A General Model for Authenticated Data Structures

Chip Martel[*], Glen Nuckolls, Prem Devanbu, Michael Gertz, April Kwong
Department of Computer Science
University of California
Davis, CA 95616 USA
{martel|nuckolls|devanbu|gertz|kwonga}@cs.ucdavis.edu

Stuart G. Stubblebine
Stubblebine Research Labs
stuart@stubblebine.com

Technical Report CSE-2001-9
UC Davis Department of Computer Science
December 6, 2001

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 is a novel approach which allows *untrusted* publishers to securely answer queries from clients on behalf of trusted off-line data owners. Publishers validate answers using compact, hard-to-forge *verification objects* ($\mathcal{VO}$s), which clients can check efficiently. This approach provides greater scalability (by adding more publishers) and better security (on-line publishers don't need to be trusted).

To make authentic publication attractive, it is important for the $\mathcal{VO}$s to be small, efficiently computable and verifiable. This has led researchers to develop several different data representations for efficient $\mathcal{VO}$ computation. In this paper, we develop a new data model called *Search DAGs*. Within this model, we develop a generalized algorithm for the construction of $\mathcal{VO}$s. We show that the $\mathcal{VO}$s thus constructed are *secure* and *compact*. In addition, they are *efficient* to compute and verify. We demonstrate how this approach 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, compactness and efficiency results immediately follow from our general theorems concerning Search DAGs. Going futher, we also use Search DAGs to prove the security of two complex data models for efficient multi-dimensional range searches. This allows compact $\mathcal{VO}$s to be computed (size $O(\log N + T)$) for typical 1D and 2D 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 1D and 3-sided range queries and compute the $\mathcal{VO}$s with $O(\log_B N + T/B)$ I/O operations using linear size data structures.

This work was supported by NSF grant CCR 85961

[*]Contact author, phone +1 530 752 2651, fax +1 530 752 4767, email martel@cs.ucdavis.edu

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 online 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 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.

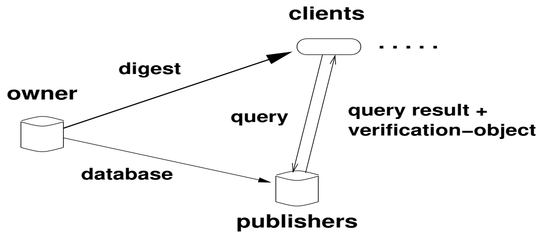

Figure 1: Authentic Publication Scheme

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 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 privledged 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 *compact*, 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 search trees [13, 14], and recently, skip-lists [4, 5]. 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 2D-range trees for this in [8], which, like all prior work in this area, used a binary search tree based 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 datatbases), 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. Thus we introduce 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 the traditional authentic data models 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 then use our security theorem to prove the security of authenticated versions of two more complex data structures. We briefly summarize the results below.

Main Results. There are three main results.

1. General model for efficient, authenticated data structures. We introduce the Search DAG model, which characterizes a broad class of data structures. We then describe a general method to create efficient authenticated versions of any data structure in this class. We prove that any data model 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, and skip lists can be placed conveniently in the Search DAG framework.

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 develop improved authenticated data strutures for the important class of multi-dimensional range queries which request all points in a d-dimensional rectangle.[1] 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 [12]. Answers to 2D range and 3-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, we use Search DAGs to answer d-dimensional range queries with a $\mathcal{VO}$ of size and construction time $O(\log^{d-1} N + T)$. For 3-sided range queries, we use Search DAGs to create an authenticated version of the complex data structure described in [1] to get $\mathcal{VO}$s of size $O(\log N + B + T)$ (where B is the disk block size). The $\mathcal{VO}$ for 3-sided queries can be computed using only $O(\log_B N + T/B)$ I/O operations and using linear size data structures.

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 of the original (non-authentic) data structures, and produce compact $\mathcal{VO}$s. Security results for the $\mathcal{VO}$s also immediately follow from the general security theorem. These results lead us to believe 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 [11] and micro-payments [17]. The proposed techniques are based on the original work by Merkle [13] and refinements by Naor and Nissim [14] for certificate revocation. Recently, Goodrich and Tomassia showed that skip lists provide compact and simple $\mathcal{VO}$s for these settings [4, 5]. Devanbu and Stubblebine showed how to create authenticated versions of stacks and queues [7]. In [8], we introduced the general idea of authentic data publication, and we showed how to securely answer range queries. This work was extended to authentic publication of XML documents in [9] 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 can use explicit rather than implicit constraint information. This allows us to create authentic versions of more complex data structures. We extend and improve these prior results, and we provide a general framework for the computation of

[1] An example 2D-range query is: return all points (x, y) such that $10 < x < 15$ and $30 < y < 50$.

compact $\mathcal{VO}$s based on different data structures. Necula [15] describes an alternate approach using proof-carrying code, which bundles programs and logical proofs of their relevant properties. Note that a similar logic-based approach (proof-carrying answers) in our case would lead in general to impractical proofs which would be as large as the databases themselves. Instead, we use an approach that relies on cryptography.

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 efficiently compute compact $\mathcal{VO}$s for 1D range queries. In Section 3, we describe the search DAG model, and prove both its security and efficiency. In Section 4 we show that several existing secure structures can be easily modeled as search DAGS, and we then apply search DAGs to B-trees and tries. In Section 5, we extend this framework to multi-dimensional range queries, focusing on the general case as well as specialized queries (3-sided queries) and efficient computation schemes (fractional cascading). We then summarize our results and outline future work.

2 Background: 1-Dimensional Range Queries

We outline the principles of our data publication scheme and illustrate their use for membership and 1D range queries. We show the traditional approach to this setting and our new scheme. Authentic publication protocols involve three parties: the *owner*, who creates and is responsible for the content [2]; the *publisher*, who handles on-line query processing, and *client*, who submits queries to the publisher. The *client* relies on the *owner* to create accurate data, but does not trust the *publisher*. The authenticity of our schemes rely primarily on *collision-intractable hash functions* (CIHF) . Our protocols typically involve several steps. First, the *owner* computes a digest d of the content using a CIHF over a data structure containing all the data. This 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 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*, *publisher* and *owner* is **secure** if, given a digest (computed by *owner*), a $\mathcal{VO}$ and an answer computed by *publisher*, the *client* will only recompute the digest when the answer is just what the *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 $x_1 < \ldots < x_n$. 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 $h(x_i)$ ($h1 \ldots h4$, as shown in Figure 2A). Each internal node's value is the hash of the values of its children. This construction, due to R. Merkle [13], is called a Merkle hash tree and it has been used by several authors to solve problems related to authentic publication [14, 11, 8]. The root digest value h^* 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 2A, the publisher can prove that 23 is in the data set by providing as a $\mathcal{VO}$: 23, 45, $h34$. We proceed botom-up. Using the first two values the 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 better illustrate our general data model, 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

[2] there might instead be a *certifier* who is responsible for assuring the correctness of the data and distributing a digest

(see figure 2B). 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 h^*, 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 standard Merkle tree as in Figure 2A).

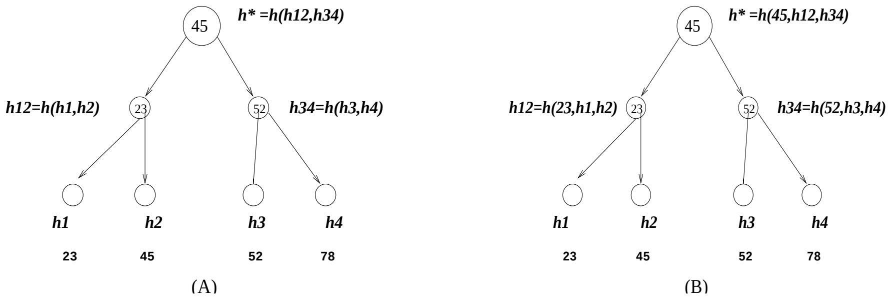

Figure 2: Computation of digest values (A) and including split values (B)

Our new approach may be less efficient (since we hash three values instead of two) but it gives additional flexibility which we will 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.

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 $a(s)$, 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 will 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 2nd 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 D. An example is the binary search tree 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 authenitcated 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:

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

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 $f(s)$ (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 will denote $\mathcal{VO}(q)$. Let $v_1(= s), v_2, \ldots, v_n$ be the nodes visited when P is run with input q on the *owner*'s DAG. We also let $u^i_1, \ldots, u^i_{k_i}$ be the successors of v_i. The correct $\mathcal{VO}$ for q is then the following values:

$a(s), f(u^1_1), \ldots, f(u^1_{k_1});$
$a(v_2), f(u^2_1)), \ldots, f(u^2_{k_2});$
$\ldots\ ;$
$a(v_n), f(u^n_1), \ldots, f(u^n_{k_n});$

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 to $f(s)$. 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 to $f(u_k^1)$. When it matches, we input $a(v_2)$ to P (otherwise we would reject). P then outputs (j, k) to indicate the next node visited (with $j = 1$ or 2). We hash step three and compare the result to $f(u_k^j)$, and when they match we input $a(v_3)$ to P, and so on. After inputting $a(v_n)$, 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:

$x_1, y_1^1, y_2^1, \dots y_{k_1}^1;$
$x_2, y_1^2, y_2^2, \dots y_{k_2}^2;$
$\dots;$
$x_n, y_1^n, y_2^n, \dots y_{k_n}^n$

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 to 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 you 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 fooling our verification, but it is often not sufficient. For example, even if an attacker uses an alternate step $\tilde{x}_i, \tilde{y}_1^i, \tilde{y}_2^i, \dots \tilde{y}_{k_i}^i$ which has the same hash value as the correct step $x_i, y_1^i, y_2^i, \dots y_{k_i}^i$, 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 $\mathcal{VO}$ $V \neq \mathcal{VO}(q)$, if V is not a forgery of $\mathcal{VO}(q)$, then $V_P(q)$ rejects V.

Proof :

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

If $n = 1$,
$V_P(q)$ 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 $n-1$ steps of $V_P(q)$ are correct (with $n \geq 2$). We show that $V_P(q)$ rejects after processing step n unless $\overline{s_n}$ is correct.

Since the first $n-1$ steps are correct, P has been given the correct first $n-1$ data values, and thus its output (which we denote (j,k)) after step $n-1$ is the correct next node to visit. $V_P(q)$ will next compare the hash of step n with y_k^j. Since $j < n$ we know y_k^j is the correct value to compare to (again by the induction hypothesis), and since V is not a forgery of $\mathcal{VO}(q)$, the hash of step n will only match if the vector of values is the same as in step n of $\mathcal{VO}(q)$. Thus we reject unless step n is correct. □

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 $N(q)$ be the number of nodes visited in the search which answers q and $T(q)$ be the time taken by P to process q before it starts the search.

Theorem 4 Consider a Search DAG where G has bounded degree, the data values $a(v)$ associated with the nodes are of bounded size, and P can process a data value $a(v)$ in $O(1)$ time. Then for any query q, we can build $\mathcal{VO}$s of size $O(N(q))$ which can be constructed and verified in time $O(N(q)+T(q))$.

Proof: The $\mathcal{VO}$ for q has $N(q)$ 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. □

We note the above boundedness assumptions apply to many normal search procedures such as binary search trees, multi-dimensional range trees, and skip-lists. In the next section we show that it is easy to cast these in our model and thus get good $\mathcal{VO}$ schemes. However, the main advantages of our model is for the more complex data structures we deal with in the final sections.

Our general method may lead to some inefficiencies, 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. We will also describe a transformation which deals with high degree nodes.

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 Search DAGS. 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 compact $\mathcal{VO}$ for a 1D 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 [14]).

Skip lists [16] can provide an attractive alternative to trees, and Goodrich and Tamassia recently showed how to create efficient $\mathcal{VO}$s for skip list answers [4, 5]: $O(\log n)$ 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 and Tamassia 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

In this section we illustrate a way to reduce node-degrees which will also be used with some more complex data structures.

Binary search trees (BST) 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 is a BST, so we get smaller $\mathcal{VO}$s for both membership and 1D 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 asymptotic I/O operations as for the B-tree.

This demonstrates a general method 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 1D range query on N data points using disk blocks of size B. It follows from theorems 3 and 4 that:

Theorem 5 For a 1D range query with T answer points we get a (binary) $\mathcal{VO}$ 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 [3] and its special form as a suffix tree are used in many applications [10] including our recent use of tries to authenticate XML documents [9]. As a tree-like structue, 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.

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 occurances in $O(m+k)$ time. They have many uses in string matching in general and computational biology in particular [10]. Since a suffix tree is a trie, we want to 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. With some care it is possible to 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 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 nodes in the subtree rooted at v correspond to all starting points of P in S (see figure 3). The space can be 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.

We would like to use the search DAG model to provide authentication for suffix trees. The simplest way is to use the $O(n^2)$ space version of the suffix tree which stores the entire substring corresponding to each edge of the tree. In this case, our logical search DAG is just the tree with each edge transformed into a chain of nodes, one for each character in the string (see figure 3).

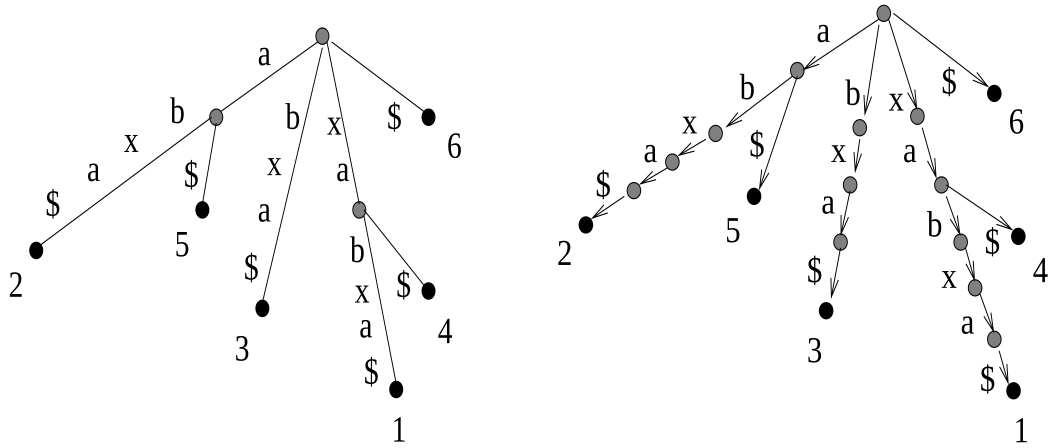

Figure 3: Suffix tree for $S =$ xabxa\$ and resulting naive Search DAG

For a fixed alphabet, the degree of the suffix tree and thus the resulting search DAG is constant. The client is prefectly 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* might choose to simply store the compacted suffix tree and use this to answer queries and construct $\mathcal{VO}$s. However, contructing the $\mathcal{VO}$ now poses a problem when the query produces a string which terminates early in an edge substring. In this case, the *publisher* must provide an intermediate supporting value in order to keep the $\mathcal{VO}$ size optimal. There are as many of these as the length of the substring, so that the *publisher* must either calculate these values each time, or store them using $O(n^2)$ space, defeating the point of using the compacted tree. These difficulties can be overcome using a better Search DAG.

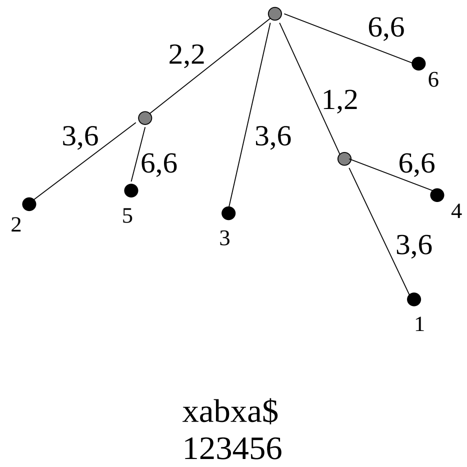

Figure 4: Compacted version of the suffix tree in figure 3

A compacted suffix tree consists of an edge labled 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 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 edges directed away from the root.
2. Another n nodes, one for each character/position of the original string S. The *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).

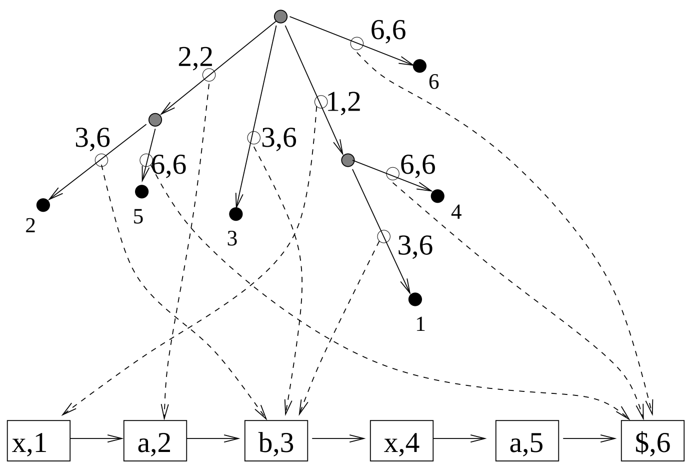

Figure 5: Search DAG for the compressed suffix tree. Pointers to string nodes are shown from the corresponding index data for clarity.

We search this structure as follows: we start by finding the outgoing edge from the root which starts with the first character of P. This gets us a pointer to the position i in S as well as the length l of the edge label. Thus we check $S[i],\ S[i+1], \ldots, S[i+l-1]$ stopping if P is used up (a match is found), we get a mismatch (P is not in S), or, if all of the characters match and there is more of P, we continue on the next edge in the same manner.

It is straightforward to check that this is a DAG. To see that it has bounded out-degree, note that we assume the alphabet is bounded and each edge from a single internal node in the suffix tree starts with a different character.

Since there is a constant amount of data associated with each node or edge of the original graph, it is clear that there is a constant amount of associated data at each node (which is processed by a search in constant time).

We can now apply our result for Search DAGs to get the following result:

Theorem 6 For a pattern P of length m in a string S of length n with k occurences of P in S, we can compute $\mathcal{VO}$s of size $O(m+k)$ which can be constructed and verified in the same time.

5 Multi-dimensional Range Queries

In this section, we introduce efficient authenticated schemes for multi-dimensional range queries. The general case using range trees is outlined in Section 5.1 as a simple introduction. In Section 5.2, we discuss an improved but more complex computation scheme using fractional cascading. $\mathcal{VO}$s for 3-sided queries, a special and important type of range queries, are detailed in Section 5.3. In all cases we get efficient authenticated structures using Search DAGs.

5.1 Multi-dimensional Range Trees

For a database of N points, d-dimensional rectangular queries of the form $(x_{l_1}, x_{u_1}), (x_{l_2}, x_{u_2}), \ldots, (x_{l_d}, x_{u_d})$, ask for all points $(x_1, x_2, \ldots, x_d)$ such that $x_{l_i} \leq x_i \leq x_{u_i},\ 1 \leq i \leq d$. These points can be efficiently found using *multidimensional range trees (mdrts)* [2] in time $O(\log^d N + T)$. In this section, we show that mdrts can be used to efficiently compute $\mathcal{VO}$s for rectangular queries as well. In the following, we first outline the basic characteristics of mdrts and then show how to convert them to a Search DAG to get compact $\mathcal{VO}$s. We discuss 2D queries, but the framework extends to higher dimensions. These basic results on mdrts largely duplicate our results in [8], but we discuss the details since we build on them later in our improved results.

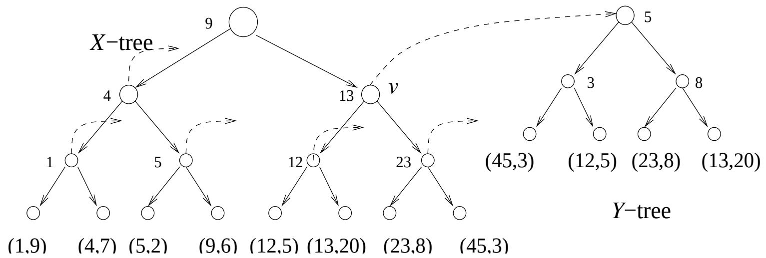

Figure 6: Example of a 2D Range Tree

Consider the example *mdrt* shown in Figure 6, consisting of one X-tree, and multiple Y-trees which represents points in the 2D space. The X-tree sorts the points by x coordinate. Each interior node in the X-tree is an ancestor of a set of points which are also represented in a Y-tree. Consider the interior node v in the X-tree of figure 6, which is the ancestor of points with coordinates (12,5), (13,20), (23,8) and (45,3). An mdrt contains a link from v to the root of an *associated Y-tree*, denoted as $Y(v)$. This Y-tree contains the same four points; however, in this tree, they are sorted by the y coordinate. Likewise each interior node v_i in the X-tree has a pointer to an associated binary search tree $Y(v_i)$ for the points in the subtree below v_i. For higher dimension *mdrt*'s, each node v of a $j+1$-dimensional *mdrt* contains a pointer to a j-dimensional *mdrt*. The nodes of the final 1D-tree (Y-tree(s) above) do not have such pointers.

Consider a 2D rectangular query of the form $(x_l, x_u), (y_l, y_u)$. To answer this query, we first define a node v in the X-tree as a *Canonical Covering Root (CCR)* if all descendents of v satisfy the x-range of the query, but not all the descendents of $v's$ parent do. There are $O(\log N)$ CCRs, which can be found in $O(\log N)$ time [2]. For each CCR, the corresponding Y-tree is searched for the requisite range, thus yielding an $O(\log^2 N + T)$ time for 2D range searches (recall T is the number of answer points). In general, d-dimensional range queries can be computed in time $O(\log^d N + T)$. Range trees require $O(N \log^{d-1} N)$ storage space, and can be constructed in time $O(N \log^{d-1} N)$.

To convert an mdrt to a Search DAG, the nodes and arcs of the DAG are exactly as in the mdrt. Each internal node has its split value (in the appropriate dimension as in Figures 2 and 6) as its associated data values. A leaf has the (x, y) coordinates of its associated point. This is sufficient to guide a search which finds the CCR roots and then searches for the appropriate leaves in the associated Y trees. A trace of this search is the $\mathcal{VO}$. Each node has degree at most three and a bounded amount of data. So by Theorem 4 we immediately get a $\mathcal{VO}$ of size and construction time equal to the number of nodes visited: $O(\log^2 N + T)$.

The above 2D construction extends easily to a d-dimension *mdrt* (e.g. for 3 dimensions, each node of the Y-tree points to a Z-tree sorted on the z-coordinate).

Theorem 7 We can verify a d-dimensional range query using a $\mathcal{VO}$ of size $O(\log^d N + T)$ which can be computed in the same time.

5.2 Improved Multi-dimensional Range Queries: Fractional Cascading

We apply our Search DAG model to a more efficient range query structure. *Fractional cascading* [2, 6] reduces the search time for a d-dimensional query by a factor of $O(\log N)$ to $O(\log^{d-1} N + T)$. We review fractional cascading and use the Search DAG model to construct $O(\log N)$ size $\mathcal{VO}$s for 2D range queries.

With 2D mdrts, we first find $O(\log N)$ x-dimension CCRs and then, for each CCR, we search the associated Y-tree. Each Y-search can produce a $\Theta(\log N)$ piece of the $\mathcal{VO}$. In fractional cascading each node v in the X-tree has a pointer to an associated array denoted $Y(v)$ containing all points in the subtree rooted at v. These points are sorted by their y-coordinate and the array also contains pointers to help speed up the search. Let $Y(v)[i].y$ denote the y coordinate of the ith element of $Y(v)$. We illustrate the setting in figure 7 where w, a are the left/right children of v in the X-tree. An element $Y(v)[j]$ has a pointer to $Y(w)[i]$, where i is the smallest index such that $Y(w)[i].y \geq Y(v)[j].y$. $Y(v)[j]$ also points to the analogous entry $Y(a)[m]$ (see Figure 7 where the pointers from $Y(v)[2]$ (which has value 55) go to $Y(w)[1]$ $(= 55)$ and $Y(a)[2]$ $(= 60)$ illustrate this).

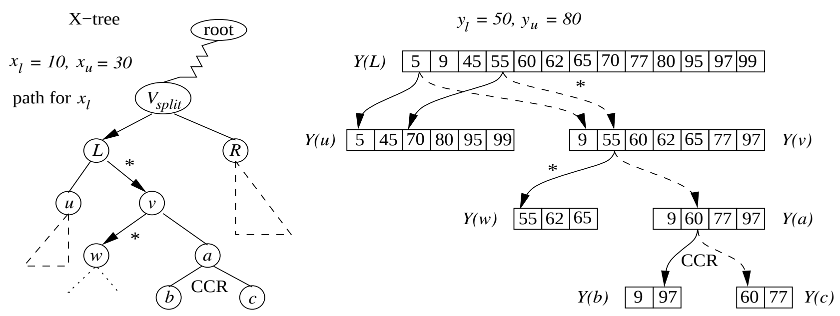

Figure 7: Fractional Cascading Scheme (only y-coordinates and select pointers are shown)

For a 2D query $(x_l, x_u), (y_l, y_u)$, we first do a 1D range search for (x_l, x_u) in the X-tree, to find V_{split}, the first node with answer points under both its children, and the Cannonical Covering Roots (CCR) as in Figure 7. Starred edges represent arcs on a search for x_l, so all vertices in the subtree rooted at a satisfy the x-range, so vertex a is a CCR.

If V_{split} is a leaf, it is the answer. Otherwise let L and R be the children of V_{split} as in the figure. $Y(L)$ now has all the points in the subtree rooted at L; we seek the smallest j such that $Y(L)[j].y \geq y_l$ ($j = 4$ and $Y(L)[4] = 55$ in Figure 7). The pointers associated with $Y(L)[j]$ directly indicate candidate Y-range points in the arrays associated with CCRs (in our example, we follow the pointers from 55 in $Y(L)$ to 55 in $Y(v)$ and then to 60 in $Y(a)$). Once we hit the array associated with a CCR, we simply scan until we hit an out of range element (e.g., we collect 60 and 77 from $Y(a)$ then hit 97 which is too large). The cascading pointers allow scans rather than searches, and provide the additional efficiency over mdrts. The search in the X-tree tells us which nodes s under L are roots of CCRs, so we do these scans on the corresponding $Y(s)$ arrays.

The Search DAG Our DAG builds on the mdrt Search DAG of the previous section. The X-tree and associated Y-tree's are the same, except that the leaves of a Y-tree now only hold y coordinates and are no longer sinks. The nodes, which were leaves of a tree $Y(v)$ in the regular mdrt, will now have pointers to the corresponding elements of the fractional cascading array for $Y(v)$. Just as for the mdrt, we identify the CCR's in the X-tree and the split node V_{split}. A search in the Y-trees associated with $V'_{split}s$ children

(L, R in our example) lets us find the correct places to begin our fractional cascade search (value 55 in the prior example).

Once the smallest y value at least as large as y_l is found, we begin to traverse the arrays corresponding to nodes in the X-tree which are CCRs. At each array corresponding to a CCR, we search to the right until we encounter a y-value outside of the query range. Although the physical implementation of the fractional cascading algorithm might suggest that the nodes at each array have three successors: one to the element to its right and two to its "child" arrays, the logical view indicates that this is not quite the case. When searching the array to the right, the pointers to the lower arrays are ignored. Thus, for the purposes of a sequential array search, there is exactly one successor for each node except the right end sink node. This tells us that in the Search DAG, the nodes for the elements of the arrays as used for a sequential search can be distinct from the nodes reached via pointers from higher arrays [3]. This suggests simply keeping two arrays, each supporting one of the two functions of the array in the search (see figure 8). We refer to the array with data used for the sequential search as $Y_S(v)$ and the array with the child pointers as $Y_T(v)$, where v is the associated node in the X tree. We note that the array elements here are actually just nodes in the Search DAG. The associated data field at nodes of $Y_T(v)$ will be empty in our case since the search procedure used the information from the X tree search to determine which of the successor arrays to follow when a node of $Y_T(v)$ is reached. The Y_S arrays are searched when these are for CCR nodes in the X-tree. Finally, to convert this to a DAG, each array element becomes a node. Y_S nodes have data values and one outgoing pointer to its right neighbor. Y_T nodes have no data, only pointers to their two children and to their corresponding Y_S node.

The search in these structures proceeds just as in normal fractional cascading: search the X-tree to find v_{split} and CCRs, go to v_{split}'s Y tree to find the correct position to start the fractional cascade, then follow pointers to find the starting point for a linear search of answer points for each CCR.

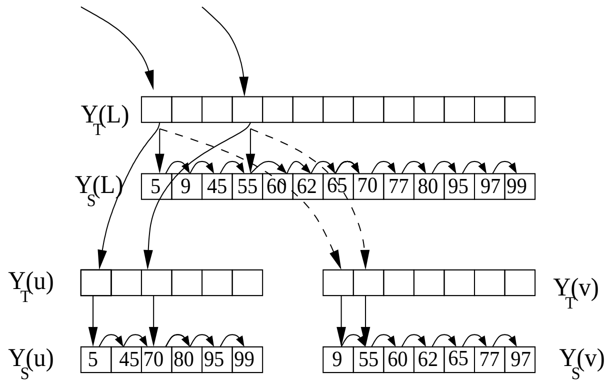

Figure 8: Search DAG for the Fractional Cascading Scheme

Digesting Fractional Cascades

Although our model tells us that the digest values are simply computed from the sinks of our search DAG up to the source, we now give a simple description of the function for this particular search DAG.

The structure of each Y_S array tells us to hash values right-to-left. Once each of the values for the Y_S arrays are computed, the Y_T array values are just the hash of the values of their three successors. Each entire Y_T array is digested using the binary Y-tree associated with it, and then the X-tree is digested in the usual way with the addition of the digest of the associated Y structure at each node. This process can be accomplished at construction time with constant overhead.

The above 2D construction easily generalizes to higher dimensions (note only the final dimension trees

[3] We could use a single array, but this would produce a different DAG and more complicated digest function.

use fractional cascades). Since we get a constant degree Search DAG for fractional cascading, we get immediately from theorem 4:

Theorem 8 The fractional cascading $\mathcal{VO}$ 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 $O(\log^{d-1} N + T)$.

5.3 3-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 (this applies with and without fractional cascading). These drawbacks can be addressed if we consider a more restricted problem. *3-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 3-sided range queries in [1]. They use 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 structure which allows us to create an I/O efficient authenticated structure. We then look at the structure in more detail 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 is determined with constant I/Os. We refer to their structure for storing $\theta(B^2)$ points as an *ASV structure*.

A search tree, which we will call the *base tree*, is used to determine which of the ASV structures need to be examined. The base tree is a balanced tree with branching factor $\theta(B)$ and each node has an associated ASV 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 ASV structure might contain answer points using $O(\log_B N + T/B)$ I/O operations. The search uses structural data at the nodes to terminate the search 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 ASV 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 ASV structure. Since the data and hash values associated with each node in this Search DAG can be accessed with O(1) I/Os, our efficiency theorem applies:

Theorem 9 $\mathcal{VO}$s exist for 3-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 ASV structure to carefully 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 compact, 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 fractional cascading 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 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.

References

[1] L. Arge, V. Samoladas, and J.S. Vitter. On Two-Dimensional Indexability and Optimal Range Search Indexing In *Proc. of the 18th Symposium on Principles of Database Systems (PODS '99)*, 346-357, 1999.

[2] M. D. Berg , M. V. Kreveld, M. Overmars and O. Schwarzkopf. *Computational Geometry.* Springer, New York, 2000.

[3] M. Goodrich , R. Tamassia. *Data Structures and Algorithms in Java* John Wiley and Sons Inc., 2001.

[4] M. Goodrich , R. Tamassia. Efficient Authenticated Dictionaries with Skip Lists and Commutative Hashing, Preprint, 2001

[5] M. Goodrich, R. Tamassia, and A. Schwerin. Implementations of an Authenticated Dictionary with Skip Lists and Commutative Hashing. To appear in *DISCEX II, 2001*

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

[7] P. Devanbu, S. Stubblebine. Stack and Queue Integrity on Hostile Platforms, IEEE Transactions on Software Engineering 26(2), 2000.

[8] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic Third-party Data Publication, 14th IFIP 11.3 Working Conference in Database Security (DBSec 2000), 2000.

[9] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible Authentication of XML documents. In the 8th ACM Conference on Computer and Communications Security, 2001.

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

[11] S. Haber and W. S. Stornetta. How to timestamp a digital document *J. of Cryptology*, 3(2), 1991.

[12] 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), pp. 589–612, 1996.

[13] R.C. Merkle. A certified digital signature. In *Advances in Cryptology–Crypto '89*, Lecture Notes in Computer Science, Vol. 435, 218–238, Springer, 1990.

[14] M. Naor, K. Nissim. Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[15] G. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 106-119, 1997.

[16] W. Pugh. Skip Lists: a Probabilistic Alternative to Balanced Trees *CACM* 33(6): 668–676, 1990.

[17] S. Charanjit and M. Yung. Paytree: Amortized Signature for flexible micropayments, *Second Usenix Workshop on Electronic Commerce Proceedings*, 1996

[18] J. D. Tygar. Open Problems In Electronic Commerce In *Proceedings of the 18th Symposium on Principles of Database Systems (PODS '99)*, 101, 1999.

7 Appendix: Three-Sided Range Queries

In this section we describe the details of theorem 9 followed by a more efficient Search DAG for 3-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 ASV 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, Samoladas, Vitter scheme.

Applying the Model for I/O Efficiency

We now describe the Search DAG for achieving I/O efficient $\mathcal{VO}$s in more detail than in the main paper. To do this we must first describe more details of the Arge et. al. data structures.

We start by 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 ASV structures of v and all of $v's$ descendents in the base tree). We divide $P(v)$ into B equal size sets $S_1, S_2, \ldots, S_B$ by their x-coordinates. We 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 ASV 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 wtih $v's$ 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 heighest point).

From the above description it should be fairly clear how to search for points which satisfy a 3-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 ASV 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 ASV structure.

The part of the search DAG associated with an ASV 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.

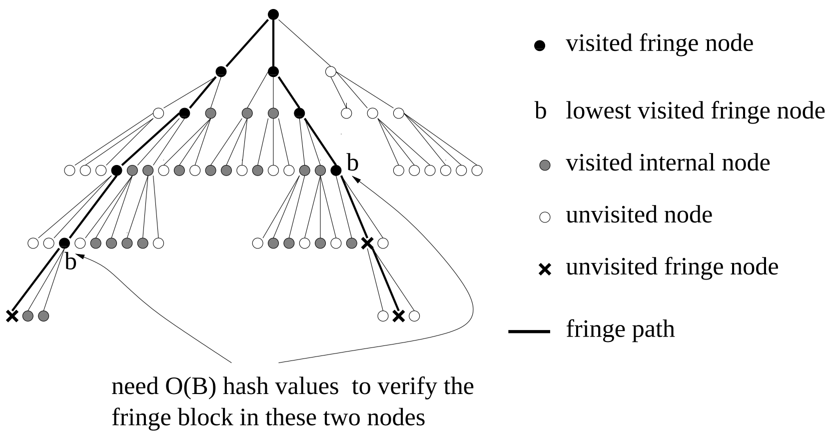

Figure 9: Base tree and fringe paths

We search the base tree just as for the original data structure. For each ASV 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 (repeated from the main text):

Theorem 9 $\mathcal{VO}$s exist for 3-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 assymptotically 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 will need to discuss the data structure in a bit 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 you get $\Theta(B)$ amortized answer points for each node visited, except for what we call *fringe nodes* (see figure 9). These are nodes in the base tree along the extreme right and and left path of nodes visited, and thus may only partially overlap the x-range. [4] 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 ASV structures requires more detailed knowledge of their indexing scheme. For a given set of B^2 points to be stored in an ASV structure, they find $B+1$ critical values $y_0 < y_1 < \ldots < y_B$ such that they can associate with each $y_i,\ i > 0$ a sequence Y_i of blocks of data (of the points in the ASV structure) ordered by their x coordinates such that:

- Y_i contains each point (x, y) in the ASV structure with $y \geq y_i$
- 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 redundency 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 ASV structure has an associated x-range x_{min}, x_{max} and y-range y_{i-1}, y_i. A query of the form: $x_l \leq x \leq x_u$ and $y \geq y_l$, will access this block of points iff $y_{i-1} < y_l \leq y_i$ and the X-range overlaps (x_{min}, x_{max}). Note that the y-range is really saying that the block is in sequence Y_i.

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 ASV index by using three levels of binary search trees (BST). 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.

[4] The amortized argument is that if a visited node isn't a fringe node, 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 ASV structure will be answer points and can be charged to this visit.

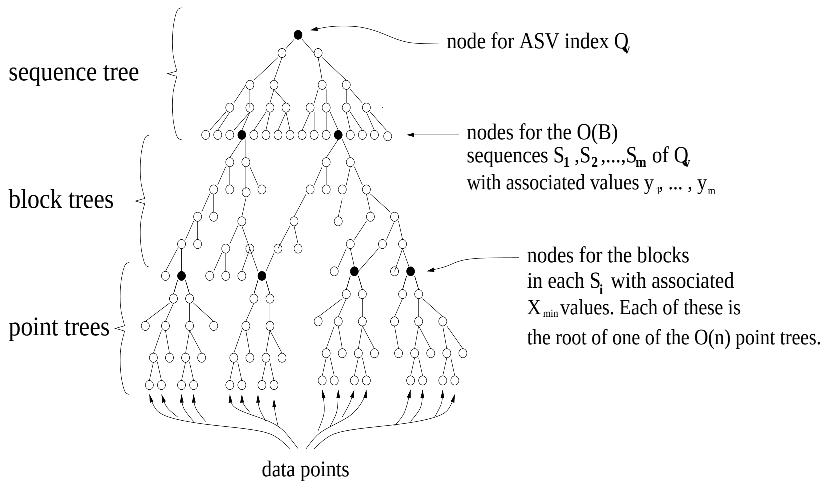

Figure 10: Search DAG for an ASV structure (partially shown)

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 ASV structure, so only this many distinct trees (and the overall structure is a DAG, not a tree). See figure 10.

Thus when we access an ASV 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 1D range search in the tree for Y_i to 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 3-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 ASV 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 of for the AVS 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 ASV structure above y_l. Thus, when we access the associated ASV structure for a fringe node, our search trees give us the same type of performance as for a 1D 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 9). There are at most two such ASV structures accessed (one for the left/right path) and each may contribute $\Theta(B)$ size to the final $\mathcal{VO}$. □