Algorithmica (2004) 39: 21–41
DOI: 10.1007/s00453-003-1076-8

Algorithmica

A General Model for Authenticated Data Structures[1]

Charles Martel,[2] Glen Nuckolls,[2] Premkumar Devanbu,[2] Michael Gertz,[2]
April Kwong,[2] and Stuart G. Stubblebine[3]

Abstract. Query answers from on-line databases can easily be corrupted by hackers or malicious database publishers. Thus it is important to provide mechanisms which allow clients to trust the results from on-line queries. Authentic publication allows *untrusted* publishers to answer securely queries from clients on behalf of trusted off-line data owners. Publishers validate answers using hard-to-forge *verification objects* ($\mathcal{VO}$s), which clients can check efficiently. This approach provides greater scalability, by making it easy to add more publishers, and better security, since on-line publishers do not need to be trusted.

To make authentic publication attractive, it is important for the $\mathcal{VO}$s to be small, efficient to compute, and efficient to verify. This has lead researchers to develop independently several different schemes for efficient $\mathcal{VO}$ computation based on specific data structures. Our goal is to develop a unifying framework for these disparate results, leading to a generalized security result. In this paper we characterize a broad class of data structures which we call *Search DAGs*, and we develop a generalized algorithm for the construction of $\mathcal{VO}$s for Search DAGs. We prove that the $\mathcal{VO}$s thus constructed are *secure*, and that they are *efficient* to compute and verify. We demonstrate how this approach easily captures existing work on simple structures such as binary trees, multi-dimensional range trees, tries, and skip lists. Once these are shown to be Search DAGs, the requisite security and efficiency results immediately follow from our general theorems. Going further, we also use Search DAGs to produce and prove the security of authenticated versions of two complex data models for efficient multi-dimensional range searches. This allows efficient $\mathcal{VO}$s to be computed (size $O(\log N + T)$) for typical one- and two-dimensional range queries, where the query answer is of size T and the database is of size N. We also show I/O-efficient schemes to construct the $\mathcal{VO}$s. For a system with disk blocks of size B, we answer one-dimensional and three-sided range queries and compute the $\mathcal{VO}$s with $O(\log_B N + T/B)$ I/O operations using linear size data structures.

Key Words. Authentic publication, Database integrity, Data structures, Security.

1. Introduction.

Large, complex, networked systems often have flaws that allow malicious outsiders to hack into them. Even mature, reliable systems are hard to configure and administer properly. One can rarely be sure that a large information system on the Internet is secure. Thus, the current state of security makes the trustworthiness of on-line publishing sources suspect.

How, then, can one provide increased assurance for the integrity of high-impact information (e.g., financial, medical, defense) securely on the Internet? In *authentic publication* a *client*, who only trusts a database *owner* (or creator), can use an *untrusted*, third-party publisher for query processing. The *owner* herself can remain safely off-line and simply provide the database periodically to publishers, who answer queries on the

[1] This work was supported by NSF Grant CCR 85961.
[2] Department of Computer Science, University of California, Davis, CA 95616 USA. {martel,nuckolls, devanbu,gertz,kwonga}@cs.ucdavis.edu.
[3] Stubblebine Research Labs, 8 Wayne Blvd., Madison, NJ 07940, USA. stuart@stubblebine.com.

Received December 6, 2001; revised February 18, 2003. Communicated by L. Arge.
Online publication January 27, 2004.

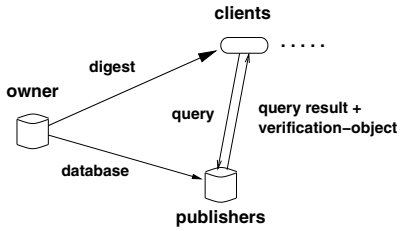

Fig. 1. Authentic publication scheme.

owner's behalf. When a *client* submits a query, the *publisher* responds with an answer $\mathcal{Q}$ and a *verification object* ($\mathcal{VO}$). The client uses the $\mathcal{VO}$ and some *digest values* provided securely by the *owner* to check that the answer is correct (see Figure 1). These schemes provide the following crucial guarantee: if the answer $\mathcal{Q}$ to a query is correct the *client* always accepts it, and if $\mathcal{Q}$ is incorrect, the *client* will detect that the accompanying $\mathcal{VO}$ is incorrect unless the attacker has found specific collisions in a collision-intractable hash function. In some settings there could also be a trusted *certifier* who constructs and securely distributes the digest values.

The use of an untrusted publisher reduces the risks of operating a secure on-line system: an attacker who gains control of a specific *publisher* would not be able to fool *clients*, who would reliably detect incorrect answers and simply find another *publisher*. It also allows graceful scaling by adding additional *publishers* to meet increasing demand from *clients*. Note that there are no secrets in this scheme: an adversary trying to fool the client is assumed to know all the data, the hash function, and all the digest values. Thus there is no privileged information to be compromised. Note, however, that this approach is currently only practical when the data is relatively static.

The $\mathcal{VO}$s must satisfy some critical requirements. First, they must be *secure*: it should be infeasible for a *publisher* to forge an acceptable $\mathcal{VO}$ for a wrong answer. Second, $\mathcal{VO}$s should be small, to reduce transmission overhead. Finally, they must be both *efficiently constructible* by the *publisher* and *efficiently verifiable* by the *client*. The $\mathcal{VO}$s we construct are cryptographic structures which use collision-intractable hash-functions to make forgery difficult.

The importance of finding good verification schemes for a broad range of queries has already led to the development of secure schemes for a variety of data structures. The original structures focused on membership queries: binary trees [20], [21], [22] and, recently, skip lists [12]. However, we are interested in handling a much richer set of multi-attribute queries such as "return all patches for versions 4.1 to 5.3 of Netscape which were released after July 1, 2000." We used two-dimensional range trees for this in [9], which, like all prior work in this area, used a binary-search-tree-like structure. However, many additional data structures can be used to support efficient answers to other types of queries. These include queries on strings (as for genetic databases), on XML documents, on images, and much more. Supporting these queries efficiently may require authentic versions of a variety of data structures.

However, as the data structures get more complex it can be hard to develop authenticated versions of these structures and to be sure the authenticated versions are secure. Thus we use a simple yet general data model which we call a *Search DAG* (for Directed

Acyclic Graph), and prove a security theorem for $\mathcal{VO}$s of Search DAGs. Our Search DAG model uses a new approach to verification which is general enough to include all existing authenticated data structures and much more. In particular, we can model hybrid data structures (e.g., combinations of trees, arrays, and linked lists) and the use of constraint information associated with the structure (e.g., ranges of values contained in a subtree). We also use our security theorem to prove the security of an authenticated version of a more complex I/O efficient data structure.

We briefly summarize the results below.

Main Results. There are three main results.

1. *General method for producing efficient authenticated versions of many existing data structures*. We improve on the efficiency of a result by Naor and Nissim [22] for producing authenticated structures from arbitrary unauthenticated versions. Our method usually eliminates the log factor (of the data structure size) overhead by making more effective use of the structure of the unauthenticated data structure. Though it is less general due to the additional structure, it still covers many data structures of interest which we demonstrate by using it to produce several new authenticated structures. We use the Search DAG model, which characterizes a broad class of data structures, and describe a general method to create efficient authenticated versions of any data structure in this class. We prove that any data structure which can be viewed as searching a DAG supports authentic publication and typically does so using $\mathcal{VO}$s whose size and construction time are linear in the search time of the underlying data model. Current approaches to authentic publication using binary trees, 2-3 trees, skip lists, and multi-dimensional range trees can be placed conveniently in the Search DAG framework. Thus our security theorems provide security proofs for these existing authentication structures.

2. *I/O Efficient $\mathcal{VO}$ construction*. Publishers with large data-sets may need to use I/O efficient structures (such as B-trees). We show how $\mathcal{VO}$s can be constructed with good I/O performance.

3. *New authenticated data structures*. We apply our model to produce improved authenticated data structures for the important class of multi-dimensional range queries which request all points in a d-dimensional rectangle.[4] This also models multi-attribute queries. In addition, answers to multi-dimensional range queries are important for supporting constraint query languages and queries on class hierarchies in object oriented databases [16]. Answers to two-dimensional range and three-sided queries (rectangles with one direction going to infinity) are the most important special cases to handle.

For range queries over N points with answers of size T, it is easy to use Search DAGs to create an authenticated version of Willard's data structure [26] which answers d-dimensional range queries with a $\mathcal{VO}$ of size and construction time $O(\log^{d-1} N + T)$. For three-sided range queries, we use Search DAGs to create an authenticated version of the complex data structure described in [2] to get $\mathcal{VO}$s of size $O(\log N + B + T)$, where B, the disk block size, is a parameter representing the amount of data which can be loaded into memory using one I/O operation. The $\mathcal{VO}$ for three-sided queries can be computed using only $O(\log_B N + T/B)$ I/O operations and using linear size data structures.

[4] An example two-dimensional range query is: return all points (x, y) such that $10 < x < 15$ and $30 < y < 50$.

In addition to these results, there are additional advantages to our approach. Since the $\mathcal{VO}$ construction and verification process both mirror the search procedure, creating an authenticated version of a given data structure automatically provides a template for many different search procedures on that structure. This simplifies the addition of authentication to existing structures and the use of existing search routines on the structure. We discuss this further in the sections on binary search trees and suffix trees.

In summary, we have applied the general Search DAG model to several efficient data structures. In each case the resulting $\mathcal{VO}$ construction algorithms exhibit *the same* asymptotic time, I/O, and space behavior as the original (non-authentic) data structures, and produce small $\mathcal{VO}$s. Security results for the $\mathcal{VO}$s also immediately follow from the general security theorem. These results show that our approach is applicable to a wide range of other data structures.

Related Work. The idea of data authentication has been considered for timestamping [15] and micro-payments [5]. The proposed techniques are based on the original work by Merkle [20], [21] and refinements by Naor and Nissim [22] for certificate revocation. Goodrich et al. showed that skip lists provide small, simple $\mathcal{VO}$s for authenticated dictionaries in [12] and Anagnostopoulos et al. describe persistent authenticated dictionaries in [1]. Devanbu and Stubblebine showed how to create authenticated versions of stacks and queues [10]. Recently, Goodrich et al. [13] show that a broad class of geometric data structures fit our general model and thus have efficient authenticated versions. This provides further evidence that this general result will be useful for data structures beyond those we describe in this paper. In [9] we introduced the general idea of authentic data publication, and we showed how to answer several types of relational database queries securely. This work was extended to authentic publication of XML documents in [8] using authenticated tries.

The papers above, like this one, share a common theme of leveraging the trust provided by a few digest values from a trusted party over multiple hashes, with the goal of protecting the integrity of the content, by efficient verification. However, our verification approach is more general since it explicitly uses both constraint information and a data structure's search procedure. This allows us to create authentic versions of more complex data structures fairly automatically. Buldas et al. [4] also use some constraint information in their work on certificate attestation, but the focus is limited to membership queries and binary search trees.

Naor and Nissim [22] describe a method for producing authenticated versions of arbitrary data structures. Their approach is to authenticate each part of the data structure accessed. Verifying the result of a membership query in a dictionary D incurs a cost of $O(T_{\mathrm{q}} \log |D|)$ where T_{q} is the query time in the unauthenticated dictionary.[5] Our approach eliminates the log factor overhead by incorporating the internal structure of the unauthenticated data structure in a very general way. Though incorporating this internal structure reduces the generality of our method, it is still quite general and captures most data structures of interest.

[5] Naor and Nissim focus on authenticated dictionaries though, as they note, their method generalizes to arbitrary structures.

Structure of the Paper. In Section 2 we introduce the basic security properties of verification objects ($\mathcal{VO}$s) and, as an example, show how to compute small $\mathcal{VO}$s for one-dimensional range queries efficiently. In Section 3 we describe the Search DAG model, and prove both its security and efficiency. In Section 4 we show that several existing authenticated structures can be easily modeled as Search DAGS, and we then apply Search DAGs to B-trees, tries, suffix trees, and multi-dimensional search trees. In Section 5 we apply our framework to an I/O efficient scheme for three-sided range queries. We then summarize our results and outline future work.

2. Background: One-Dimensional Range Queries. We outline the principles of our data publication scheme and illustrate their use for membership and one-dimensional range queries, showing the traditional approach to this setting and our new scheme. Authentic publication protocols involve three parties: an *owner*, who creates and is responsible for the content,[6] a *publisher*, who handles on-line query processing, and a *client*, who submits queries to the publisher. The *client* relies on the *owner* to create accurate data, but does not trust the *publisher*. The security of our schemes relies primarily on *collision-intractable hash functions* (CIHF).

Our protocols typically involve several steps. First, the *owner* computes a digest $\mathcal{D}$ of the content using a CIHF over a data structure containing all the data. This $\mathcal{D}$ is distributed securely to *clients*, perhaps using a public-key signature scheme. The *owner* then sends the data to the *publisher*. When queries are received from the *client*, the *publisher* sends back an answer $\mathcal{Q}$ and a verification object $\mathcal{VO}$. Using the $\mathcal{VO}$ and $\mathcal{Q}$, the *client* can recompute $\mathcal{D}$ to verify that $\mathcal{Q}$ is exactly what the *owner* would have given. We seek to guarantee the following important security property:

DEFINITION 1. An authentic publication protocol involving a *client*, a *publisher*, and an *owner* is **secure** if, given a digest (computed by the *owner*), a $\mathcal{VO}$ and an answer computed by the *publisher*, the *client* will only recompute the digest when the answer is just what the *honest owner* would have given, or the *publisher* has engineered a collision in the hash function used (a more formal definition is given in Section 3).

As background, we now describe the use of binary search trees for queries over an ordered set of data items $x_1 < \cdots < x_n$. First, build a binary search tree whose leaves are associated with the x_i values. Next, we compute a digest of the tree thus: using a CIHF h, the value of the leaf associated with x_i is $h(x_i)$ ($h1 \cdots h4$, as shown in Figure 2(a)). Each internal node's value is the hash of the values of its children. This construction, due to Merkle [20], [21], is called a Merkle hash tree and it has been used by several authors to solve problems related to authentic publication [22], [15], [9]. The root digest value $\mathcal{D}$ is distributed securely by the *owner* to the *client*. The data is then distributed to the *publisher*, who can build the same binary tree, and recompute the hash values.

For the tree in Figure 2(a), the publisher can prove that 23 is in the data set by providing three values as a $\mathcal{VO}$: $23,\ 45,\ h34$. We proceed bottom-up. Using the first two values the

[6] There might instead be a *certifier* responsible for assuring the correctness of the data and distributing a digest.

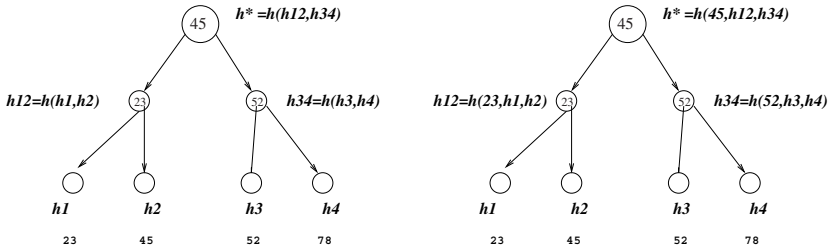

Fig. 2. Computation of digest values (a) and including split values (b).

user computes $h1$, $h2$, and then $h12$. $h12$ is then hashed with $h34$ to get the root digest. This $\mathcal{VO}$ also proves that 30 is not in the tree.

To illustrate our general data model better, where we want to be able to represent structural information, we now describe a different approach to digesting the tree and verifying an answer. Assume that internal node v_i has a data value d_i associated with it, where d_i is the largest value in the left subtree of vertex v_i (thus when searching for a value x, we go left at node v_i if $x \leq d_i$, otherwise right). The digest value of an internal node v_i will now be $h(d_i, L, R)$ where L, R are the digest values of the left and right children of v_i (see Figure 2(b)). With this type of digest it is easy to show that a value is (or is not) in the tree using a top-down approach. For example, to show 50 is not in tree we give as our $\mathcal{VO}$:

45, $h12$, $h34$;
52, $h3$, $h4$;
52

The first three values hash to the root digest $\mathcal{D}$, confirming that 45 is the correct split value. Since $50 > 45$ we know to move right (to the node with digest value $h34$). The next three values hash to $h34$, confirming 52 is the correct split value, so we know to move left. Finally 52 hashes to $h3$ (the left child digest value) confirming it is the correct leaf value and that 50 is not in the tree (since a search for 50 ends at a leaf of value 52).

In general this digest of the tree allows the *publisher* to give the *client* a $\mathcal{VO}$ which lets the *client* simulate a proper search in the tree and find the correct leaf values. Similarly, if the client asks for all values in the range (50, 60), it is straightforward to give a $\mathcal{VO}$ which simulates a search for all values in the tree which fall in that range. The $\mathcal{VO}$ will have a triple of values for each node visited in the search (range searches can also be easily verified using the simpler digesting scheme in Figure 2(a)).

Our new approach may be less efficient (since we hash three values instead of two) but it gives additional flexibility which we use for multi-dimensional searches where we may want to stop the search before reaching the leaves. This type of top-down $\mathcal{VO}$ can also make the $\mathcal{VO}$ more transparent.

3. A General Model for Query Verification. In the authentic data-publication setting the owner determines the most important part of the query verification structure: the

logical view which supports digesting and query verification. In this section we introduce a general model for digesting data and verifying answers to queries on the data. It is general enough to capture most existing data structures with search procedures and allows general use of constraint information about the data structures. This allows us to model more complex data structures than any of the prior authenticated structures.

Several existing computational models [17], [18], [24], [25] use (di)graphs and operations on node collections to represent the steps involved in computation. An overview of these is presented in [3]. Our model, called a Search DAG, can be viewed as a restricted version of these models and is used in a somewhat different context. Though it is intended to model certain, but not all, computations, it also needs to capture and generalize aspects of the efficient verification scheme we described in Section 2.

3.1. *Defining the Search DAG Model.* A *Search DAG* consists of a directed acyclic graph (DAG) $G = (V, E)$ and an associated deterministic search procedure P (we define P more precisely shortly). G has a unique *source* node s with in-degree zero. For each $v \in V$, the owner defines data associated with v which is denoted $a(v)$. This data can contain information about the successors of v as well as any other information relevant to the search procedure (such as constraint information) as well as data for query answers. The *sink* nodes of the graph have out-degree zero. We assume that each non-sink node has its successors specified in order (so we can refer to, e.g., the third successor of v).

The owner is also responsible for defining a deterministic search procedure P which takes a query q, searches G to find the appropriate associated data, and returns the correct answer for q.

Based on q, P begins searching G by reading the associated data $a(s)$, and outputting the next node v_2 to be visited (where v_2 is a successor of s). P next reads the data associated with v_2 to determine the next node to visit. P continues in this manner (always visiting successors of previously visited nodes) until it completes its search, then outputs **done** and Q, the actual answer to the query q. P can also output **reject** if it reads an invalid or inconsistent data item or a bad query. For convenience of our verification procedure, we assume that the *next vertex* output of P is in the form (j, k) which means: next visit the kth successor of the jth node visited (e.g., (3, 2) says next visit the second successor of the third node visited in the search). For any query q, the correct answer to q is defined to be the output of P when run on the owner's DAG G. This lets us define our setting for a broad class of queries without having to specify their specific semantics.

We note that in many settings the *client* will know the properties of G (e.g., for a binary search tree) and will be able to do the search without being given an explicit procedure P. Typically the non-sink nodes will have associated data which guides the search and the sink nodes will have associated data from an underlying data set. An example is the binary search tree described in Section 2 where a split value is stored at each internal node of the tree, and the data values are stored at the leaves.

Given a Search DAG, we now describe a general way to digest the DAG, create a $\mathcal{VO}$ for any answer to a query, and use the $\mathcal{VO}$ to verify the answer. Thus for any Search DAG we can automatically make it an authenticated data structure. We then show that the authenticated version retains much of the efficiency of the original Search DAG.

3.2. *Digesting the DAG.* The owner computes the digest value of the source in G using a collision-intractable hash function h. The digest function f is applied to every node in G and has a simple recursive definition:

$$f(v) = \begin{cases} h(a(v)), & v \text{ is a sink node,} \\ h(a(v), f(v_1), f(v_2), \ldots, f(v_k)), & \text{where } v_1, \ldots, v_k \text{ are the successors} \\ & \text{of } v \text{ in order.} \end{cases}$$

We can now describe the general authentic publication scheme at a high level. The *owner* chooses an appropriate Search DAG for his data set and gives a copy to the *publisher* along with h. Using a secure protocol, the *owner* sends the *client* the digest value $\mathcal{D} = f(s)$ (the "root hash"), the hash function h, and the search procedure P. We assume that P and h only need to be sent once, while $f(s)$ may need to be resent periodically to reflect updates to the *owner*'s data set. We now show how to verify answers.

3.3. *The Verification Objects and Verification Procedure.* We have defined correct answers to queries and how to digest our DAG. We now show that an untrusted publisher can provide a $\mathcal{VO}$ for any query defined for the search procedure P. We describe the structure of such a $\mathcal{VO}$ and the *client*'s verification process. We present a straightforward verification procedure for clarity, but a number of implementations are possible which adhere to the basic model.

We start by describing a correct $\mathcal{VO}$ for a query q, which we denote $\mathcal{VO}(q)$. Let $v_1(= s), v_2, \ldots, v_r$ be the nodes visited when P is run with input q on the *owner*'s DAG. We also let $u_1^i, \ldots, u_{k_i}^i$ be the successors of v_i. The correct $\mathcal{VO}$ for q is then the following values:

$$\begin{array}{l} a(s), f(u_1^1), \ldots, f(u_{k_1}^1); \\ a(v_2), f(u_1^2)), \ldots, f(u_{k_2}^2); \\ \ldots ; \\ a(v_r), f(u_1^r), \ldots, f(u_{k_r}^r); \end{array}$$

Each vector is ended by a ";" and is called a *step* of the $\mathcal{VO}$. An example of this type of $\mathcal{VO}$ is given in Section 2 for binary search trees. As in that case, the *client* verifies that the first step is correct by hashing the individual items in step one $(a(s), f(u_i^1))$ and comparing the result with $f(s)$. By construction, they match, so we input $a(s)$ to P. If the next node visited is the kth successor of s, the first output from P will be $(1, k)$. The second step is then checked by hashing the second step's values and comparing the result with $f(u_k^1)$. When it matches, we input $a(v_2)$ to P (otherwise we would reject). P then outputs (j, m) to indicate the next node visited (with $j = 1$ or 2). We hash step three and compare the result with $f(u_m^j)$, and when they match we input $a(v_3)$ to P, and so on. After inputting $a(v_r)$, P will produce the answer and halt.

Thus a correct $\mathcal{VO}$ is a sequence of vectors of integers (one vector per line) separated by semicolons. Any $\mathcal{VO}$ not in this form is immediately rejected. Thus any *syntactically correct* $\mathcal{VO}$ can be described as:

$$\begin{array}{l} x_1, y_1^1, y_2^1, \ldots y_{k_1}^1; \\ x_2, y_1^2, y_2^2, \ldots y_{k_2}^2; \end{array}$$

$\ldots;$
$x_r, y_1^r, y_2^r, \ldots y_{k_r}^r;$

We let $\overline{s_i}$ refer to the values in the ith step of the $\mathcal{VO}$. Given a query q and a syntactically correct $\mathcal{VO}$ V the verification process, $V_P(q)$, for V proceeds just as we described above for the correct $\mathcal{VO}$ for q (repeatedly hashing the values in a step, comparing the result with what it should be, and then giving the next data value to P to get the next node to visit). We continue in this manner until P halts and the answer is output (the verification is successful), a computed value mismatches (reject V), or we run out of steps in V (again reject V). In essence, V_P proceeds just as P would when searching G except that, at each node, the additional verification data for that node is processed.

3.4. *Security Theorem for the Verification Procedure.* We prove that a $\mathcal{VO}$ is accepted by our verification process V_P only if $V_P(q)$ verifies and extracts the same query data that the owner would have, unless the publisher was able to forge the $\mathcal{VO}$ for q. We first consider a particular type of bad $\mathcal{VO}$ which could trick our verification procedure.

DEFINITION 2. A syntactically correct $\mathcal{VO}$ V is a **forgery** of $\mathcal{VO}(q)$ if V has a step $\overline{s_i}$, such that $\overline{s_i}$ is not the same as the ith step of $\mathcal{VO}(q)$, but both steps hash to the same value using h.

We note that forgery is a necessary condition for subverting verification, but it is often not sufficient. For example, even if an attacker uses an alternate step $\tilde{x}_i, \tilde{y}_1^i, \tilde{y}_2^i, \ldots, \tilde{y}_{k_i}^i$ which has the same hash value as the correct step $x_i, y_1^i, y_2^i, \ldots, y_{k_i}^i$, the value $\tilde{x}_i$ may not be a valid input to P, so will be rejected. From our definition of a correct $\mathcal{VO}$ and query answer, we know that if the user is provided with a correct $\mathcal{VO}$ for a query q, then $V_P(q)$ will accept it and return the correct query data set Q. We can now prove our main security theorem for Search DAGs.

THEOREM 3. *Given a candidate $\mathcal{VO}$ V with $V \neq \mathcal{VO}(q)$, if V is not a forgery of $\mathcal{VO}(q)$, then $V_P(q)$ rejects V.*

PROOF. We prove that for all r up to the length of $\mathcal{VO}(q)$, if the first $r-1$ steps of V are correct, but line r is incorrect, then $V_P(q)$ rejects V after processing line r. The theorem follows immediately from this. The proof is by induction on r.

If $r = 1$, then $V_P(q)$ starts by hashing step one of V and comparing this to $f(s)$. Since V is not a forgery of $\mathcal{VO}(q)$ these two values match only if step one is the correct vector.

Induction step

Now assume that the first $r-1$ steps of $V_P(q)$ are correct (with $r \geq 2$). We show that $V_P(q)$ rejects after processing step r unless $\overline{s_r}$ is correct.

Since the first $r-1$ steps are correct, P has been given the correct first $r-1$ data values, and thus its output (which we denote (j,k)) after step $r-1$ is the correct next node to visit. $V_P(q)$ will next compare the hash of step r with y_k^j. Since $j < r$ we know y_k^j is the correct value to compare with (again by the induction hypothesis), and since V

is not a forgery of $\mathcal{VO}(q)$, the hash of step r will only match if the vector of values is the same as in step r of $\mathcal{VO}(q)$. Thus we reject unless step r is correct. □

3.5. *Complexity Results*. The security theorem applies to all Search DAGs, but our focus is on Search DAGs created from efficient search procedures. We now prove an efficiency theorem for an important class of Search DAGs. To do so, let $\mathcal{N}(q, G, P)$ be the number of nodes visited in the search of DAG G using procedure P to answer query q and let $\mathcal{P}(q, P)$ be the time taken by P to process q before it starts the search.

A search procedure P and a, typically infinite, set of Search DAGs S, are a *Bounded Search DAG Class* (S,P) if every Search DAG G in S has bounded out-degree, the data values $a(v)$ associated with the nodes of G are of bounded size, and P can process each data value $a(v)$ in $O(1)$ time.

THEOREM 4. *For a Bounded Search DAG class* (S, P), *for any search DAG* G *in* S, *any query* q *has* $\mathcal{VO}$*s of size* $O(\mathcal{N}(q, G, P))$ *which can be constructed and verified in time* $O(\mathcal{N}(q, G, P) + \mathcal{P}(q, P))$.

PROOF. The $\mathcal{VO}$ for q has $\mathcal{N}(q, G, P)$ steps, each of $O(1)$ size by the bounded assumptions of $a(v)$ and the degree of G. If each step is processed in $O(1)$ time the verification/construction time follows. □

Consider a set of unauthenticated data structures (e.g., all binary search trees) where $T(D, q)$ is the time to answer query q on a data structure D in the set. Suppose we have an associated bounded search DAG class (S, P) such that, for each unauthenticated data structure D in the original set, there is a corresponding search DAG G in S which, for each query q has, $\mathcal{N}(q, G, P) = O(T(D, q))$.

Then our method produces an authenticated version of each data structure D, such that each query q has proofs of size and verification time $O(T(D, q))$. In this case we eliminate the $\log(|D|)$ factor overhead produced in the Naor and Nissim method to create an authenticated version of D [22]. We note that their method is more automatic and general than ours since we require that a class of Search DAGs be produced from the original data structure. However, for most data structures it is fairly easy to create a Bounded Search DAG class with only a constant factor in extra overhead. Our examples in the later sections and prior results on authenticated structures demonstrate this ease of transformation for many important data structures. In addition, Goodrich et al. [13] show additional structures which have a Bounded Search DAG class with the same overhead.

The main situation where transforming to a Bounded Search DAG class is likely to create non-constant overhead, occurs when constant time indexing or hashing is used.

Our general authentication method may introduce larger constant factors than an approach tailored to the specific data structure, but once the basic digesting/verifying method is known, it is often easy to modify the verification process to exploit specific properties and improve efficiency. Most common is to do the verification "bottom-up" by starting with values from the fringe of the search and hashing them to get the predecessor node's value.

In the next section we show that it is easy to cast a number of basic data structures as Bounded Search DAG classes and thus get efficient $\mathcal{VO}$ schemes. However, the main

advantage of our model is for the more complex data structures we deal with in the final sections.

4. Efficient $\mathcal{VO}$s for Dictionaries, Range Queries, and Strings. We now give a few simple examples of data structures which are easily modeled as a Bounded Search DAG class. We start with structures which support dictionary queries: is element x in the data set? The structures also can support efficient insertions and deletions of new elements. Our general results for Search DAGs give easy security proofs for several standard structures.

A binary Merkle tree is the classic way to support an authenticated dictionary and it also defines a small $\mathcal{VO}$ for a one-dimensional range query. Section 2 described how to model a binary search tree as a Search DAG. It is also trivial to convert a 2-3 tree into a Search DAG if one wants to support efficient updates (as in [22]).

Skip lists [23] can provide an attractive alternative to trees, and Goodrich et al. recently showed how to create efficient $\mathcal{VO}$s for skip-list answers [12]: $O(\log N)$ size with small constants and efficient updates. A skip list is easily viewed as a DAG, and it is easy to create an authenticated skip list using Search DAGs. The obvious DAG for skip lists gives $O(\log N)$ access time but $O(N)$ update time. By splitting the nodes which are at the bottom of towers, it is easy to get a Search DAG which also has $O(\log N)$ update time. The digesting scheme used by Goodrich et al. is similar to this improved Search DAG, but they get better constants by using a bottom up approach and introducing a commutative hash function.

4.1. *I/O Efficient Construction of $\mathcal{VO}$s.* Binary search trees (BSTs) and skip lists are good for main memory implementations, but for large data sets requiring secondary storage, they have poor I/O performance. A classic way to get good I/O performance is to use a B-tree. Again, it is easy to cast a B-tree in Search DAG terms where each node's data is the $B-1$ split values to decide where to go next. Unfortunately, this would lead to larger $\mathcal{VO}$s of size $B \log_B N$ for data sets of size N. We would get a similar size $\mathcal{VO}$ using the traditional bottom-up verification.

We can reduce the size of the $\mathcal{VO}$ by replacing each B-tree node by a BST of height $\log B$. A search through this tree determines the next B-tree node to visit (we get a pointer to the root of the BST corresponding to that B-tree node). This new Search DAG has binary degree, so we get smaller $\mathcal{VO}$s for both membership and one-dimensional range queries. However, it is easy to store this tree in an I/O efficient manner since each BST corresponding to a B-tree node and its associated values can be stored in $O(1)$ disk blocks. Thus traversing this binary tree uses the same number of asymptotic I/O operations as for the B-tree.

Chazelle and Guibas describe a similar method for reducing the degree of a node in order to reduce the search time [6]. The main purpose for our use of a BST here is to reduce the $\mathcal{VO}$ size which, though a similar goal, is independent of reducing the search time. [7]

[7] Maniatis and Baker describe a similar scheme for reducing the size of the authentication data in [19].

This demonstrates a general technique which may allow high degree Search DAG nodes to be replaced by a tree of lower degree nodes. This also emphasizes the fact that the Search DAG is only a logical view of the data, and need not restrict the publisher's physical implementation. Now, consider a one-dimensional range query on N data points using disk blocks of size B. It follows from Theorems 3 and 4 that:

THEOREM 5. *For a one-dimensional range query with T answer points we get a $\mathcal{VO}$ of size $\Theta(\log N + T)$ which can be constructed with $\Theta(\log_B N + T/B)$ I/O operations using a multi-way tree of size $\Theta(N)$.*

PROOF. The $\mathcal{VO}$ size and security follow from Theorems 3 and 4 since we have a BST as our Search DAG. For the I/O results, note that we can store the BST for a B-tree node in O(1) disk blocks. The search looks at O(1) disk blocks for $\log B$ levels of the BST, and we only look at two leaf disk blocks (first and last) which do not contain $\Theta(B)$ answer points. □

The above describes a static multi-way tree, but I/O efficient updates to a B-tree translate into I/O efficient updates to our BST version of the tree as well.

4.2. *String Applications*. Data structures to support fast string searches are important in many settings. A trie [11] and its special form as a suffix tree are used in many applications [14] including our recent use of tries to authenticate XML documents [8]. As a tree-like structure, a trie is easily represented as a Search DAG. As long as the alphabet size σ is a constant, Theorems 3 and 4 apply to the Search DAG. Tries allow us to find a pattern P of length m in $O(m)$ time. Thus using a trie as a Search DAG, we can build a $\mathcal{VO}$ of size $O(m)$ which can be constructed in $O(m)$ time. Of course since the branching factor can be σ, the $\mathcal{VO}$ size is really $O(m\sigma)$. We can reduce the $\mathcal{VO}$ size to $O(m \log \sigma)$ by replacing high degree nodes by binary trees as we did for B-trees. Once we have digested the trie, it is also straightforward to authenticate answers to more complex searches, such as returning all patterns in a trie which match a regular expression. We just need to use the search procedure for regular expressions (as in [14]).

Suffix trees allow us to preprocess a long string S of length n in $O(n)$ time and space. We can then find if a pattern is in S in $O(m)$ time and find all k occurrences in $O(m+k)$ time. Suffix trees have many uses in string matching in general and computational biology in particular [14]. Since a suffix tree is a trie, we can use the results on authenticated tries. However, because a suffix tree is a compacted trie, a simple extension of our trie result blows up the space used. However, we can create an authenticated suffix tree which still uses $O(n)$ space, has the same search times, and has $\mathcal{VO}$s whose size is linear in the search time. As before, a hidden multiplicative constant of σ can be reduced to $\log \sigma$.

Although its construction is somewhat involved, a suffix tree can be described fairly simply. Every edge has a label, which is a non-empty substring of S, associated with it, and each internal node will have at least two edges to lower nodes in the tree. The substrings associated with the edges out of a given node each start with a distinct character. To find all occurrences of P in S, we follow the path from the root to a node v whose concatenated string labels are equal to P or have P as a prefix. The leaves in the subtree rooted at v correspond to all starting points of P in S (see Figure 3). The space can be

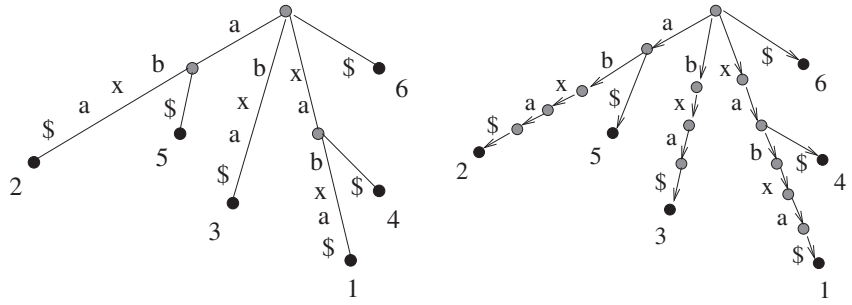

Fig. 3. Uncompacted suffix tree for $S =$ xabxa\$ and the resulting Search DAG.

reduced in a *compacted trie* by representing each edge label implicitly by position pairs (a, b) where a and b are the start and end positions respectively of the substring in S as in Figure 4.

It is easy to get a Search DAG if we use the $O(n^2)$ space version of the suffix tree which stores the entire substring corresponding to each edge of the tree. Our DAG is just the tree with each edge transformed into a chain of nodes, one for each character in the string (see Figure 3). We also want to retain the original edges of the suffix tree to make it easy to get to the leaves (which have the positions) quickly once we find a match.

For a fixed alphabet, the degree of the resulting Search DAG is constant. The client is perfectly happy with this model since his queries are answered in $O(m + k)$ time by the publisher, with optimal $\mathcal{VO}$s of size $O(m + k)$ and verification taking $O(m + k)$ as well. However, the publisher needs $O(n^2)$ storage as opposed to the optimal $O(n)$ for suffix trees.

The *publisher* would prefer simply to store the compacted suffix tree which consists of an edge labeled tree together with the original string where the edge labels are implicit pointers to positions in the string. To get an efficient Search DAG we explicitly combine

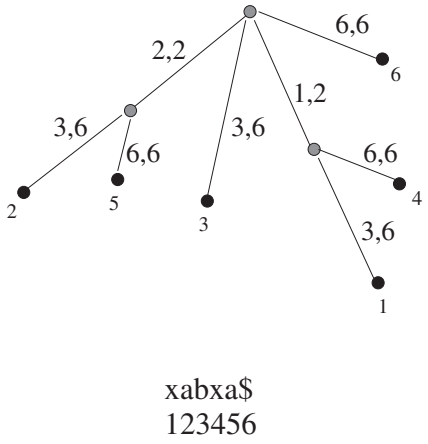

Fig. 4. Compacted version of the suffix tree in Figure 3.

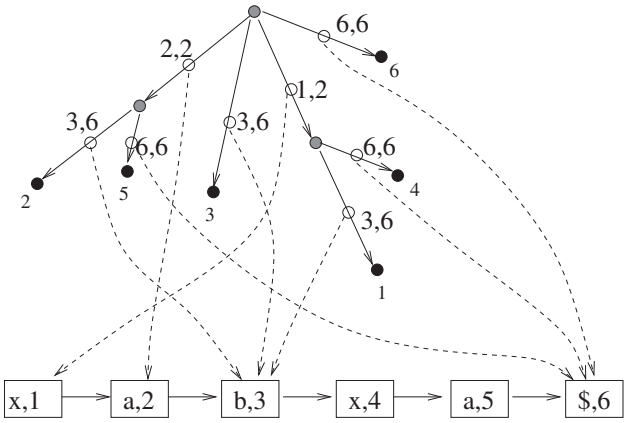

Fig. 5. Search DAG for the compressed suffix tree. Pointers to string nodes are shown from the corresponding index data for clarity.

these two structures (see Figure 5). During the search for a pattern P, the original string is referenced for each edge traversed. The Search DAG consists of:

1. The nodes of the compacted suffix tree with the same outgoing edges.
2. Another n nodes, one for each character/position of the original string S. These *string nodes* will have edges from position i to $i + 1$ for $i < n$.
3. The associated data for each of the n string nodes is just the character and position it represents.
4. The associated data at each internal node of the suffix tree will be the length of each substring on an outgoing edge.
5. Data at a leaf of the suffix tree is the position of the corresponding suffix of S as usual.
6. For each internal node of the suffix tree, a link to the string node corresponding to the first position of the substring associated with each outgoing edge (note that these pointers replace the edge labels on the compacted tree which are really implicit pointers).

It is straightforward to check that this is a bounded degree DAG with at most a constant amount of data associated with each node (given the bounded alphabet size). The search mimics the standard suffix tree search of a compacted trie in a natural way. Thus we can now apply our result for a Bounded Search DAG class to get the following result:

THEOREM 6. *If we want to find all occurrences of a pattern P of length m in a string S of length n with k occurrences of P in S, we can compute $\mathcal{VO}$s of size $O(m + k)$ for the answer which can be constructed and verified in the same time.*

4.3. *Multi-Dimensional Range Trees.* For a database containing N d-dimensional points, an orthogonal range query has the form $(l_1, u_1), (l_2, u_2), \ldots, (l_d, u_d)$ and asks for all points $(x_1, x_2, \ldots, x_d)$ such that $l_i \leq x_i \leq u_i,\ 1 \leq i \leq d$. These points can be efficiently found using *multi-dimensional range trees* (*MDRTs*) [7] in time $O(\log^d N + T)$. We briefly describe MDRTs which are easy to cast as Search DAGs. We discuss two-

dimensional queries, but they are easy to extended to higher dimensions. Authenticated MDRTs were also discussed in [9].

A two-dimensional MDRT (for N points (x_i, y_i)) consists of a *base* tree which is a BST whose leaves are the points sorted by the x-coordinate. In addition, each node v in the base tree has an associated BST $T(v)$ which stores all the points in the tree rooted at v, but in $T(v)$ they are sorted by the y-coordinate.

All points which satisfy a two-dimensional range query (xl, xu), (yl, yu) can be found by first finding in the base tree the $O(\log N)$ roots of disjoint subtrees such that their union is the points (x_i, y_i) satisfying $xl \leq x_i \leq xu$. For each such root v, we search in $T(v)$ for points satisfying $yl \leq y_i \leq yu$.

If each node v in the base tree has a pointer to the root of $T(v)$, and we associate the split value with each node of the base and associated BSTs, we now have enough information to direct the search. Thus, we get a Search DAG with each node of degree at most three.

So by Theorem 4 we immediately get a $\mathcal{VO}$ of size and construction time equal to the number of nodes visited, which is $O(\log^2 N + T)$ where T is the number of satisfying points. The above two-dimensional construction extends easily to a d-dimension MDRT (e.g., for three dimensions, for every node v of the base-tree, each node of $T(v)$ has an associated tree sorted on the z-coordinate).

THEOREM 7. *We can verify a d-dimensional range query on a data set of size N with T answer points using a $\mathcal{VO}$ of size $O(\log^d N + T)$ and this can be computed in the same time.*

4.4. *Improved Multi-Dimensional Range Queries.* Willard [26] shows that the search time can be improved by adding additional pointers to an MDRT. This avoids repeated searches to find the points in the associated trees which satisfy $yl \leq y_i \leq yu$. This reduces the search time and thus the $\mathcal{VO}$ size by a factor of $O(\log N)$. For our purposes we can view his additions as follows.[8] Let v_{l} and v_{r} be the left and right children of a node v in the base tree. For each node v in the base tree, a leaf of $T(v)$ (associated with point (x, y)) now has three outgoing pointers: one to the next larger leaf in $T(v)$, one to the leaf of $T(v_\ell)$ associated with (x', y') such that $y' \geq y$ and y' is the smallest such y-value in $T(v_\ell)$. There is also a pointer to a leaf of $T(v_{\mathrm{r}})$ whose associated point (x'', y'') is such that $y'' \geq y$ and y'' is the smallest such y-value in $T(v_{\mathrm{r}})$.

This structure allows us to do an initial search to find a y value which is in range, and we can then scan the leaves of the tree to collect all the points satisfying the y-range. We then follow pointers to the next tree we want to start the new scan. The new structure is a bounded degree DAG, so we can again apply Theorem 4 using the complexity result from [26]. In the general d-dimensional case these extra pointers are only used in trees associated with the final dimension.

THEOREM 8. *The improved $\mathcal{VO}$ for a d-dimensional range query q with T satisfying points is of size $O(\log^{d-1} N + T)$ and can be computed and verified in time $O(\log^{d-1} N + T)$.*

[8] We use almost the same structure Willard describes, except we keep the leaves in the associated trees in a singly linked list rather than a doubly linked list.

5. Three-Sided Range Queries. The two-dimensional range trees we discussed in the prior section have a few disadvantages: they use $\Theta(N \log N)$ space and they are not I/O efficient. These drawbacks can be addressed if we consider a more restricted problem. *Three-sided range queries* are defined by an x-range (x_l, x_u) and a one-sided y-range (y_l, ∞). Arge et al. present an I/O and space efficient indexing scheme for three-sided range queries in [2] that uses linear storage and $O(\log_B N + T/B)$ I/O operations to answer queries. We use the Search DAG model to construct a $\mathcal{VO}$ for this structure using constant additional storage and I/Os. We start with a high level overview of their scheme for the I/O efficient result and then look at the structure in more detail in order to decrease the $\mathcal{VO}$ size.

Arge et al. divide the N points into sets of size $\Theta(B^2)$ with each set stored in $\Theta(B)$ data blocks. As a result, the blocks can be indexed using $O(B)$ values. Thus, the blocks within a set which can contain answer points are determined with constant I/Os. We refer to their structure for storing $\Theta(B^2)$ points as an *auxiliary structure*.

A search tree, which we call the *base tree*, is used to determine which of the auxiliary structures need to be examined. The base tree is a balanced tree with branching factor $\Theta(B)$ and each node has an associated auxiliary structure. An internal node also has $O(B)$ key values which direct the search in the base tree. Arge et al. show that an initial search in the base tree can identify all nodes whose associated auxiliary structure might contain answer points using $O(\log_B N + T/B)$ I/O operations. The search uses structural data at the nodes to allow the search to terminate before reaching the base tree leaves.

Since the focus of this indexing scheme is efficient I/O operations, we would like to have a verification result stated in terms of I/Os as well. We can treat their data structure as a Search DAG. For each auxiliary structure there is a node of degree $\Theta(B)$ with the index information. Its successors are the $\Theta(B)$ data blocks, and these are the sink nodes. Each node in the base tree has its children as successors as well as the index node of its associated auxiliary structure. Since the data and hash values associated with each node in this Search DAG can be accessed with $O(1)I/O$s, our efficiency theorem applies:

THEOREM 9. *$\mathcal{VO}$s exist for three-sided queries which can be constructed with $O(\log_B N + T/B)$ I/Os.*

This approach has $\mathcal{VO}$s of size $\Theta(B \log_B N + T)$. Using properties of the auxiliary structure carefully to organize the search, we obtain a more efficient search and reduce the size and computational time of the $\mathcal{VO}$ to $O(\log N + T + B)$ while maintaining I/O efficiency. We also reduce the node degrees using the technique for B-trees presented in Section 4.1. The details of this and Theorem 9 are presented in the Appendix.

6. Conclusions and Future Work. In this paper we presented efficient methods to compute small, secure $\mathcal{VO}$s for a fairly broad class of data structures. The Search DAG model seems general enough to allow efficient $\mathcal{VO}$s to be computed for most data structures. One might hope, however, for an even more general method which would take logical constraints on the data structure and produce a search procedure and digest scheme.

One major issue which needs further work is updating the database. Our data publication schemes assume fairly static data sets, an assumption which is often reasonable but excludes a variety of data publication scenarios. Most of the data structures we discuss do admit efficient algorithms for updates. The conversion of a data structure to a Search DAG will often leave the update properties of the original data structure intact: it will be almost as easy to update the Search DAG and digest as the original data structure. This was true for most of our structures, but for Willard's improved scheme for multi-dimensional range trees, the Search DAG is harder to update (due to its chain of hash values). Similarly, the obvious Search DAG for skip lists is hard to update (though as shown in [12] this is easily fixed). Finally, authentic publication covers a broad range of secure query processing on the Internet—much work remains to handle other indexing structures, types of queries, other data models, and other trust models such as data integration from different owners.

Acknowledgments. The authors thank the anonymous reviewers for their many helpful suggestions and comments.

Appendix. Three-Sided Range Queries. In this section we describe the details of Theorem 9 followed by a more efficient Search DAG for three-sided queries. The I/O efficient result of Theorem 9 is a fairly straightforward application of the Search DAG model. However, the improved version relies on more details of the auxiliary structures. We reduce the size of the $\mathcal{VO}$ and construction time while maintaining the space and I/O efficiency. Our approach continues to build on the Arge et al. scheme.

Applying the Model for I/O Efficiency. We give more details of the Arge et al. data structures by first describing the base tree which is a multi-way tree of branching factor $\Theta(B)$ which divides the points by their x-coordinate. For simplicity we assume each internal node has exactly B children. Each internal node v has a set of points $P(v)$ associated with it (this is the set of nodes which will be in the auxiliary structures of v and all of v's descendants in the base tree). Arge et al. divide $P(v)$ into B equal size sets $S_1, S_2, \ldots, S_B$ by their x-coordinates. They then remove the B points in each set S_i with the largest y-value (the "highest" points). These B^2 points are put into the auxiliary structure associated with v. The remaining points are the sets associated with each child of v (so S_1 with its B highest points removed is the set associated with v's first child). Thus we can associate with each node v an x-range (its leftmost and rightmost point in $P(v)$) and y-height (its highest point).

From the above description it should be fairly clear how to search for points which satisfy a three-sided query (points (x, y) with $x_l \leq x \leq x_u$ and $y_l \leq y$). After visiting a base-tree node v and its associated auxiliary structure, we visit each child u of v such that (1) u's x-range overlaps (x_l, x_u) and (2) u's y-height is at least y_l. Any node which fails to satisfy (1) or (2) cannot have any answer points.

We can now describe the Search DAG in more detail. Each node v in the base tree is a node of the Search DAG, and the data associated with v is the $O(B)$ index values used to determine which of its $O(B)$ children to search (x-range and y-height values).

The successors of v are its children in the base tree and a single node whose data is the index information of the associated auxiliary structure.

The part of the Search DAG associated with an auxiliary structure will have a node containing the $O(B)$ index values and also a node for each of the $O(B)$ data blocks whose associated data is their $O(B)$ data points (more details on this in the next section). These $O(B)$ data block nodes are the successors of the "index" node and the data block nodes are sinks.

We search the base tree just as for the original data structure. For each auxiliary structure examined, we go to the index node and from its data determine which of its successors has the answer data. With these details in place we now give the proof of Theorem 9 (repeated from the main text):

THEOREM 9. *$\mathcal{VO}$s exist for three-sided queries which can be constructed with $O(\log_B N + T/B)$ I/Os.*

PROOF. Each node visited in the Search DAG corresponds directly to an I/O operation in the original data structure. We also store, with each non-sink node, the digest values of all its successors. Since each node has $O(B)$ data and successors, the $\mathcal{VO}$ data for each node can be stored in $O(1)$ blocks, and thus can be constructed and verified using asymptotically the same number of I/Os as nodes visited. □

A More Efficient Search and $\mathcal{VO}$. The Search DAG we described above is I/O efficient to construct, but for each node visited we have to put $\Theta(B)$ values into the $\mathcal{VO}$. Thus the $\mathcal{VO}$ may be as large as $\Theta(B \log_B N)$ even if there are no satisfying answer points. Since B will often be quite large, we want to reduce this overhead. To show how to do this we need to discuss the data structure in a little more detail.

First note that any time we get $\Theta(B)$ answer points from a visited node we are OK: we put $\Theta(B)$ values into the $\mathcal{VO}$, but we get $\Theta(B)$ answer points. Arge et al. show that we get $\Theta(B)$ amortized answer points for each node visited, except for what we call *fringe nodes* (see Figure 6). These are nodes in the base tree along the extreme right and left

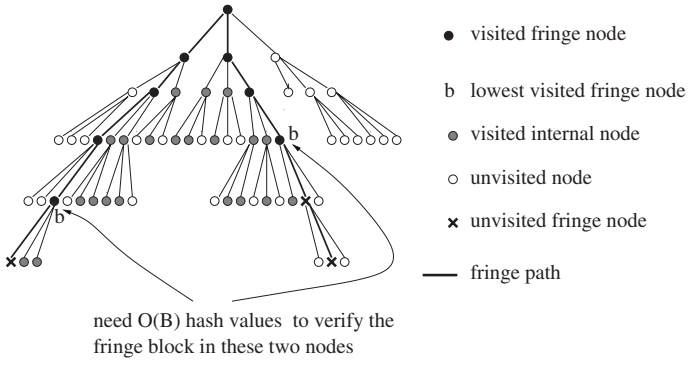

Fig. 6. Base tree and fringe paths.

path of nodes visited, and thus may only partially overlap the x-range.[9] Our improved Search DAG will reduce the size of the $\mathcal{VO}$ associated with these fringe node accesses.

One part is fairly easy. Since the base tree is basically a multi-way tree, we can convert each base tree node into a binary tree of height $O(\log_2 B)$ just as we did for B-trees. The "binary" base tree now has constant degree and the sequence of fringe nodes in it has length $O(\log_2 B) \times \log_B N = \log_2 N$.

Reducing the degree of the nodes for the auxiliary structures requires more detailed knowledge of their indexing scheme. For a given set of B^2 points to be stored in an auxiliary structure, we find B critical values $y_0 < y_1 < \cdots < y_{B-1}$ such that, associated with each y_i, $1 \leq i \leq B$, is a sequence Y_i of blocks of data (of the points in the auxiliary structure) ordered by their x-coordinates such that:

- The blocks in Y_i together contain each point (x, y) in the auxiliary structure with $y > y_{i-1}$.
- The blocks in Y_i have non-overlapping ranges of x-coordinates.
- If $y_{i-1} < y \leq y_i$, then any two consecutive blocks in Y_i have at least B points with height y or greater.

Note that the same point may have to be in more than one data block and the same data block may be used in more than one sequence Y_i. However, the total redundancy is only constant, so we still use only $O(N)$ space to store N points. Note also that the total number of blocks used is $O(B)$.

Each block of data points in an auxiliary structure has an associated x-range $x_{\min}, x_{\max}$ and y-range y_i, y_j. A query of the form $x_l \leq x \leq x_u$ and $y \geq y_l$, will access this block of points iff $y_i < y_l \leq y_j$ and the x-range overlaps $(x_{\min}, x_{\max})$. Note that the y-range indicates that the block is in sequences $Y_{i+1}, \ldots, Y_j$.

Since Arge et al. only worry about I/O performance, they find the correct data blocks by scanning all $\Theta(B)$ index values. We speed up searching the auxiliary index by using three levels of binary search trees (BSTs). The first level BST has the y_i values of the blocks associated with its leaves. Associated with each such leaf is the sequence of blocks Y_i associated with it.

The second level *sequence* BSTs, one for each sequence Y_i, have the $x_{\min}$ values of the blocks in the sequence associated with their leaves. They are used to search for the blocks in Y_i which also overlap the query x-range. The third level *point* BSTs, one for each block of points, are used to search for the points within a block. The leaves of this tree are the actual data points sorted by x-coordinate and these are the sinks of our Search DAG. Note that although there are logically $O(B^2)$ of these point trees, there are only $O(B)$ blocks of points in an auxiliary structure, so only this many distinct trees (and the overall structure is a DAG, not a tree). See Figure 7.

Thus when we access an auxiliary structure we first search in the top tree to find the smallest y_i, such that $y_l \leq y_i$ this gives us the root of the correct *sequence* tree which has all the blocks in Y_i. We now do a one-dimensional range search in the tree for Y_i to

[9] The amortized argument is that if a visited node is not a fringe node, then all its points satisfy the query's x-range, and it has at least one point above y_l. Thus all B of the "high" points removed from this node and put into its parent's auxiliary structure will be answer points and can be charged to this visit.

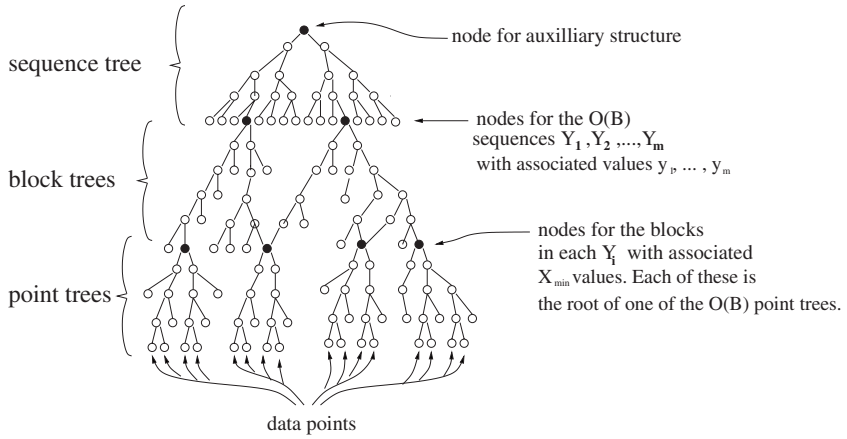

Fig. 7. Search DAG for an auxiliary structure (partially shown).

find the blocks which overlap the x-range of our query. Finally we get the actual points from each data block with another x-range search.

THEOREM 10. *We can answer a three-sided query using* $\Theta(\log_B N + T/B)$ *I/Os and build a* $\mathcal{VO}$ *of size* $\Theta(\log N + T + B)$ *using linear size data structures.*

PROOF. Recall that we reduce the degree of the base tree nodes by converting each such node into a binary tree. We can pack each of these binary trees as well as their digest values into $O(1)$ disk blocks. Similarly, each top level binary auxiliary structure tree and each sequence and point tree fit into $O(1)$ disk blocks (along with their digest values). Thus, we can simulate the search used by Arge et al. in our Search DAG with the same I/O performance.

For $\mathcal{VO}$ size we only need to analyze the fringe nodes, since each non-fringe node examined in the search corresponds to a contribution of $\Theta(B)$ points to the query answer, so non-fringe nodes contribute $O(T)$ in total to the $\mathcal{VO}$ size. As indicated above, we now use only an $O(\log\ N)$ size $\mathcal{VO}$ to describe the fringe path search in the base tree.

We now look at the $\mathcal{VO}$ size for the auxiliary structures associated with fringe nodes. A fringe node which is not the lowest node visited on the left or right fringe path has all of the points in its auxiliary structure above y_l. Thus, when we access the associated auxiliary structure for a fringe node, our search trees give us the same type of performance as for a one-dimensional range query on the x-range: $O(\log_2\ B + k)$ size $\mathcal{VO}$s where k is the number of points satisfying the x-range. Since there are $O(\log_B\ N)$ such "fringe" structures accessed, we get $O(\log\ N + T)$ total size. The final issue is to deal with a fringe node at the very bottom of the search which could have very few points above y_l (nodes labeled b in Figure 6). There are at most two such auxiliary structures accessed (one for the left/right path) and each may contribute $\Theta(B)$ size to the final $\mathcal{VO}$. □

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. *Persistent Authenticated Dictionaries and Their Applications*. Lecture Notes in Computer Science, 2200, pages 379-393, Springer-Verlag, Berlin, 2001.

[2] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the* 18*th Symposium on Principles of Database Systems*, pages 346-357, 1999.

[3] A. M. Ben-Amram, What is a "pointer machine"? *ACM SIGACT News*, 26(2):88–95, 1995.

[4] A. Buldas, P. Laud, and H. Lipmaa. Eliminating counterevidence with applications to accountable certificate management. *Journal of Computer Security*, 10:273–296, 2002.

[5] C. Jutia and M. Yung, Paytree: amortized Signature for flexible micropayments. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, pages 213–221, 1996.

[6] B. Chazelle and L. J. Guibas. Fractional Cascading: I. A Data Structuring Technique, *Algorithmica*, 1(2): 133–162, 1986.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer-Verlag, New York, 2000.

[8] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of XML documents. In *Proceedings of the* 8*th ACM Conference on Computer and Communications Security*, pages 136–145, ACM Press, New York 2001. To appear in the *Journal of Computer Security*

[9] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic Third-party Data Publication. *Journal of Computer Security*, 3(11):291–314, 2003.

[10] P. Devanbu and S. Stubblebine. Stack and queue integrity on hostile platforms. *IEEE Transactions on Software Engineering*, 26(2):100–108, 2000.

[11] M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, New York, 2001.

[12] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivablity Conference and Exposition* (*DISCEX II*), volume 2, pages 1068-1084, 2001.

[13] M. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen. Authenticated data structures for graph and geometric searching. *Topics in Cryptology - The Cryptographers' Track at the RSA Conference* 2003. Lecture Notes in Computer Science, 2612, pages 295–313, Springer-Verlag, Berlin, 2003.

[14] D. Gusfield. *Algorithms on Strings*, *Trees*, *and Sequences*. Cambridge University Press, Cambridge, 1997.

[15] S. Haber and W. S. Stornetta. How to timestamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[16] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. Vitter. Indexing for Data Models with Constraints and Classes. *Journal of Computer and System Sciences*, 52(3): 589–612, 1996.

[17] D. E. Knuth. *The Art of Computer Programming*, *Vol*. 1: *Fundamental Algorithms*. Addison Wesley, Reading, MA, 1968, 1973, 1997.

[18] A. N. Kolmogorov and V. A. Uspenskii. On the definition of an algorithm. *Uspekhi Matatematicheskikh*. *Nauk*, 13:3–28, 1958. English translation in *AMS Translations* II, 29:217–245.

[19] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the* 11*th USENIX Security Symposium*, pages 297-312, August 2002.

[20] R. C. Merkle. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122-134, 1980.

[21] R. C. Merkle. A certified digital signature. *Advances in Cryptology–Crypto* '89, Lecture Notes in Computer Science, 435, pages 218–238. Springer-Verlag, Berlin, 1990.

[22] M. Naor and K. Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

[23] W. Pugh. Skip Lists: a Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990

[24] A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490-508, 1980.

[25] K. V. Shvachko. Different modifications of pointer machines and their computational power. In *Proceedings of the Symposium on Mathematical Foundation of Computer Science*, pages 426-435, 1991.

[26] D. E. Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing*, 14:232–253, 1985.