further requests can be served by the cache data in the CR. In addition to the caching feature, CRs also store *pointers* to pieces of data that passed through it, but it decided not to cache it due to space limits. Hence, incoming data requests can be *detoured* to a neighbor CR which may have the requested piece of data, reducing the overall bandwidth consumption and latency in the network that would result by forwarding the request directly to the server.

4.2.3 Content identification

In order to address resources in the Internet and cache them in the CR, we use identifiers that are simultaneously independent from the forwarding, routing, storage location and the underlying transport protocol. Thus, we use content identifiers that are solely based on the content called cryptographic identifiers (cryptoID) (Moskowitz et al., 2008). The benefit of using cryptoIDs are threefold: first, cryptoIDs result from a strong cryptographic hash over a data block, strongly binding the content identifier with the data that it carries; second, the cryptoID namespace is homogeneous since it results from a standard cryptographic hash function and does not need an external authority to manage the namespace; third, cryptoIDs are not bound to any specific protocol, i.e., content identification is not an internal parameter from a protocol but it exists by its own.

The basic unit of communication used in the in-network caching architecture is a data *chunk*. A chunk is a piece of data that is identified by a cryptoID with variable length. Content providers generate data chunks and use a cryptographic hash function to generate the chunks' cryptoIDs. Then, they aggregate the cryptoIDs together into meta information structure called *metadata*. The metadata also contains additional information about the content, for example,

version and validity, and the chunk list is ordered to allow the correct reconstruction of the original content. Therefore, clients need to retrieve the content metadata prior to the data chunks download from a trusted place, e.g., a security plane described previously. For legacy applications, we use CR-proxies to perform the name to metadata resolution and the chunk retrieval (described below).

4.2.4 Content security

We use the SHT as the authentication data structure for the secure caching model. Content providers generate SHT over pieces of content and sign the root hash of the SHT of a content. Later, whenever a client request for that content, the provider sends it together with the authentication path, allowing for intermediate CRs to verify the content integrity. CRs also need to have the provider's public key in order to verify the signature on the root hash. Therefore, we assume that CRs are managed by an ISP or a network administrator who has rights to add or remove public keys in the CR. In this scenario, administrators can obtain the public key directly from the content provider and insert into the CRs. Content providers can also publish their public keys into a security plane and administrators can manually verify their digital signature and insert them into the CRs.

4.3 Implementation

In this section we present the implementation of the CR mechanism. The CR is implemented as a background service running in Linux machines, composed of a kernel module and a userspace management unit, shown in Fig. 6.

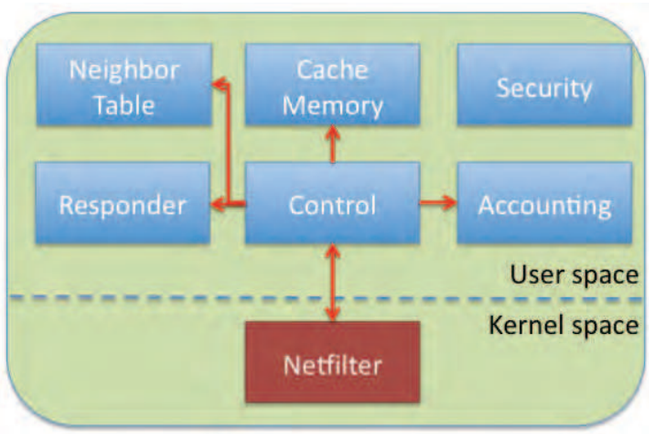


Fig. 6. Internal modules of a content router.

The *Netfilter* module is located in the kernel space and it is responsible for intercepting chunk request and response messages from the network, and delivering them to the *Control* module. This module uses the kernel *netlink* interface to capture packets directly from the kernel space and divert them to the user space in the *Control* module. The *Control* module handles the packet processing and forwarding, receiving data from the kernel space, caching and forwarding based on the *neighborhood table*. The *Security* module is responsible for SHT verification using the appended authentication path. The *Cache Memory* is responsible for storing the data itself and the initial version is implemented as a hash table. The *accounting* module is responsible for collecting the statistics about the data popularity based on the requests and responses passing through the router. These statistics will be used by the

cache memory to help the cache eviction policies. The *Neighborhood Table* contains forwarding information collected from in-transit data messages in the network together with the last-seen information. Finally, the *Responder* module is responsible for returning cached data to clients on the server's behalf.

The forwarding mechanism based on cryptoIDs between content routers use a special header containing details about the carried data. Fig. 7 illustrates the packet header used for the content discovery and forwarding mechanism based on cryptoIDs.

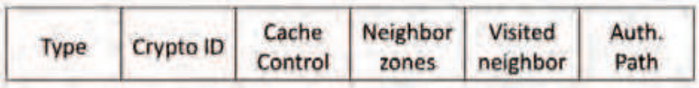


Fig. 7. Caching control header

The *type* field has a 8-bit field describing the type of the message, for instance, chunk request or response and signaling between CRs. The *cryptoID* is the permanent content identifier and it is generated using a cryptographic hash function, e.g., SHA-1, over the data. The *cache control* field has 8-bit length and provides signaling information for the routers, for example, whether a data chunk has already been previously cached in the network. In this case, the *cached* flag has one bit and it is stored within the *cache control* header. The *neighbor zone* field has 8-bit length and contains the number of neighbors that a message should visit before going directly to the server. The *Auth. Path* field contains the variable length authentication path for data verification.

The current version of the CR is implemented over UDP datagram as the forwarding mechanism, running on ports 22000 and 22001 in Linux OS machines. Clients send data requests to the servers and intermediate CRs cache these information in the popularity table, as they will be used as input parameter for caching policies. Whenever a CR intercepts a passing-by request or response, it may cache it based on the caching policies, e.g., popularity of the requests and responses. Whenever there is a data chunk message, CRs have a probability to cache it in their cache memory to serve for further requests.

4.4 Evaluation

In this section, we evaluate the CR proposal regarding the security mechanism based on the SHT.

4.4.1 Experimental set-up

In order to evaluate the CR authentication mechanism and to compare with per packet signature scheme, we implemented a CR prototype in C language. We used a Mac OSX 10.6, 2.16GHz, 2 GB RAM for the evaluation scenarios. In the first scenario, we evaluated the speed of the RSA public key signature and verification times and the SHA-1 hash function using the OpenSSL cryptographic library. The purpose of the evaluation is to establish the magnitude between a hash verification time and a digital signature and verification times. For the second, third and forth evaluations, we used the topology described in Fig. 8. The topology is composed of a client, a server that sends some data to the client and a CR in the border of the network where the client is located. For each test case, we collected 10 samples and considered the average value to plot the graphics.

Fig. 8. Evaluation topology with one CR.

4.5 Experimental results & analysis

Tab. 2 shows the experimental evaluation of different cryptographic algorithms for signature and verification. For the SHA-1 verification speed, we considered a packet of 1024 bytes. As the experimental results show, a digital signature costs roughly 388 times slower than a hash verification and 18,51 times slower than a hash verification (SHA-1 vs. RSA 1024). The comparison is to show that if we can reduce the number of digital signatures in a large file transfer, we can considerably reduce the processing overhead resulted from the verification process. In addition, clients generating data wouldn't suffer from the delay due to the signature process.

Type	Signatures/s	Verification/s
SHA-1	-	222,402
SHA-256	-	96,759
RSA 1024 bits	573	12012
RSA 2048 bits	95	3601
ECC 160 bits	4830	1044
ECC 163 bits	1376	563

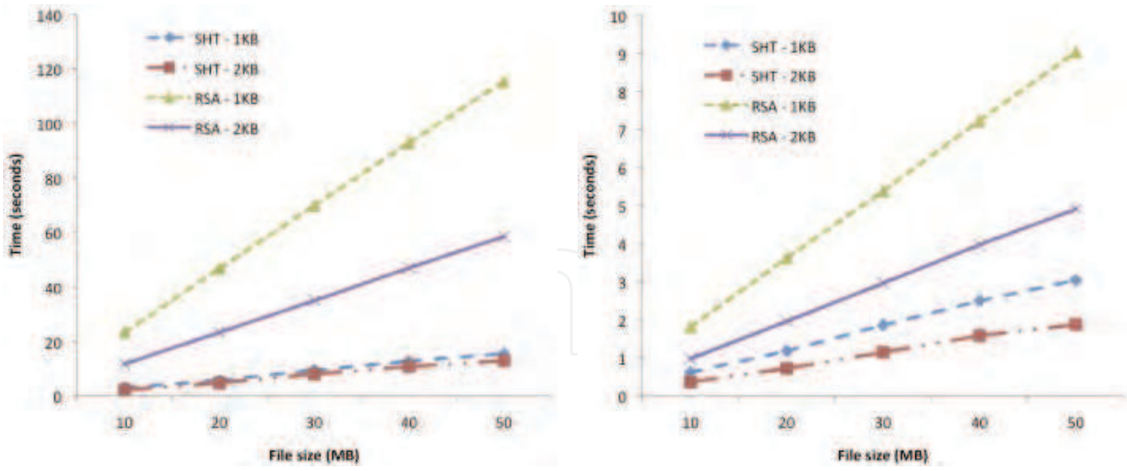
Table 2. Signature and verification speeds with different cryptographic algorithms

In the second evaluation scenario, we analyzed the root hash generation time using the SHT algorithms with the SHA-256 cryptographic hash function. We selected files ranging from 10 to 50 MB and used block sizes of 1, 2 and 4KB in the algorithm. The results are summarized in Fig. 9(a).

The figure shows that the *Root Hash* computation grows linearly with the file size and the number of data blocks. This result is predicted since the number of hash computations in the hash tree is linear to the number of data blocks. Note that the root hash has an equivalent functionality as the public key in the PKI, since it is used to verify the authenticity of a signature, but with much faster computation time.

In the second evaluation, we compared the SHT authentication path generation time with a 1024-bit RSA signature time, shown in Fig. 9(b). We implemented two applications for this evaluation: the first one reads from an input file in blocks of 1, 2 and 4 Kbytes and apply the skewed hash tree function, and the second one reads from an input file in blocks of 1, 2 and 4 Kbytes and digitally sign each block with a 1024-bit RSA key. Both of them used the SHA-256 cryptographic hash algorithm to produce the digest messages to be signed. We excluded the 1024-bit RSA key generation time from the results, since we generated it once and used the same key in the evaluation.

The results show that SHT mechanism is on average 8 times faster than the per packet signature approach. This result is expected since the digital signature computation uses large prime numbers, requiring high processing in the CPU. One the other hand, hash functions



(a) SHT AP Generation Times (b) SHT Verification Times

Fig. 9. (a) SHT Root Hash Generation Times.

rely on bit shifting to generate the digests, resulting in lower power consumption and memory storage.

In the third evaluation, we compared the authentication path verification time in a CR with a 1024-bit RSA verification time considering different file and block sizes. We used the same applications, block size and cryptographic hash function as in the previous scenario. The experimental results from both applications are summarized in Fig. 9. The verification times in the SHT is on average 3 times faster than the per packet signature scheme, which are expected since the hash value computation is much faster than the public key cryptography.

4.6 Parallel authentication over HTTP

Some software companies are already providing Metalink[7] files for users, such as Ubuntu and OpenOffice, so clients have more source options to download the packages. The benefit for the vendors is the reduction on the load on their main servers since users can also use P2P protocols to retrieve data. Apple also started to implement their own protocol for parallel content download, known as *apple streaming* (Pantos, 2010). In this protocol, users receive a *playlist* file containing a list of URLs from where a client can download the data. Each URL points to a segment of the original data, for example, 10 seconds of a music, thus, users can fetch all segments in parallel, reducing the overall download time.

Although the Metalink framework improves the performance of content download, the security mechanisms are not explicitly addressed, and they are basically inherited from the traditional security protocols. Both CDN and Metalink framework use the HTTPS as the default security mechanism to provide content authentication and integrity. For the former case (CDN), it is not actually a problem since the owner of the CDN also owns the infrastructure. Thus, the surrogate servers are considered *secure* servers and the owners are responsible for its maintenance and protection against attacks. But if there is an attack on a

[7] The *metalink* proposal aims to provide Web users with a metadata file containing information about how multiple data chunks can be retrieved from a list of sources, the geographical location of the servers and the preference level on each server.

surrogate server and a target content is tampered, the HTTPS will not accuse any problem, since the end-points are authenticated. Unfortunately, the authenticity of the data is inherited from the authenticity of the host, which is not always true[8]. For the latter case (Metalink), as the content provider may not own the infrastructure that will deliver the content, e.g., a P2P network, the security issues are more critical, as malicious node can tamper the data, preventing users to correctly retrieve the content.

There is no native security mechanism to provide data authentication and integrity efficiently in information-oriented networks, leaving the client unprotected against corrupted data. One naive approach is to establish SSL/TLS tunnels with each server to authenticate the storage place. However, this approach has some drawbacks: first, it is inefficient to open multiple SSL/TLS channels, since it consumes resources on both sides, decreasing the scalability in the server; second, in this specific scenario, we are actually authenticating the storage server and not the data itself. We argue that the trust relationship is misplaced since we are placing the trust in the connection instead of the content itself.

Another approach adopted by content providers is to provide the hash digest (e.g. MD5 or SHA-1) of the entire content to guarantee the content integrity. Although this approach works well for a unicast communication scenario, where there is just one download channel, applications are only able to verify the content integrity after the complete file download, making it hard to spot corrupted data chunks in the middle of the transmission.

In this section, we present an amortized verification mechanism using composite hash trees (Wong et al., 2010b), allowing applications to efficiently verify data chunks as they arrive from multiple sources. The hash tree mechanism allows for fast verification and requires just one hash computation per data segment in the best case. The proposed mechanism can be tweaked to satisfy specific application requirements, e.g., the total overhead and also the dependency between data chunks. The main difference of our approach compared to the traditional SSL/TLS-based authentication is that we enforce the content authentication and integrity based on the information that each data chunk carries instead of binding the authentication procedure to one specific source. The proposed approach has the following benefits: i) data can be more easily shared among users without requiring the verification of the serving host since the authentication information is embedded in the data; ii) fast verification, we just need one hash function per data block to check the integrity in the optimal case; iii) cheap authentication, one digital signature regardless of the number of data chunks; and iv) higher granularity to detect corrupted data chunks, making it possible to re-download it as soon as it is detected.

4.7 Parallel verification proposal

We first start presenting the metrics and design goals for our parallel verification mechanism. Then, we map these requirements on the composite hash tree data structure for authentication and verification procedures. Lastly, we describe an application scenario for the composite

[8] As an illustration of this scenario, consider two friends Alice and Bob. Alice trusts Bob and vice-versa and they know that they will not harm each other. Alice needs to borrow some money from Bob and Bob acknowledges that. Despite the fact that Alice knows Bob (authenticated him), there is no guarantee the bill that Bob will give to her is original or fake one (content authentication). Bob is also honest and does not want to fool Alice, but if he has received a bill that is fake and didn't realize that, he will give to Alice as a original one. Therefore, the authentication of the source does not yield to authentication of the content.

hash tree in the parallel content retrieval context and present an analytical evaluation of the proposed verification mechanism.

4.7.1 Design & rationale

In order to design a parallel verification mechanism, we considered three metrics for our model: *ordering, verification overhead* and *CPU processing cost*.

- *Ordering.* This metric considers the degree of dependency between the data chunks during the verification procedure. For example, hash chains (Yih-Chun Hu, M. Jakobsson and A. Perrig, 2005) require *strict* ordering in the verification procedure, while per packet signature (Catharina Candolin, 2005) or Merkle Trees (Merkle, 1989) can provide independent packet verification (therefore, these mechanisms support *true* parallel verification).
- *Verification information overhead.* The verification information overhead, e.g., the amount of data that a packet should carry in order to provide independent verification, should be as small as possible.
- *CPU processing cost.* The verification should be fast and, preferably, at line speed.

Based on previous requirements, our goal is to have a mechanism that has none (or low) ordering requirements, low verification information overhead and low CPU processing costs. In order to achieve these requirements, we propose an authentication/verification data structure based on *composite hash trees* since it provides an efficient data verification mechanism with *low verification overhead and CPU processing cost* at the cost of *an initial verification ordering* requirement.

4.7.2 Parallel verification procedure

The parallel verification procedure uses the composite hash tree mechanism to provide parallel verification information retrieval together with the data blocks. The goal is to retrieve data chunks from the servers and simultaneously establish a verification relationship between the previously received data authentication blocks with the incoming ones. Fig. 10 shows an example of parallel data chunk retrieval and verification from multiple Web-servers.

In order to enable the parallel verification procedure in the Web, clients must first retrieve the CHT from either a trusted source or embedded in a digital certificate, illustrated in the step 10(a). After the verification procedure of the CHT, the client can initially open two parallel connections to retrieve the two authentication data blocks (AD) that are direct children of the CHT in the tree. After retrieving one AD, the client can verify it and open more connections to retrieve more data chunks in parallel, as shown in step 10(b). The number of connections is limited to two in the beginning of the procedure, increasing by a factor of α connections for every AD retrieved, as illustrated in Fig.10. Finally, in step 10(c), the AD is used to verify the incoming data chunks.

The verification procedure is similar to the one presented in Fig. 5. The figure has two columns, the first one indicates the received data chunks and the second one shows the *next chunk* window which could be downloaded next. As more ADs arrive in the client, there are more options of data chunks to be downloaded since each AD contains a list of RH that can be used to authenticate the data chunks in the hash tree leaves. Therefore, every time that an AD arrives in the left side, it is expanded and the blocks that it can verify are placed in

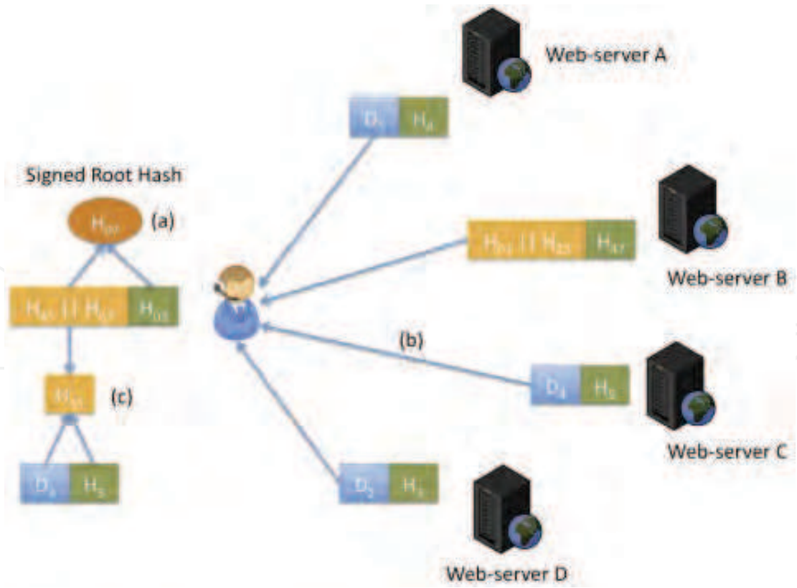


Fig. 10. Parallel data verification scenario. (a) First, the application retrieves the RH and verifies the digital signature. (b) The application retrieves ADs and subsequent data blocks from multiple sources. (c) Data blocks are verified using the previously received ADs.

the right column. For example, after the receiver authenticates the AD_0 containing the hash values $H_{01}||H_{23}$, the user can start downloading data blocks D_0, D_1, D_2 and D_3 in parallel and verify them as they arrive.

After the destination receives the AD with the concatenated hash values $H_{01}||H_{23}$, the receiver can retrieve and authenticate the data blocks D_0, D_1, D_2, D_3 in whichever order. The same procedure is taken when the AD with concatenated hash values $H_{45}||H_{67}$ is received in the destination, allowing the parallel retrieval and authentication of data blocks D_4, D_5, D_6 and D_7.

4.8 Evaluation

In order to compare with other approaches, we perform an analytical evaluation of the CHT overhead using different configurations. As demonstrated in Section 3.3, the composite hash tree has two overhead associated, the *Authentication Path* (O_{AP}) overhead and *Authentication Data Block* (O_{AD}) overhead. The O_{AP} is the sum all AP in each intermediate hash tree, defined by the CHT height h and the O_{AD} computes the sum of the product of the intermediate RH values that are aggregated into α hash values. The total CHT overhead of a CHT (O_T) with height h and aggregation index α is:

$$O_T = N * h + (N * h)/2^h = O(N) \tag{12}$$

The CHT parameters can be tuned to fit the overhead and dependency requirements specific to applications, for instance, in delay sensitive applications, e.g., video streaming, it is interesting that we start downloading the blocks with low latency between them. As applications can open multiple channels, it can check the available bandwidth on each connection and select the one that is providing higher throughput. On the other hand, applications that are not delay sensitive, e.g., file-sharing applications, we can use CHT with higher intermediate hash

trees but with lower verification overhead. In that case, smaller data blocks provide faster dissemination, and in our case, it allows us to switch faster between sources after completing a chunk download

In order to analyze the performance of CHT with different parameters, we selected a file of 1 GB which we divided in blocks of 64KB, resulting in 16384 data blocks and we chose an AD with size of 8KB. The decision to choose small data blocks is due to the possibility of switching between sources faster since we can finish one download faster in order to start with another source with higher throughput, similar to the way how P2P systems work. We first start computing the α value:

$$\alpha = \frac{block\ size}{hash\ size} = \frac{8KB}{20B} = 400 \tag{13}$$

Therefore, each AD will hold 400 intermediate *Root Hash*es. The hierarchy dependency will be:

$$H' = \lceil \log_{2^h \alpha}(N/2^h) \rceil = \log_{800} 8192 \approx 1.35 = 2 \tag{14}$$

And the total overhead will be (according to Eq. 12):

$$O_T = N * h + (N * h)/2^h * 20(hash\ size) = 480KB \tag{15}$$

Tab. 3 summarizes the overhead for different h values for a file of 1 GB divided in blocks of 64KB.

CHT configuration	$h = 1$	$h = 2$	$h = 3$	$h = 4$	$h = 5$
Overhead (KB)	480	800	1080	1360	1650

Table 3. CHT overhead vs. authentication hierarchies

Hence, the total overhead for a CHT with $h = 1$ and $\alpha = 400$ is 480KB in a file of 1GB, resulting in less than 0.5% of total overhead at the cost of two verification steps before authenticating the data blocks. Another benefit from the security point of view is the fact that all blocks are algorithmically bound together, making it possible to clients to authenticate the authentication information. Compared to a regular *.torrent* used in BitTorrent, the main benefit is that we provide a mechanism to authenticate the partitioned authentication data, while the transfer of the *.torrent* file would require some other mechanism, e.g., hash chains or a single cryptographic hash over the entire metadata, to authenticate the structure containing all the piece IDs.

Fig. 11 summarizes the CHT overhead using different configurations of h and block sizes. Note that the overhead does not grow linearly, but logarithmically with the height of the internal hash tree (h), and the α parameter does not influence the overhead, but just the hierarchical dependency. Tab. 4 shows a comparison of the overhead with different file sizes and CHT configurations.

Fig. 12 shows the hierarchical dependency needed to authenticate data blocks with different h and α parameters. For this analysis, we considered a file of 1 GB divided in blocks of 64KB, resulting in 16384 data blocks. By using higher values of h, we are able to reduce the number of intermediate AD that we need to authenticate before verifying the data blocks themselves.

The graphic illustrates that for a given α value, the selection of the internal hash tree height h value does not interfere with the number of hierarchy dependencies but changes the overall

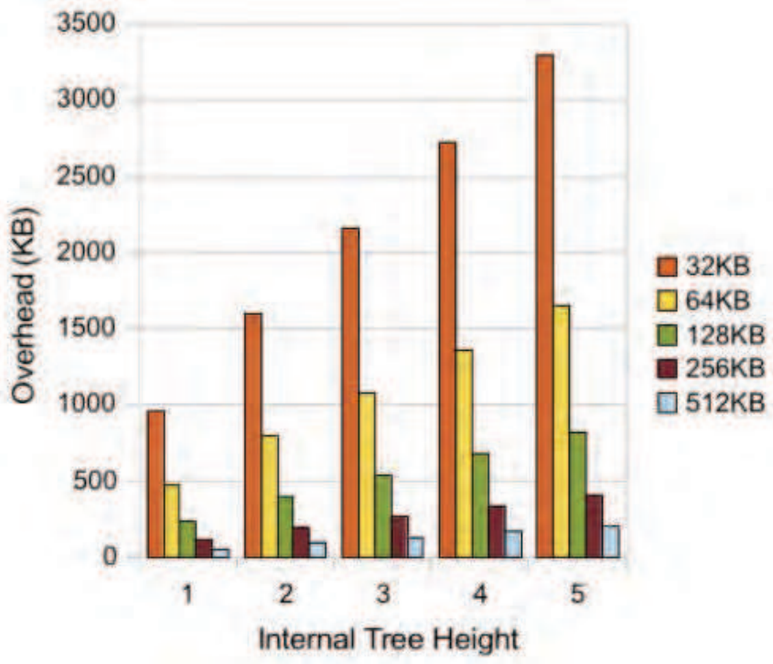


Fig. 11. CHT Overhead comparison using different internal hash trees for a file of 1GB divided in blocks of 32, 64, 128, 256, 512KB.

CHT conf.	1 GB	2 GB	5 GB	10 GB	20 GB	32 GB
CHT(1, 400)	0.47	0.94	2.34	4.68	9.37	15
CHT(2, 400)	0.78	1.56	3.91	7.81	15.62	25
CHT(3, 400)	1.05	2.11	5.27	10.54	21.09	33.75
CHT(4, 400)	1.33	2.65	6.64	13.28	26.56	42.50
CHT(5, 400)	1.61	3.22	8.06	16.11	32.22	51.56
Merkle Tree	4.38	9.38	25.5	54.13	114.51	190

Table 4. CHT overhead (MB) vs. file size using data chunks of 64 KB.

overhead. For instance, if we pick $\alpha = 400$, it is equivalent to select h equal to $1, 2$ or 3 since they will result in the same hierarchical dependency between blocks. However, as Fig. 12 shows, higher h values result in higher overhead. Therefore, the best option here is to select the smallest $h = 1$ to minimize the overhead. On the other hand, if we consider $\alpha = 50$, the value of $h = 1$, $h = 2, 3, 4$ and $h = 5$ have different hierarchical values and also overheads, being a choice of the application to select the one that best fits the application's requirements.

4.8.1 Legacy data support

The proposed parallel authentication mechanism also supports legacy data from content providers, meaning that providers do not need to introduce any modifications in the files, for instance, to fragment the files into data chunks beforehand to insert the verification data (AP). As the verification data is unique and it is generated from the data segment, it is possible to detach the verification information from the data. Therefore, applications can retrieve data segments from possible sources and the AP from an authentication server or a security plane.

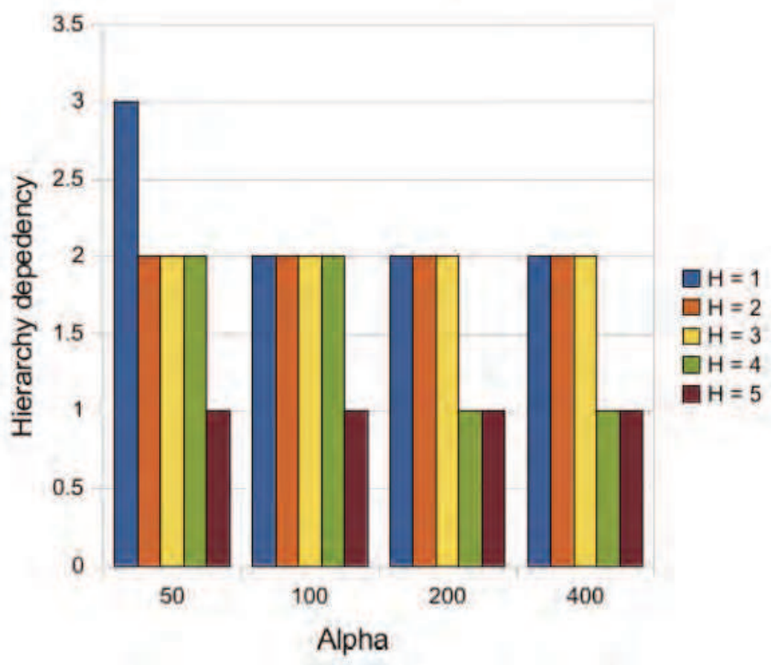


Fig. 12. Hierarchy dependency vs. aggregation index (α) using different internal hash tree heights.

The content retrieval procedure from different sources starts with the metadata file retrieval from a trusted source, e.g. Metalink signed metadata or from a security plane. The metadata contains the segment sizes and the corresponding authentication ID used to authenticate the data block. Then, a client contacts a directory server to retrieve the *authentication data blocks* and the *authentication path* for each segment. Next, the client starts the verification of the AD until reaching the AP of each data block, discarding the intermediate values. In the next step, the client retrieves the metadata containing the segment sizes in order to download the segments from multiple sources using multiple protocols, e.g. HTTP and FTP. In HTTP, it is possible to use the `HTTP Range Request` header to request a specific segment size, and in FTP we can use the `seek` directive to request a data range. After retrieving the data segment, the application applies a cryptographic hash over the data segment and computes the intermediate *root hash* using the previously retrieved AP for the data block.

Another extension supported by the parallel retrieval is the *opportunistic* verification. The idea of the opportunistic authentication is that users start to retrieve both data and authentication data simultaneously from multiple sources instead of downloading the verification information from the authentication server. In this approach, applications do not need to wait for the AD retrieval before the data. The application just places these unverified data blocks in an outstanding table and, as soon as the verification data arrives, it checks the integrity and saves into the destination file.

4.9 Related approaches

SINE (C. Gaspard, S. Goldberg, W. Itani, E. Bertino and C. Nita-Rotaru., 2009) provides Web content integrity using a hash list scheme. The idea is to add the hash of the following block in the previous block and digitally sign the first block sent to the client, which is also known

as *chain anchor*. Therefore, modifications in any of the following blocks can be spotted by computing just one hash function over the next block. The main benefits of SINE is that it requires just one digital signature to authenticate an entire piece of data regardless of the number of data blocks and use one hash function to check the integrity, resulting in both low verification header and CPU cost. The main drawback compared to CHT is the strict verification order of the pieces, therefore, not supporting parallel verification of data chunks.

Regular Merkle Trees (Merkle, 1989) create a hash tree over a set of data blocks and each piece of data carries $\log_2 N$ hash values allowing them to authenticate data blocks with the corresponding root hash. The benefits are the independent data block verification and the low CPU processing costs. The main drawback is the verification information that each data block must carry, resulting in a total overhead of $N * \log_2 N$, being a considerable overhead for files with large number of blocks.

Packet Level Authentication (PLA) (Catharina Candolin, 2005) is a security model focused on per packet authentication, providing data authenticity and integrity in the network. Before a data block is sent to the destination, it is digitally signed by its provider, who is also endorsed by a trusted third party. The benefit is the independent block authentication with constant verification information overhead. However, the main drawback is the cost associated to the digital signature and verification, making it unfeasible to use in low processing devices. Tab. 5 summarizes the comparison between these mechanisms with the CHT approach. We took into account the *ordering* requirement, the *verification data overhead* and the *CPU cost associated* with the verification.

Mechanism	Block Association	Verification data	CPU cost
Hash chain	strict ordering	O(N)	low
Merkle Tree	independent	O(N*$\log_2$N)	low-medium
PLA	independent	O(N)	high
CHT	independent*	O(N)	low

Table 5. Comparison between verification techniques

The CHT mechanism inherits the low verification data overhead and CPU cost from the hash tree mechanism at the cost of an initial dependence between the first data block. After the second one, it works similarly to the regular Merkle Tree, but with linear overhead instead of O(N*$\log_2$N).

5. Conclusion

In this paper, we have proposed two hash tree mechanisms, the *skewed hash tree* (SHT) and the *composite hash tree* (CHT) mechanisms to provide efficient content authentication mechanism based on the content rather than the connection. The first technique (SHT) is an extension of the Merkle Tree, which allows for random size authentication. The SHT mechanism can be created over a list of blocks and required one simple digital signature to authenticate all of them. The second mechanism, CHT, allows for efficient content authentication with reduced authentication overhead. CHT uses smaller Merkle Trees to reduce the overall authentication overhead at the cost of some hierarchical authentication dependence.

In order to validate our ideas, we implemented these two techniques and applied into two different scenarios: secure caching and parallel authentication over HTTP. The first evaluation scenario has shown that SHT can provide 8 and 3 times faster signature and verification speeds compared to public key cryptography, and the second evaluation scenario has showed

that the CHT authentication overhead for parallel authentication and be less than 1% in some configurations.

6. References

Arianfar, S., Ott, J., Eggert, L., Nikander, P. & Wong, W. (2010). A transport protocol for content-centric networks. extended abstract, *18th International Conference on Network Protocols (ICNP'10), Kyoto, Japan* .

Bayardo, R. J. & Sorensen, J. (2005). Merkle tree authentication of http responses, *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, ACM, New York, NY, USA, pp. 1182–1183.
URL: *http://doi.acm.org/10.1145/1062745.1062929*

C. Gaspard, S. Goldberg, W. Itani, E. Bertino and C. Nita-Rotaru. (2009). SINE: Cache-Friendly Integrity for the Web, *5th Network Protocol Security Workshop (NPSec'09)* .

Catharina Candolin, Janne Lundberg, H. K. (2005). Packet level authentication in military networks, *Proceedings of the 6th Australian Information Warfare & IT Security Conference*.

Leighton, T. (2009). Improving performance on the internet, *Commun. ACM* 52(2): 44–51.

Merkle, R. C. (1989). A certified digital signature, *Proceedings on Advances in cryptology*, CRYPTO '89, Springer-Verlag New York, Inc., New York, NY, USA, pp. 218–238.
URL: *http://portal.acm.org/citation.cfm?id=118209.118230*

Moskowitz, R., Nikander, P., Jokela, P. & Henderson, T. (2008). RFC 5201: Host Identity Protocol.
URL: *http://www.ietf.org/rfc/rfc5201.txt*

Pantos, R. (2010). HTTP live streaming, Internet Draft draft-pantos-http-live-streaming (Work in Progress).

Reis, C., Gribble, S. D., Kohno, T. & Weaver, N. C. (2008). Detecting in-flight page changes with web tripwires, *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, USENIX Association, Berkeley, CA, USA, pp. 31–44.

Rodriguez, P., Spanner, C. & Biersack, E. W. (2001). Analysis of web caching architectures: Hierarchical and distributed caching, *IEEE/ACM Transactions on Networking* 9: 404–418.

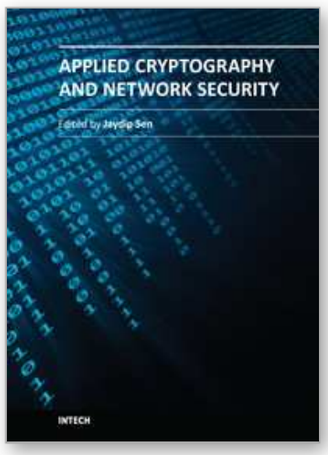
Tamassia, R. & Triandopoulos, N. (2007). Efficient content authentication in peer-to-peer networks, *Proceedings of the 5th international conference on Applied Cryptography and Network Security*, ACNS '07, Springer-Verlag, Berlin, Heidelberg, pp. 354–372.

Wong, W., Giraldi, M., Magalhaes, M. & Kangasharju, J. (2011). Content routers: Fetching data on network path, *IEEE International Conference on Communications (ICC'11), Kyoto, Japan.* pp. 1–6.

Wong, W., Magalhaes, M. F. & Kangasharju, J. (2010a). Piece fingerprinting: Binding content and data blocks together in peer-to-peer networks, *IEEE Global Communications Conference (Globecom'10), Miami, Florida, USA* .

Wong, W., Magalhaes, M. F. & Kangasharju, J. (2010b). Towards verifiable parallel content retrieval, 6^{th} *Workshop on Secure Network Protocols (NPSec'10), Kyoto, Japan* .

Yih-Chun Hu, M. Jakobsson and A. Perrig (2005). Efficient Constructions for One-Way Hash Chains, *Applied Cryptography and Network Security* pp. 423–441.



Applied Cryptography and Network Security

Edited by Dr. Jaydip Sen

ISBN 978-953-51-0218-2
Hard cover, 376 pages
Publisher InTech
Published online 14, March, 2012
Published in print edition March, 2012

Cryptography will continue to play important roles in developing of new security solutions which will be in great demand with the advent of high-speed next-generation communication systems and networks. This book discusses some of the critical security challenges faced by today's computing world and provides insights to possible mechanisms to defend against these attacks. The book contains sixteen chapters which deal with security and privacy issues in computing and communication networks, quantum cryptography and the evolutionary concepts of cryptography and their applications like chaos-based cryptography and DNA cryptography. It will be useful for researchers, engineers, graduate and doctoral students working in cryptography and security related areas. It will also be useful for faculty members of graduate schools and universities.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen