Authenticated Data Structures, Generically

Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi

University of Maryland, College Park, USA

Abstract

An authenticated data structure (ADS) is a data structure whose operations can be carried out by an untrusted *prover*, the results of which a *verifier* can efficiently check as authentic. This is done by having the prover produce a compact proof that the verifier can check along with each query result. ADSs thus support outsourcing data maintenance and processing tasks to untrusted servers without loss of integrity. Past work on ADSs has focused on particular data structures (or limited classes of data structures), one at a time, often with support only for particular operations. This paper presents a generic method, using a simple extension to a ML-like functional programming language we call $\lambda\bullet$ (lambda-auth), with which one can program authenticated operations over any data structure constructed from standard type constructors, including recursive types, sums, and products. The programmer writes the data structure largely as usual; it can then be compiled to code to be run by the prover and verifier. Using a formalization of $\lambda\bullet$ we prove that all well-typed $\lambda\bullet$ programs result in code that is secure under the standard cryptographic assumption of collision-resistant hash functions. We have implemented our approach as an extension to the OCaml compiler, and have used it to produce authenticated versions of many interesting data structures including binary search trees, red-black trees, skip lists, and more. Performance experiments show that our approach is efficient, giving up little compared to the hand-optimized data structures developed previously.

1. Introduction

Consider a data provider who would like to allow third parties to mirror its data, providing a query interface over it to clients. The data provider wants to assure clients that mirrors will answer queries over the data truthfully, even if the mirrors (or another party that compromises the mirror) have an incentive to lie. As examples, the data provider might be providing stock market data, a certificate revocation list, the Tor relay list, or the state of the current Bitcoin ledger [18].

Such a scenario can be supported using *authenticated data structures* (ADS) [20, 4, 27]. ADS computations involve two roles, the *prover* and the *verifier*. The mirror plays the role of the prover, storing the data of interest and answering queries about it. The client plays the role of the verifier, posing queries to the prover and verifying that the returned results are authentic. At any point in time, the verifier holds only a short *digest* that can be viewed as summarizing the current contents of the data; an authentic copy of the digest is provided by the data owner. When the verifier sends the prover a query, the prover computes the result and returns it along with a *proof* that the returned result is correct; both the proof and the time to produce it are linear in the time to compute the query result. The verifier can attempt to verify the proof using its current digest (taking time linear in the size of the proof), and will accept the returned result only if the proof verifies. The data owner may also have the capability to update its data stored at the prover; in this case, the result includes an updated digest and the proof shows that this updated digest was computed correctly. ADS computations have two properties. *Correctness* implies that when both parties execute the protocol correctly, the proofs given by the prover verify correctly and the verifier always receives the correct result. *Security*[1] implies that a computationally bounded, malicious prover cannot fool the verifier into accepting an incorrect result.

Authenticated data structures can be traced back to Merkle [15]; the well-known *Merkle hash tree* can be viewed as providing an authenticated version of a bounded-length array. More recently, authenticated versions of data structures as diverse as sets [19, 23], dictionaries [10, 1], range trees [13], graphs [11], skip lists [10, 9], B-trees [17], hash trees [21], and more [12] have been proposed. In each of these cases, the design of the data structure, the supporting operations, and how they can be proved authentic have been reconsidered from scratch, involving a new, potentially tricky proof of security. Arguably, this state of affairs has hindered the advancement of new data structure designs, or customizations of existing designs, as previous ideas are not easily reused or reapplied. We believe that, motivated by cloud computing, ADSs will make their way into systems for secure computation more often if they become easier to build.

This paper presents $\lambda\bullet$ (pronounced "lambda auth"), a language for programming authenticated data structures. $\lambda\bullet$ represents the first *generic* approach to building dynamic authenticated data structures with provable guarantees. The key observation underlying $\lambda\bullet$'s design is that, whatever the data structure or operation, the computations performed by the prover and verifier can be made structurally the same: the prover can construct the proof by performing work at key points while executing the query, and the verifier can check that proof by using it to "replay" the query and check at each key point that the computation is self-consistent.

$\lambda\bullet$ implements this idea using what we call *authenticated types*, written $\bullet\tau$, with coercions auth and unauth for introducing and eliminating values of authenticated type. Using standard functional programming features, the programmer writes her ADS's data type definition and its corresponding operations (i.e., queries and updates) to use authenticated types. For example, as we show later in the paper, the programmer could write an efficient authenticated binary search tree using the (OCaml-style) type definition $\mathsf{type\ bst} = \mathsf{Tip} \mid \mathsf{Bin\ of} \bullet\mathsf{bst} \times \mathsf{int} \times \bullet\mathsf{bst}$ along with essentially standard routines for querying and insertion. Then, given such a program, the $\lambda\bullet$ compiler produces code for both a prover and a verifier. The prover's code augments each of a program's routines to generate a proof along with the answer, whereas each verifier routine verifies a proof produced by its counterpart at the prover. These proofs consist of a stream of what we call *shallow projections* of the data the prover visits while running its routine: the prover's code adds to this stream at each unauth call, while the verifier's code draws

[1] This property is sometimes called *soundness* but we eschew this term to avoid confusion with its standard usage in programming languages.

www.manaraa.com

from the stream at the corresponding call, checking for consistency. We give a more detailed overview of how this approach works, and how authenticated types are represented, in Section 2. Importantly, as we show in Sections 3 and 4, *any* well-typed program written in $\lambda\bullet$ compiles to a prover and verifier which are correct and secure, where security holds under the standard cryptographic assumption of *collision-resistant hash functions*.[2]

$\lambda\bullet$ provides two key benefits over prior work. First, it is extremely *flexible*. We can use $\lambda\bullet$ to implement any dynamic data structure, both queries and updates, expressible using ML-style data types (involving sums, products, and recursive types). Our theoretical development, though not our implementation, also supports authenticated functions. Previous work by Martel et al. [14] can also be used to build DAG-oriented ADSs, but they support only queries and not (incremental) updates, and (less importantly) require the data structure have a single root and do not support authenticated functions. $\lambda\bullet$'s flexibility does not compromise its performance. To the best of our knowledge the asymptotic performance of every prior ADS construction from the literature based on collision-resistant hashing can be matched by $\lambda\bullet$. We have implemented an optimizing $\lambda\bullet$ compiler as an extension to the OCaml compiler (described in Section 5), and using it we have implemented Merkle trees, authenticated binary search trees, red-black+ trees, skip lists, and planar separator trees, and improvements to standard Bitcoin data structures. Experiments given in Section 6 confirm the expected asymptotic performance of $\lambda\bullet$ ADSs, show the benefit of the two compiler optimizations we implemented (which exploit space/time tradeoffs), and demonstrate the performance of $\lambda\bullet$ ADSs is competitive with hand-rolled versions.

$\lambda\bullet$'s second main benefit is *ease of use*. We find that it is relatively simple to construct an ADS using $\lambda\bullet$: just write the standard data structure in a purely functional style, and sprinkle in uses of authenticated types; we give a flavor for this in the next section. Pleasantly, there is no need for the ADS designer to prove anything: Assuming the resulting program type checks, the programmer is assured that the produced prover and verifier code enjoy both correctness and security. $\lambda\bullet$ ADSs can be freely composed and customized just as one might expect with normal data structures, a fact which we hope will make them more readily deployable. All of this is in contrast to the state of practice with ADSs today, summarized in Section 7, which tends to favor hand-rolled versions that are hard to build, customize, and compose.

At present, $\lambda\bullet$ has two main limitations. First, it uses collision-resistant hashes as its basis for implementing authenticated types. While this is by far the most common approach to building ADSs, more recent work [23, 22] shows that advanced cryptographic primitives can sometimes provide efficiency gains. Second, while all its ADSs are correct and secure, $\lambda\bullet$ provides no guarantees that a particular program will be (space-)efficient; it is up to the programmer to use authenticated types wisely. In the limit, the programmer could leave out authenticated types entirely, with the result that the verifier would have to store the entire data structure, not its digest, and essentially duplicate the prover's computation. While we believe that efficiency is reasonably straightforward to attain in general—certainly simpler than designing and proving correctness and security of an ADS from scratch—an interesting direction is to automate the construction of efficient representations.

In summary, this paper makes the following contributions:

1. We present $\lambda\bullet$, a purely functional language in which one can write a rich array of authenticated data structures using a novel feature we call *authenticated types*.

[2] Informally, $hash$ is a collision-resistant hash function if it is computationally infeasible to find distinct inputs x, x' such that $hash(x) = hash(x')$.

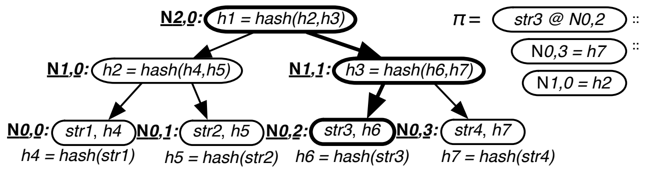


Figure 1. A Merkle tree and proof π for fetch(2) describing the highlighted path.

2. We formalize the semantics and type rules for $\lambda\bullet$ and prove that all well-typed $\lambda\bullet$ programs produce ADS protocols that are both correct and secure.
3. We have implemented $\lambda\bullet$ as an extension to the OCaml compiler and implemented a variety of existing and new ADSs, showing good asymptotic performance that is competitive with hand-rolled implementations. Our code will be made freely available and is included in the supplemental material.

2. Overview

This section presents an overview of our approach. We begin by describing Merkle trees, the canonical example of an authenticated data structure. Then we give a flavor for $\lambda\bullet$ by showing how we can use it to implement Merkle trees. We conclude with a discussion of the flexibility and ease-of-use benefits of using $\lambda\bullet$ to write efficient authenticated data structures.

2.1 Background: Merkle trees

The canonical example of an ADS is a *Merkle tree* [16], which is the authenticated version of a full binary tree with data associated with the leaves but not the interior nodes. A Merkle tree of height h can represent an array of $n = 2^{h-1}$ elements, $x_0, ..., x_{n-1}$. Each leaf node is coupled with a *digest* that consists of the hash of the associated element, while each internal node contains a digest that is the hash of the concatenation of the digests of its two children. A depiction of a Merkle tree for $h = 2$ is given in Figure 1. Each leaf is associated with a string $str1$, $str2$, etc. Each node is numbered according to its position in the tree, with x, y indicating x as the row and y as the column.

The canonical Merkle tree query fetches the value x_i at index $i \in [0, n-1]$. When thus queried, the prover (call it P) returns the value x_i along with the set of digests π needed to compute the root digest. The verifier (call it V) keeps a copy of the root digest itself, and checks the proof by recomputing this digest from the proof and make sure the two match. Figure 1 shows the proof π for a fetch at position $i = 2$ (i.e., the leaf at position $N0, 2$). Verification proceeds bottom up: V computes the hash of str_3, which is h_6, and concatenates that with the hash h_7 provided in π. It then concatenates these two and takes the hash to compute what should be the digest for node $N1, 1$, i.e., h_3. Then it concatenates h_2, the hash for $N1, 0$ provided in π, with its computed digest for $N1, 1$ and hashes the result—this should be the digest of the root of the tree. It then confirms this computed digest equals h_1, the digest it stores for the whole tree.

Performance analysis. Because the tree is perfectly balanced, the size of the proof is always $\log_2 n$; additionally the computational cost for each of P and V is $\log_2 n$. The overall size of the data structure stored by P is $O(n)$, whereas V at no point requires more than a constant amount of storage or memory. In particular, V only stores a constant-sized digest of the tree between fetch operations, and because P produces a proof consisting of a list of (constant-sized) hashes, P can stream these hashes to V as they are produced,

المنارة للاستشارات
www.manaraa.com

```
type tree = Tip of string | Bin of •tree × •tree
type bit = L | R
let rec fetch (idx:bit list) (t:•tree) : string =
  match idx, unauth t with
  | [], Tip a → a
  | L :: idx, Bin(l,_) → fetch idx l
  | R :: idx, Bin(_,r) → fetch idx r
```

Figure 2. Merkle trees in $\lambda\bullet$. The tree is assumed to be complete, i.e., with a power-of-two number of leaves.

effectively pipelining its execution with that of V. For this reason, we often refer to π as a proof *stream*.

Security analysis. As described in the Introduction, we are interested in two properties of this scheme. *Correctness* says that when P executes the query correctly, then V gets the same result as it would have if it had just computed $f(t)$ normally. The second property, *security*, says that a computationally bounded, cheating prover P^* cannot cause V to accept an incorrect answer. The basis of this property is the use of collision-resistant hashes: we can show that if P^* can cause V to accept an incorrect answer then the proof returned by P^* will yield a collision. We state these properties precisely, in the context of $\lambda\bullet$, in Section 4.

2.2 Introducing $\lambda\bullet$, a language for programming ADSs

The Merkle tree verification procedure was carefully designed with the properties of the underlying data structure in mind. In particular, there can be but one *path* from the root to a given leaf, and from this path we can determine digests sufficient to recompute the root digest. The question is: how might we generalize this approach to arbitrary data structures t involving arbitrary computations f? We designed $\lambda\bullet$ as a solution to this problem.

$\lambda\bullet$ is a completely standard, purely functional programming language extended with what we call *authenticated types* $\bullet\tau$, along with coercions *auth* and *unauth*, which have type $\forall\alpha.\alpha \rightarrow \bullet\alpha$ (for introducing authenticated values) and type $\forall\alpha.\bullet\alpha \rightarrow \alpha$ (for eliminating authenticated values), respectively. A function p using authenticated types is compiled to variations p_P and p_V for the prover and verifier, respectively. Data of type $\bullet\tau$ stored at the prover is like a normal value of type τ but augmented with digests, while data of type $\bullet\tau$ stored at the verifier is simply a compact digest. The *auth*/*unauth* coercions at the prover facilitate proof generation, while at the verifier they check a provided proof. In short, $\lambda\bullet$' design exploits the observation that proof generation and proof verification can be made structurally *identical* essentially by piggybacking them on top of the ideal computation of $f(t)$.

Example. As an illustration, Figure 2 defines a $\lambda\bullet$ implementation of Merkle trees, using OCaml-inspired syntax (which closely matches that of our implementation). The type tree is simply a binary tree with strings stored at the leaves. The fetch function takes an index expressed as a list of bits, which is interpreted by fetch as a path through the tree, with L bits directing the traversal to the left, and R bits directing it right. The function returns the string associated with the Tip that is eventually reached. Notice that since the argument t has type •tree, the function must call *unauth* t to coerce it to a tree to be matched against (we give a use of *auth* in Section 2.3).

Interpretation of authenticated types. The $\lambda\bullet$ compiler will produce two variations of a program p: program p_P for the prover, and p_V for the verifier. All standard constructs in p have the usual semantics in both variations, but authenticated types are interpreted differently.

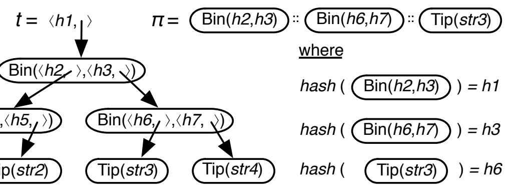


Figure 3. A Merkle tree t in $\lambda\bullet$ (of type •tree from Figure 2) and proof stream π for query fetch t [R; L]. Hashes of relevant shallow projections are given in the lower right.

Prover For p_P, values of type $\bullet\tau$ consist of pairs $\langle h, v\rangle$ where v has type τ and h is its digest, i.e., a hash of the *shallow projection* of v. Shallow projection is formally defined in Figure 9, but the intuition is simple: the shallow projection of a value is just the value itself for all values of type τ where τ does not consist of any authenticated types $\bullet\tau_0$, while the shallow projection of an authenticated value $\langle h, v\rangle$ is just the digest h. Looking at the definition of type tree in the figure, we can see that recursive references to the tree in the Bin case are authenticated. As such, the prover's representation is just as described in the previous subsection: each node of the tree has the form Bin$(\langle h_1, v_1\rangle, \langle h_2, v_2\rangle)$, which consists of the left subtree v_1 and its digest h_1, and the right subtree v_2 and its digest h_2. Each digest consists of the hash of the shallow projection of its respective tree. So, if v_1 was a leaf Tip(s), the shallow projection is just Tip(s) itself, and thus h_1 is the hash of Tip(s). On the other hand, suppose v_1 was a node Bin$(\langle h_{11}, v_{11}\rangle, \langle h_{12}, v_{12}\rangle)$. Then Bin$(h_{11}, h_{12})$ is this node's shallow projection, and h_1 is its digest.

Verifier For p_V, values of type $\bullet\tau$ consist solely of the digest h of some value of type τ. As such, for our example, while the prover maintains the entire tree data structure, the verifier only keeps the digest of the root. In general, values in p_V are the shallow projections of their corresponding values in p_P; we define this notion formally in the next section.

Turning to the coercions, for both the prover and verifier the *auth* v coercion computes the hash h of the shallow projection of v; for the prover, this hash is paired with v and for the verifier we retain only h itself. The interesting part is the semantics for *unauth*. For the Prover, the argument has the form $\langle h, v\rangle$ and the coercion simply returns v. In addition, it computes the shallow projection of v and adds it to the proof π, which is just a list of such shallow projections. We often refer to π as a proof *stream* to emphasize the list structure. For the verifier, the argument given to *unauth* is a hash h. It compares this hash against the hash of the element s at the head of the proof stream, which is a shallow projection of type τ. Assuming all is well, this element is the one the Prover put there and so the hashes will match. If so, the coercion returns s. Otherwise there is a problem and verification fails.

Example Merkle tree query. Figure 3 depicts the prover's version of an object of type •tree corresponding to the Merkle tree from Figure 1. These trees are structurally similar but not identical; in particular, notice that a node's digest is stored with the pointer to that node, rather than at the node itself. Suppose the prover executes the query (fetch [R; L] t), which corresponds to our earlier example query. The figure also depicts the proof stream π it produces, along with the hashes of shallow projections of relevant tree elements. The first thing the prover will execute is *unauth* t, which returns the pointer to the first node, and stores its shallow projection Bin(h_2,h_3) in the proof stream—notice that this is the same as the pointed-to node but the sub-tree pointers have been dropped. Execution continues to the third case of the

المنارة للاستشارات
www.manaraa.com

match, which recursively calls (fetch [L] r), where r is bound to the authenticated value $\langle h_3, v\rangle$ such that v is the right subtree. The prover then invokes $unauth$ on this pair, returning v and adding Bin(h_6,h_7) to the proof stream. This time we take the second case of the **match**, recursing on $\langle h_6, \text{Tip}(str_3)\rangle$. This time $unauth$ returns Tip(str_3) and adds its shallow projection (Tip(str_3) itself) to the proof stream. Execution concludes with str_3 as the final result while the final proof stream π consists of three elements, representing the three nodes visited.

The verifier begins with the proof stream π and just the digest of t, which is h_1. It then runs (fetch [R; L] t) using its version of the code. It first executes $unauth$ h_1, which compares h_1 to the hash of the first element s_0 of the proof stream, which is Bin(h_2,h_3). The hashes match, as per the equations given in the lower right of the figure, and thus execution continues using s_0. Execution proceeds to the third case of the **match**, recursively calling fetch with [L] and h_3, the digest of the right subtree. This time, calling $unauth$ h_3 results in comparing h_3 to the hash of the second element in the proof stream, which is Bin(h_6,h_7), and once again the hashes match and the proof stream element is returned. The second branch of the **match** fires, so the recursive call passes [] and h_6, the digest of the left subtree. In this case, $unauth$ h_6 compares h_6 to the hash of the final element of the proof stream, Tip(str_3), which is returned as the hashes match. Thus execution concludes with the final result as str_3. As all hash checks succeeded and the final result matches, and the proof stream is empty, the verifier has established the prover's execution is correct, with high probability.

Analysis. Our λ• Merkle trees are asymptotically as efficient as the originals, and as secure. As before, the verifier maintains only the constant-sized digest between queries, and the size of the fetch proof and the time to generate and verify it (in a pipelined fashion) is $O(\log_2 n)$: the proof stream consists of one (constant-size) shallow projection for each recursive call to fetch. The argument for security once again rests on collision-resistant hashes, though λ• verification effectively checks the root digest top-down rather than bottom-up. Our proof stream has some redundancy (it contains hashes h_2, h_3, h_6, and h_7, whereas in Figure 1 the proof contains only h_2 and h_7) but this is only a constant factor and as we show in Section 5.2 it can be optimized away.

2.3 Discussion: Benefits of λ•

The primary benefit of writing ADSs in λ• over prior approaches is *flexibility* and *ease of use*. λ• can support essentially any computation over a DAG-oriented data structure that is expressed as functional program. Moreover, as we show in Section 4, by virtue of writing an ADS in λ• we are sure it is both correct and secure; there is no need for a designer to make a new argument for security with each new data structure. As far as we are aware, λ• can be used to implement any previously proposed ADS based on collision-resistant hashing. In particular, as described in Section 6, so far we successfully implemented Merkle trees and authenticated versions of Binary Search Trees, Red-black+ Trees [20], Skip Lists [25], and variations of the Bitcoin block chain [18], all of which enjoy asymptotically identical, or better, performance than their specially-designed counterparts.

Support for updates. Martel et al. [14] also previously proposed a general-purpose scheme that supports ADSs based on DAGs. In principle, their scheme could also support the above-mentioned data structures, but only for query computations, not updates. By contrast, updates are completely natural in λ•. For example, the function update in Figure 4 updates a Merkle tree. The verifier could submit a request to the prover to run (update [R; L] t str_5). The prover will produce a proof stream π for the operation along with a new authenticated tree t' that contains the modification, and

```
let rec update (idx:bit list) (t:•tree) (newval:string) : •tree =
  match idx, unauth t with
  | [], Tip _ → auth(Tip newval)
  | L::idx', Bin(l,r) → auth(Bin(update idx' l newval, r))
  | R::idx', Bin(l,r) → auth(Bin(l, update idx' r newval))

let update_cps (idx:bit list) (t:•tree) (newval:string) : •tree =
  let rec _update (k : •(•tree → •tree)) idx t x : •tree =
    match idx, unauth t with
    | [], Tip _ → (unauth k) (auth(Tip x))
    | L :: idx', Bin(l,r) →
      _update (auth(fun t → auth(Bin(t,r)))) idx' l x
    | R :: idx', Bin(l,r) →
      _update (auth(fun t → auth(Bin(l,t)))) idx' r x
  in _update (auth(fun t → t)) idx t newval

type stack = E | SL of •stack × •tree | SR of •stack × •tree
let update_stk (idx:bit list) (t:•tree) (newval:string) : •tree =
  let rec build idx t (s:stack) : •tree × stack =
    match idx, unauth t with
    | [], Tip _ → auth(Tip newval), s
    | L::idx', Bin(l,r) → build idx l (SL(auth s, r))
    | R::idx', Bin(l,r) → build idx r (SL(auth s, l)) in
  let rec apply (child:•tree, s:stack) : •tree =
    match s with
    | E → child
    | SL(s, r) → apply (auth(Bin(child, r)), unauth s)
    | SR(s, l) → apply (auth(Bin(l, child)), unauth s) in
  apply (build idx t E)
```

Figure 4. Functions for updating a Merkle tree in λ•

which shares much of the structure of the original tree t, as per standard functional programming style. The prover can then update its root to now be t' and then send the verifier the result of the execution, which is the digest portion of t' and the proof stream π. The verifier can then use π in the usual way to verify that (the digest of) t' is indeed the right result and then update its local root.[3]

Controlling performance. The fact λ• is a general-purpose programming language means that it affords substantial flexibility to the ADS designer in customizing an ADS design to her needs.

As one possible customization, the designer might refactor operations to better control space usage. Consider the update function once again. While verification of an update can be pipelined, so that the proof stream contributes only a constant amount to the memory requirements of the verifier, we observe that executing update will require $O(\log n)$ stack space, since the function is not tail recursive. One way to eliminate this overhead is to rewrite update in continuation-passing style (CPS) such that the continuation itself is authenticated, as for the function update_cps given in the middle of Figure 4. As such, recursive uses of nested continuations will be replaced with a hash, effectively bounding the depth of the stack encoded in the continuation. To the best of our knowledge, no prior work has considered authenticated closures. Another way to achieve the same effect, but perhaps less elegantly, is to use an explicit authenticated stack as is done by update_stk given at the bottom of the figure.

[3] In general, the prover will return the shallow projection of the result of a computation back to the verifier; when the result is a normal value the prover will thus return the value itself (as with the boolean result in our (fetch [R; L] t) example query).

المنارة للاستشارات
www.manaraa.com

$$
\begin{array}{rcl}
\text{Types } \tau & ::= & 1 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha.\tau \mid \alpha \mid \bullet\tau \\
\text{Values } v & ::= & () \mid x \mid \lambda x.e \mid \mathbf{rec}\ x.\lambda y.e \\
 & \mid & \mathbf{inj}_1\ v \mid \mathbf{inj}_2\ v \mid (v_1, v_2) \mid \mathbf{roll}\ v \\
\text{Exprs } e & ::= & v \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid v_1\ v_2 \mid \mathbf{case}\ v\ v_0\ v_1 \\
 & \mid & \mathbf{prj}_1\ v \mid \mathbf{prj}_2\ v \mid \mathbf{unroll}\ v \mid \mathit{auth}\ v \mid \mathit{unauth}\ v
\end{array}
$$

Figure 5. Syntax for types and terms

The designer could also tune performance by adjusting the definition of the data structure itself. For example, we could have defined Merkle trees instead as follows:

```
type tree = Tip of string | Bin of •(tree × tree)
```

In this case, we are only hashing nodes, and will never hash tips. This definition makes more sense when the hash of the Tip is larger than the representation of Tip itself, e.g., if the tree stored integers instead of strings. As another variation, we might imagine defining a tree that only optionally authenticates its children:

```
type tree =
  | Tip of string
  | Bin of tree × tree
  | AuthBin of •(tree × tree)
```

Then the tree might go several levels using Bin before using AuthBin. This design thus increases the constant factor on asymptotic space usage, but may reduce proving/verification time.

All of these customizations are possible, and easy to experiment with, thanks to the fact that $\lambda\bullet$ is a general-purpose programming language. However, this flexibility cuts both ways: there is nothing (at the moment) stopping the programmer from producing a suboptimal design. As an extreme example, the programmer could write type tree = Tip of string | Bin of tree × tree—i.e., without any use of authenticated types! This design will be secure and correct, as with every $\lambda\bullet$ program, but will effectively require the verifier to maintain the entire tree, not simply a digest. Fortunately, there is a simple rule of thumb that may have already become evident to the reader by this point: the data structure type definition should authenticate recursive references, thus aiming for shallow projections to be constant-sized. We leave to interesting future work the task of automating the transformation of $\bullet$-free type definition to an efficient authenticated one.

3. $\lambda\bullet$: A Language with Authenticated Types

This section formalizes $\lambda\bullet$, our language for writing computations over authenticated data structures. We present the syntax, typing rules, and operational semantics for $\lambda\bullet$ programs. The next section proves that $\lambda\bullet$ computations produce correct and secure results.

3.1 Syntax

Figure 5 presents the syntax for $\lambda\bullet$. Other than authenticated types $\bullet\tau$, the type language is entirely standard, consisting of the unit type 1, function types $\tau_1 \to \tau_2$, sum types $\tau_1 + \tau_2$, product types $\tau_1 \times \tau_2$, recursive types $\mu\alpha.\tau$ and variable types α arising from these. In this syntax, our authenticated tree type defined in Figure 2 would be written $\mu\alpha.\mathsf{string} + (\bullet\alpha \times \bullet\alpha))$, where string would itself be encoded, e.g., as a list of Peano-style integers. Our language does not include parametric polymorphism, for simplicity, but we see no difficulties with adding it. The language also does not support references because mutations would risk invalidating hashes for $\bullet\tau$ values. In particular, given an authenticated value $\langle h, v\rangle$ where v is a reference, a mutation via v may invalidate h.

$$
\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash \mathbf{inj}_1\ v : \tau_1 + \tau_2}
\qquad
\frac{\Gamma \vdash v : \tau_2}{\Gamma \vdash \mathbf{inj}_2\ v : \tau_1 + \tau_2}
$$

$$
\frac{\Gamma \vdash v : \tau_1 + \tau_2 \qquad \Gamma \vdash v_1 : \tau_1 \to \tau \qquad \Gamma \vdash v_2 : \tau_2 \to \tau}{\Gamma \vdash \mathbf{case}\ v\ v_1\ v_2 : \tau}
$$

$$
\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathit{auth}\ v : \bullet\tau}
\qquad
\frac{\Gamma \vdash v : \bullet\tau}{\Gamma \vdash \mathit{unauth}\ v : \tau}
$$

Figure 6. Selected typing rules

Terms (values v and expressions e) are in administrative normal form [6] to keep the semantics simple. In this form, the grammar forces us to write **let** $x = e_1$ **in let** $x = e_2$ **in** $x\ y$ instead of the more familiar $e_1\ e_2$, for example. In addition to variables x, the term language includes functions $\lambda x.e$ and function application $v_1\ v_2$; sum-type values $\mathbf{inj}_1\ v$ and $\mathbf{inj}_2\ v$ which are eliminated by **case** $v\ v_0\ v_1$, where v_0 and v_1 are expected to be functions; products (v_1, v_2) eliminated by expressions $\mathbf{prj}_1\ v$ and $\mathbf{prj}_2\ v$; values of recursive type introduced via **roll** v and eliminated by **unroll** v; and finally fixpoints **rec** $x.\lambda y.e$ for defining recursive functions (where inside of $\lambda y.e$ references to x refer to the function itself). Authenticated types $\bullet\tau$ are introduced by coercion *auth* and eliminated by *unauth*.

3.2 Typing

The typing judgment for $\lambda\bullet$ programs is the usual one, written $\Gamma \vdash e : \tau$. It states that expression e has type τ under environment Γ, where Γ is a map from variables x to types τ. Typing rules for most constructs are standard. Selected rules are given in Figure 6: the rules for sum types are given at the top of the figure, while the rules for *auth* and *unauth* are given at the bottom.

3.3 Operational semantics

In practice, our compiler takes a program like the one in Figure 2 and outputs versions to be run by the prover and the verifier. In our formalization, we instead define distinct semantics for the same program, as determined by an execution mode m, where $m = P$ for the prover's execution, and $m = V$ for the verifier's. We also define a mode I for the *Ideal* case, representing a computation that happens in the normal way, ignoring authenticated types; this is needed for stating the security and correctness properties.

We define a small-step operational semantics having the form $\ll \pi, e \gg \to_m \ll \pi', e' \gg$, where π is the proof stream, which is a list of shallow projections s, and m is the mode. This can be read: *An expression e coupled with a proof stream π can evaluate one step in mode m to produce an expression e' and an updated proof stream π'.* We define $\ll \pi, e \gg \to_m^i \ll \pi', e' \gg$ to be the reflexive, transitive closure of the single-step relation; it states that *e evaluates, in i steps, to e' in mode m, starting with proof stream π and finishing with π'.* The proof stream is *produced* in mode P, so π is a prefix of π' in this mode. The stream is *consumed* in mode V, and thus π' is a suffix of π. The proof stream is ignored in mode I. We use the operator @ to denote the concatenation of two proof streams, treating @ as associative with the empty stream [] as the identity. We write $[s]$ as the singleton stream containing element s.

The rules for standard language features are identical in all three modes, and are standard. They are defined in the top portion of Figure 7, and we discuss them briefly in order. The rule for function application $(\lambda x.e)\ v$ substitutes v for x in e—this substitution is written $e[v\backslash x]$. Application of a recursive function is similar: when the function (**rec** $x.\lambda y.e$) is on the lhs of an application, we substitute x in the function body e with the recursive function itself.

المنارة للاستشارات
www.manaraa.com

$$
\begin{array}{lll}
\ll \pi, (\lambda x.e)\ v \gg & \rightarrow_m & \ll \pi, e[v\backslash x] \gg \\
\ll \pi, (\textbf{rec}\ x.\lambda y.e)\ v \gg & \rightarrow_m & \ll \pi, (\lambda y.e')\ v \gg \\
 & \multicolumn{2}{r}{\text{where } e' = e[(\textbf{rec}\ x.\lambda y.e)\backslash x]} \\
\ll \pi, \textbf{let}\ x = v_1\ \textbf{in}\ e_2 \gg & \rightarrow_m & \ll \pi, e_2[v_1\backslash x] \gg \\
\ll \pi, \textbf{case}\ (\textbf{inj}_1\ v)(\lambda x.e_1)(\lambda x.e_2) \gg & \rightarrow_m & \ll \pi, e_1[v\backslash x] \gg \\
\ll \pi, \textbf{case}\ (\textbf{inj}_2\ v)(\lambda x.e_1)(\lambda x.e_2) \gg & \rightarrow_m & \ll \pi, e_2[v\backslash x] \gg \\
\ll \pi, \textbf{prj}_1\ (v_1, v_2) \gg & \rightarrow_m & \ll \pi, v_1 \gg \\
\ll \pi, \textbf{prj}_2\ (v_1, v_2) \gg & \rightarrow_m & \ll \pi, v_2 \gg \\
\ll \pi, \textbf{unroll}\ (\textbf{roll}\ v) \gg & \rightarrow_m & \ll \pi, v \gg
\end{array}
$$

$$
\frac{\ll \pi, e_1 \gg \rightarrow_m \ll \pi', e_1' \gg}{\ll \pi, \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \gg \rightarrow_m \ll \pi', \textbf{let}\ x = e_1'\ \textbf{in}\ e_2 \gg}
$$

$$
\frac{\ll \pi, e \gg \rightarrow_m^i \ll \pi', e' \gg \qquad \ll \pi', e' \gg \rightarrow_m \ll \pi'', e'' \gg}{\ll \pi, e \gg \rightarrow_m^{i+1} \ll \pi'', e'' \gg}
\qquad
\ll \pi, e \gg \rightarrow_m^0 \ll \pi, e \gg
$$

Figure 7. Standard single-step and multi-step operational rules

$$
\begin{array}{lll}
\ll \pi, \textit{auth}\ v \gg & \rightarrow_I & \ll \pi, v \gg \\
\ll \pi, \textit{unauth}\ v \gg & \rightarrow_I & \ll \pi, v \gg \\
\ll \pi, \textit{auth}\ v \gg & \rightarrow_P & \ll \pi, \langle \textit{hash}\ \llparenthesis v \rrparenthesis, v \rangle \gg \\
\ll \pi, \textit{unauth}\ \langle h, v \rangle \gg & \rightarrow_P & \ll \pi\ @\ [\ \llparenthesis v \rrparenthesis\ ],\ v \gg \\
\ll \pi, \textit{auth}\ v \gg & \rightarrow_V & \ll \pi, \textit{hash}\, v \gg
\end{array}
$$

$$
\frac{\textit{hash}\ s_0 = h}{\ll [s_0]\ @\ \pi, \textit{unauth}\ h \gg \rightarrow_V \ll \pi, s_0 \gg}
$$

$$
\text{where } v ::= \ldots \mid h \mid \langle h, v \rangle
$$

Figure 8. Operational rules for authenticated values

$$
\begin{array}{lll@{\qquad}lll}
\llparenthesis () \rrparenthesis & = & () & \llparenthesis x \rrparenthesis & = & x \\
\llparenthesis \langle h, v \rangle \rrparenthesis & = & h & \llparenthesis \lambda x.e \rrparenthesis & = & \lambda x.\llparenthesis e \rrparenthesis \\
\llparenthesis \textit{auth}\ v \rrparenthesis & = & \textit{auth}\ \llparenthesis v \rrparenthesis & \llparenthesis \textit{unauth}\ v \rrparenthesis & = & \textit{unauth}\ \llparenthesis v \rrparenthesis \\
\llparenthesis (v_1, v_2) \rrparenthesis & = & (\llparenthesis v_1 \rrparenthesis, \llparenthesis v_2 \rrparenthesis) & \llparenthesis \textbf{prj}_i\ v \rrparenthesis & = & \textbf{prj}_i\ \llparenthesis v \rrparenthesis \\
\llparenthesis \textbf{roll}\ v \rrparenthesis & = & \textbf{roll}\ \llparenthesis v \rrparenthesis & \llparenthesis \textbf{unroll}\ v \rrparenthesis & = & \textbf{unroll}\ \llparenthesis v \rrparenthesis \\
\llparenthesis \textbf{rec}\ x.\lambda y.e \rrparenthesis & = & \textbf{rec}\ x.\llparenthesis \lambda y.e \rrparenthesis & \llparenthesis \textbf{inj}_i\ v \rrparenthesis & = & \textbf{inj}_i\ \llparenthesis v \rrparenthesis
\end{array}
$$

$$
\begin{array}{lcl}
\llparenthesis \textbf{case}\ v\ v_0\ v_1 \rrparenthesis & = & \textbf{case}\ \llparenthesis v \rrparenthesis\ \llparenthesis v_0 \rrparenthesis\ \llparenthesis v_1 \rrparenthesis \\
\llparenthesis \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rrparenthesis & = & \textbf{let}\ x = \llparenthesis e_1 \rrparenthesis\ \textbf{in}\ \llparenthesis e_2 \rrparenthesis
\end{array}
$$

Figure 9. Shallow projection of an expression e, written $\llparenthesis e \rrparenthesis$

Let-binding is used to sequence computations, either evaluating the bound expression e_1 one step or else, if this expression is a value v_1, substituting that value for x in the body e_2. The semantics of **case** depends on whether it is given $\textbf{inj}_1\ v$ or $\textbf{inj}_2\ v$; in the former case we substitute v in the first lambda term (the "true branch"), else we substitute it in the second ("false") one. Projection from a pair (v_1, v_2) produces v_1 for $\textbf{prj}_1$ and v_2 for $\textbf{prj}_2$. Finally, the recursive type coercions **unroll** and **roll** nullify each other.

The rules for the multi-step relation are given at the bottom of Figure 7, and are also standard.

The operational rules for authenticated values are given in Figure 8. For mode I, authenticated values of type $\bullet\tau$ are merely values of type τ and the $\textit{auth}/\textit{unauth}$ operations are no-ops. For mode P, values of type $\bullet\tau$ are implemented as a pair $\langle h, v \rangle$ of a hash h and a value v (of type τ). As shown in the $\textit{auth}$ rule, the hash is computed by applying a hash function $\textit{hash}$ over the shallow projection of v, written $\llparenthesis v \rrparenthesis$. We do not formalize the semantics of $\textit{hash}$ explicitly; in practice it can be implemented by serializing the value it is given and hashing that using a collision-resistant hash function.[4] The shallow projection operation is defined in Figure 9. It is essentially a fold over the structure of the term, preserving that structure in every case but that of values $\langle h, v \rangle$: here we simply drop the value v and retain the hash h. Another interesting case is functions $\lambda x.e$: we recursively descend into e to translate any $\langle h, v \rangle$ values that appear there. Such values will not appear in source programs, but they can arise via substitution under lambdas. Returning to Figure 8, the mode-P semantics of $\textit{unauth}\ \langle h, v \rangle$ is to strip off the hash, returning v, while adding the shallow projection of v to the end of the proof stream. Finally, for mode V the representation of $\bullet\tau$ is the hash h of a value of type τ. The $\textit{auth}$ rule constructs this representation, while the $\textit{unauth}$ rule checks that the hash value matches the shallow projection at the head of the proof stream.

4. Metatheory

We want to show that well-typed $\lambda\bullet$ programs will (a) produce correct results—that is, results that all three modes agree on—or else (b) a malicious prover has been able to find a hash collision, which we assume is computationally difficult. We call property (a) *correctness* and property (b) *security*. In this section we state and prove these two properties.

4.1 Agreement

Before defining these properties, we must define what we mean when we say that the different execution modes "agree" on their result—it cannot be that these results are syntactically equal because each mode interprets authenticated values differently. For example, consider the `update` function from Figure 4. In the ideal setting, this function will return a normal tree v—because $\bullet\tau$ values are the same as those of type τ, the tree v will contain no digests. On the other hand, the prover will return a value $\langle h, v_P \rangle$, where h is the digest of v_P. For the same insertion on the same tree, the results v and $\langle h, v_P \rangle$ in I and P modes, respectively, should "agree" without being equal: v will just be a normal tree, while v_P will contain digests (one per node)—but the elements and sub-trees, excepting the digests, should be the same. Finally, running the insertion at the verifier will return a digest h, which should be the same digest contained in the prover's returned value $\langle h, v_P \rangle$.

We formalize this connection as a three-way type-indexed relation $\Gamma \vdash e \;\; e_P \;\; e_V : \tau$, given in Figure 10, which states *In environment* Γ*, Ideal expression* e*, prover expression* e_P*, and verifier expression* e_V *all* agree *at type* τ. In every case but that of authenticated values (the last rule), agreement follows syntactic structure of the terms, and the shape of each rule matches that of the standard type rules. The rule for authenticated values formalizes the intuition given above; it states that $\Gamma \vdash v \;\; \langle h, v_P \rangle \;\; h : \bullet\tau$ holds when (a) the digest h of both the prover and verifier is the same; (b) this digest is the hash of the shallow projection of the prover's value; (c) the prover's value agrees with the Ideal value. Note that the definition makes use of the shallow projection of v_P for (c)—during real execution, this value comes from the proof stream.

The agreement relation is intimately connected to the shallow projection operator; in fact, we can prove that a prover's term only ever agrees with a verifier's term when the latter is the shallow projection of the former. Moreover, we prove for any given ideal term e, there is at most one pair of terms e_P and e_V that agree with it under a given environment Γ and type τ.

[4] For functions $\lambda x.e$, serialization involves pretty-printing the actual code.

المنارة للاستشارات

www.manaraa.com

$$\Gamma \vdash ()\;()\;() : 1 \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x\;x\;x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e \;\; e_P \;\; e_V : \tau_2}{\Gamma \vdash (\lambda x.e)\;(\lambda x.e_P)\;(\lambda x.e_V) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash v_1 \;\; v_{1P} \;\; v_{1V} : \tau_1 \to \tau_2 \qquad \Gamma \vdash v_2 \;\; v_{2P} \;\; v_{2V} : \tau_1}{\Gamma \vdash (v_1\;v_2)\;(v_{1P}\;v_{2P})\;(v_{1V}\;v_{2V}) : \tau_2}$$

$$\frac{\Gamma \vdash e_1 \;\; e_{1P} \;\; e_{1V} : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 \;\; e_{2P} \;\; e_{2V} : \tau_2}{\Gamma \vdash (\textbf{let}\; x = e_1 \;\textbf{in}\; e_2)\;(\textbf{let}\; x = e_{1P} \;\textbf{in}\; e_{2P})\;(\textbf{let}\; x = e_{1V} \;\textbf{in}\; e_{2V}) : \tau_2} \qquad \frac{\Gamma, x{:}\tau_1 \to \tau_2 \vdash (\lambda y.e)\;\;(\lambda y.e_P)\;\;(\lambda y.e_V) : \tau_1 \to \tau_2}{\Gamma \vdash (\textbf{rec}\; x.\lambda y.e)\;(\textbf{rec}\; x.\lambda y.e_P)\;(\textbf{rec}\; x.\lambda y.e_V) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau_1}{\Gamma \vdash (\textbf{inj}_1\; v)\;(\textbf{inj}_1\; v_P)\;(\textbf{inj}_1\; v_V) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau_2}{\Gamma \vdash (\textbf{inj}_2\; v)\;(\textbf{inj}_2\; v_P)\;(\textbf{inj}_2\; v_V) : \tau_1 + \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash v \;\; v_P \;\; v_V : \tau_1 + \tau_2 \\ \Gamma \vdash v_P \;\; v_{1P} \;\; v_{1V} : \tau_1 \to \tau \\ \Gamma \vdash v_V \;\; v_{1V} \;\; v_{2V} : \tau_2 \to \tau\end{array}}{\Gamma \vdash (\textbf{case}\; v\; v_1\; v_2)\;(\textbf{case}\; v_P\; v_{1P}\; v_{2P})\;(\textbf{case}\; v_V\; v_{1V}\; v_{2V}) : \tau} \qquad \frac{\Gamma \vdash v_1 \;\; v_{1P} \;\; v_{1V} : \tau_1 \qquad \Gamma \vdash v_2 \;\; v_{2P} \;\; v_{2V} : \tau_2}{\Gamma \vdash (v_1, v_2)\;(v_{1P}, v_{2P})\;(v_{1V}, v_{2V}) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau_1 \times \tau_2}{\Gamma \vdash (\textbf{prj}_1 v)\;(\textbf{prj}_1\; v_P)\;(\textbf{prj}_1\; v_V) : \tau_1} \qquad \frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau_1 \times \tau_2}{\Gamma \vdash (\textbf{prj}_2\; v)\;(\textbf{prj}_2\; v_P)\;(\textbf{prj}_2\; v_V) : \tau_2} \qquad \frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau[\mu\alpha.\tau \backslash \alpha]}{\Gamma \vdash (\textbf{roll}\; v)\;(\textbf{roll}\; v_P)\;(\textbf{roll}\; v_V) : \mu\alpha.\tau}$$

$$\frac{\Gamma \vdash v \;\; v_P \;\; v_V : \mu\alpha.\tau}{\Gamma \vdash (\textbf{unroll}\; v)\;(\textbf{unroll}\; v_P)\;(\textbf{unroll}\; v_V) : \tau[\mu\alpha.\tau \backslash \alpha]} \qquad \frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau}{\Gamma \vdash (\mathit{auth}\; v)\;(\mathit{auth}\; v_P)\;(\mathit{auth}\; v_V) : \bullet\tau}$$

$$\frac{\Gamma \vdash v \;\; v_P \;\; v_V : \bullet\tau}{\Gamma \vdash (\mathit{unauth}\; v)\;(\mathit{unauth}\; v_P)\;(\mathit{unauth}\; v_V) : \tau} \qquad \frac{\vdash v \;\; v_P \;\; (\![v_P]\!) : \tau \qquad \mathit{hash}\;(\![v_P]\!) = h}{\Gamma \vdash v \;\; \langle h, v_P \rangle \;\; h : \bullet\tau}$$

Figure 10. Agreement relation: defines those expressions that *agree* (are morally, if not syntactically, the same) in the Ideal, Prover, and Verifier modes. The most interesting rule is the last one, while the rest are three-way versions of the standard type rules.

Lemma 1 (Agreement and shallow projection)**.**
Suppose $\Gamma \vdash e \;\; e_P \;\; e_V : \tau$. *Then*

1. $(\![e_P]\!) = e_V$.
2. $\Gamma \vdash e \;\; e'_P \;\; e'_V : \tau$ *implies that* $e'_P = e_P$ *and* $e_V = e'_V$.

Proof. By induction on $\Gamma \vdash e \;\; e_P \;\; e_V : \tau$. ☐

Client and server agree. In our client and server scenario, a query/update sent by the client will reference the data structure stored at the server using a free variable, e.g., the t in the query `member` t 4. To run this query on the server, we substitute the prover's representation for t, while to verify the result at the client, we substitute t's digest. These representations should agree. The following lemma states that, given an expression to run, call it e, containing free variables with authenticated type, then substituting authenticated values that agree for the free variables of e produces versions e_I, e_P, and e_V that also agree.

Lemma 2. *Given the following:*

1. $\Gamma \vdash e : \tau$ *where* e *contains no values of type* $\bullet\tau$
2. *For all* $x_i \in \mathit{domain}(\Gamma)$,
   (a) $\Gamma(x_i) = \bullet\tau_i$ *for some* τ_i
   (b) $\vdash v_i \;\; \langle h_i, v_{Pi} \rangle \;\; h_i : \bullet\tau_i$ *for some* $\langle h_i, v_i \rangle$ *and* v_{Pi}
3. $e_P = e[\langle h_1, v_{P1} \rangle \backslash x_1]...[\langle h_n, v_{Pn} \rangle \backslash x_n]$
   $e_V = e[h_1 \backslash x_1]...[h_n \backslash x_n]$
   $e_I = e[v_1 \backslash x_1]...[v_n \backslash x_n]$

Then $\vdash e_I \;\; e_P \;\; e_V : \tau$.

The proof of this lemma follows from straightforward application of the following, more general, substitution lemma:

Lemma 3 (Substitution)**.** *If* $\Gamma, x{:}\tau' \vdash e \;\; e_P \;\; e_V : \tau$ *and* $\vdash v \;\; v_P \;\; v_V : \tau'$ *then* $\Gamma \vdash (e[v \backslash x])\;(e_P[v_P \backslash x])\;(e_V[v_V \backslash x]) : \tau$.

Proof. The proof is by induction on $\Gamma, x{:}\tau' \vdash e \;\; e_P \;\; e_V : \tau$. The only interesting case is when $\Gamma, x{:}\tau' \vdash v' \;\; \langle h, v'_P \rangle \;\; h : \bullet\tau$. The empty environment in the premise $\vdash v' \;\; v'_P \;\; (\![v'_P]\!) : \tau$ ensures that v' and v'_P contain no variables, so the substitution will be the identity and the result follows by assumption. ☐

4.2 Correctness and Security

Now we can state and prove our main theorem, Theorem 1, which encapsulates the two properties of interest, correctness and security. Informally, the theorem says that if we start with three terms e_I, e_P, and e_V that agree (which will be the case at the start of evaluating a query/update as per Lemma 2), and each term takes i evaluation steps, then we have one of two possible outcomes, based on the proof streams produced/consumed by the prover/verifier. The prover's execution will produce stream π'_p. If the verifier consumes this stream (e.g., because the server executes as it should) then the resulting terms will be related; i.e., the computation is *correct*. Moreover, if those terms are not values (that is, the computation is not complete), they can evaluate at least one additional step. On the other hand, if the verifier does not consume π'_p but rather some other stream (e.g., because the server is behaving maliciously or incorrectly), then the only way all three terms could have taken i steps is if that consumed stream contains a *hash collision*. That is, the consumed stream contains an element $s^\dagger$ that corresponds to an element s in π'_p such that $s \neq s^\dagger$ but $\mathit{hash}\; s = \mathit{hash}\; s^\dagger$. A corollary of this statement is that if at any point during their execution the three terms disagree, then it can only be because of a hash collision (something which is assumed to be computationally infeasible). We now state this theorem (and corollary) formally.

المنارة للاستشارات
www.manaraa.com

Theorem 1. *Given the following:*

- $\vdash e_I \;\; e_P \;\; e_V : \tau$
- $\ll [], e_I \gg \to_I^i \ll [], e'_I \gg$
- $\ll [], e_P \gg \to_P^i \ll \pi'_p, e'_P \gg$
- $\ll \pi_v, e_V \gg \to_V^i \ll \pi'_v, e'_V \gg$

Then

Correctness: $\pi_v = \pi'_p \mathbin{@} \pi'_v$ *implies* $\vdash e'_I \;\; e'_P \;\; e'_V : \tau$.
Moreover, either

1. *e'_I, e'_P, and e'_V are all values, or*
2. *there exist e''_I and $\ll \pi''_p, e''_P \gg$ and $\ll \pi''_v, e''_V \gg$ s.t.*
   - $\ll [], e_I \gg \to_I^{i+1} \ll [], e''_I \gg$
   - $\ll [], e_P \gg \to_P^{i+1} \ll \pi''_p, e''_P \gg$
   - $\ll \pi_v, e_V \gg \to_V^{i+1} \ll \pi''_v, e''_V \gg$ *or else it is stuck on an $unauth$.*

Security: $\pi_v \neq \pi'_p \mathbin{@} \pi'_v$ *implies there exists some π_0, π''_p, π''_v, s and $s^\dagger$ such that $\pi'_p = \pi_0 \mathbin{@} [s] \mathbin{@} \pi''_p$ and $\pi_v = \pi_0 \mathbin{@} [s^\dagger] \mathbin{@} \pi''_v \mathbin{@} \pi'_v$ where $s \neq s^\dagger$ but $hash\, s = hash\, s^\dagger$.*

Corollary 1. *If $\vdash e_I \;\; e_P \;\; e_V : \tau$ and each term takes i steps to e'_I, e'_P, and e'_V, respectively, then $\not\vdash e'_I \;\; e'_P \;\; e'_V : \tau$ implies the adversary has found a hash collision.*

The proof of the main theorem is by induction on the length i of the multi-step derivations. Key parts of the argument appeal to lemmas about single-step evaluation, which we present next. In particular, we prove correctness by proving variations of the standard *progress* and *preservation* lemmas, while we prove security by appealing to a single-step *security* lemma.

Our progress lemma states that terms in agreement are either values, or they can evaluate (at least) one step in their respective mode, excepting the possibility that in verifier mode the term may be stuck on a failed hash check (e.g., due to a corrupted proof stream); importantly, this is the only reason the verifier will fail.

Lemma 4 (Progress). *If $\vdash e \;\; e_P \;\; e_V : \tau$ then either e, e_P, and e_V are values, or for all π there exist e', e'_P, e'_V, π', π'' such that*

1. $\ll \pi, e \gg \to_I \ll \pi, e' \gg$
2. $\ll \pi, e_P \gg \to_P \ll \pi', e'_P \gg$
3. $\ll \pi, e_V \gg \to_V \ll \pi'', e'_V \gg$
   or else $unauth\;\; h$ is in redex position but cannot step because either $\pi = []$ or $\pi = [s] \mathbin{@} \pi_0$ for some π_0, and $hash\, s \neq h$.

Proof. By straightforward induction on $\vdash e \;\; e_P \;\; e_V : \tau$. □

Next we prove a variation of the standard preservation lemma. In particular we prove that if we have three terms that agree and each term can take a step, then the terms to which they step agree if and only if the proof stream is properly formed, i.e., the verifier consumes some element s of the proof stream produced by the prover, or no element is produced/consumed.

Lemma 5 (Preservation). *Given the following:*

1. $\vdash e \;\; e_P \;\; e_V : \tau$
2. $\ll \pi, e \gg \to_I \ll \pi, e' \gg$
3. $\ll \pi_p, e_P \gg \to_P \ll \pi'_p, e'_P \gg$
4. $\ll \pi_v, e_V \gg \to_V \ll \pi'_v, e'_V \gg$

Then $\vdash e' \;\; e'_P \;\; e'_V : \tau$ if and only if $\pi'_p = \pi_p$ and $\pi'_v = \pi_v$ or there exists some s such that $\pi'_p = \pi_p \mathbin{@} [s]$ and $\pi_v = [s] \mathbin{@} \pi'_v$.

Proof. By induction on $\vdash e \;\; e_P \;\; e_V : \tau$. Most cases are straightforward, and follow by application of the Substitution lemma. The two interesting cases deal with authenticated computations:

- Suppose e, e_P, and e_V are $auth\;\; v$, $auth\;\; v_P$ and $auth\;\; v_V$, respectively. Each takes a step in its respective mode, producing v, $\langle hash\, \llparenthesis v_P \rrparenthesis, v_P \rangle$, and $hash\, v_V$, respectively, resulting in $\pi_p = \pi'_p$ and $\pi_v = \pi'_v$. Now we must prove $\vdash v \;\; \langle hash\, \llparenthesis v_P \rrparenthesis, v_P \rangle \;\; hash\, v_V : \bullet\tau$, which in turn requires proving $\vdash v \;\; v_P \;\; \llparenthesis v_P \rrparenthesis : \tau$ and $hash\, \llparenthesis v_P \rrparenthesis = hash\, v_V$. Both are the consequence of Lemma 1-1 and $\vdash e \;\; e_P \;\; e_V : \tau$.
- Suppose e, e_P, and e_V are $unauth\;\; v$, $unauth\;\; v_P$ and $unauth\;\; v_V$, respectively. Since each takes a step, we know that $v_P = \langle h, v'_P \rangle$ and $v_V = h'$ for some h, h', v'_P. Since these terms agree by assumption, we know that $h = h'$. Each takes a step in its respective mode, producing terms v, v'_P, and s_0, respectively, where we must have that $\pi_v = [s_0] \mathbin{@} \pi'_v$ and $\pi'_p = \pi_p \mathbin{@} [\; \llparenthesis v'_P \rrparenthesis \;]$. Suppose $\vdash v \;\; v'_P \;\; s_0 : \tau$. Then we must show that $s_0 = \llparenthesis v'_P \rrparenthesis$, but this holds by Lemma 1-1. On the other hand, suppose that $s_0 = \llparenthesis v'_P \rrparenthesis$. We can prove $\vdash v \;\; v'_P \;\; s_0 : \tau$ as follows. By inversion on $\vdash e \;\; e_P \;\; e_V : \tau$ we know that $\vdash v \;\; \langle h, v'_P \rangle \;\; h : \bullet\tau$ and by inversion on this we know $\vdash v \;\; v'_P \;\; \llparenthesis v'_P \rrparenthesis : \tau$. But this is the desired result since $s_0 = \llparenthesis v'_P \rrparenthesis$. □

Finally, we demonstrate that terms in agreement can take a step to terms that no longer agree if and only if an adversary has managed to find a hash collision.

Lemma 6 (Security). *Given the following:*

1. $\vdash e \;\; e_P \;\; e_V : \tau$
2. $\ll \pi, e \gg \to_I \ll \pi, e' \gg$
3. $\ll \pi_p, e_P \gg \to_P \ll \pi'_p, e'_P \gg$
4. $\ll \pi_v, e_V \gg \to_V \ll \pi'_v, e'_V \gg$

Then $\not\vdash e' \;\; e'_P \;\; e'_V : \tau$ if and only if there exists a pair s and $s^\dagger$ such that $\pi'_p = \pi_p \mathbin{@} [s]$ and $\pi_v = [s^\dagger] \mathbin{@} \pi'_v$ with $s \neq s^\dagger$ but $hash\, s = hash\, s^\dagger$; i.e., s and $s^\dagger$ constitute a hash collision.

Proof. By induction on $\vdash e \;\; e_P \;\; e_V : \tau$. Most cases are vacuous because evaluation yields $\pi_p = \pi'_p$ and $\pi_v = \pi'_v$ which implies $\vdash e' \;\; e'_P \;\; e'_V : \tau$ by Preservation. The remaining cases are for let binding and $unauth$. The former follows by induction. For the latter we have e, e_P, and e_V are $unauth\;\; v$, $unauth\;\; v_P$ and $unauth\;\; v_V$, respectively. Since each takes a step, we know that $v_P = \langle h, v'_P \rangle$ and $v_V = h'$ for some h, h', v'_P. Since these terms agree by assumption, we know that $h = h'$. Each takes a step in its respective mode, producing terms v, v'_P, and s_0, respectively, where we must have that $\pi_v = [s_0] \mathbin{@} \pi'_v$ and $\pi'_p = \pi_p \mathbin{@} [\; \llparenthesis v'_P \rrparenthesis \;]$ and $hash\, s_0 = h$. Suppose $\not\vdash v \;\; v'_P \;\; s_0 : \tau$. By inversion on $\vdash e \;\; e_P \;\; e_V : \tau$ we know that $\vdash v \;\; \langle h, v'_P \rangle \;\; h : \bullet\tau$ and by inversion on this we know $\vdash v \;\; v'_P \;\; \llparenthesis v'_P \rrparenthesis : \tau$ and $hash\, \llparenthesis v'_P \rrparenthesis = h$. But then, by Lemma 1-2, for all v_{P0} and v_{V0} such that $\vdash v \;\; v_{P0} \;\; v_{V0} : \tau$ we must have $v_{P0} = v'_P$ and $v_{V0} = \llparenthesis v'_P \rrparenthesis$. So the reason we cannot prove $\vdash v \;\; v'_P \;\; s_0 : \tau$ must be that $s_0 \neq \llparenthesis v'_P \rrparenthesis$ while $hash\, s_0 = h = hash\, \llparenthesis v'_P \rrparenthesis$. Conversely, suppose that $s_0 \neq \llparenthesis v'_P \rrparenthesis$. But then Lemma 1-2 and $\vdash v \;\; v'_P \;\; \llparenthesis v'_P \rrparenthesis : \tau$ implies that we cannot prove $\vdash v \;\; v'_P \;\; s_0 : \tau$ unless $s_0 = \llparenthesis v'_P \rrparenthesis$. □

5. Implementation

This section describes our prototype extension to the OCaml compiler for supporting authenticated types. We discuss the basic approach to compilation[5], two optimizations we implement, and current limitations.

[5] Our technique for extending the OCaml compiler is based on a 2012 blogpost by Jun Furuse: `https://bitbucket.org/camlspotter/compiler-libs-hack`

المنارة للاستشارات
www.manaraa.com

5.1 Compilation

The compilation process works as follows. The programmer writes an OCaml program p like that of Figure 2 that contains uses of authenticated types. The type $\bullet\alpha$ is declared abstract, as α authtype, in the signature of the ADS module, which also declares the (polymorphic) $auth$ and $unauth$ coercions which p will import. (In what follows, we continue to write $\bullet\tau$ rather than τ authtype, for legibility.) Program p is then passed to our extended compiler which, depending on a command-line *mode* flag, replaces each application of $auth$ and $unauth$ it finds with a call to a prover- or verifier-specific implementation. The result is output and linked with the ADS module implementation.

The ADS module, given in Figure 11, defines type $\bullet\alpha$ as either a digest (just the hash, represented as a string), or as a pair of the hash and a value of type α. The next four functions define the prover's and verifier's versions of $auth$ and $unauth$, respectively; the calls to $auth$ and $unauth$ will be replaced by calls to these functions instead. We can see that their code largely matches the operational rules given in Figure 8, where the proof stream from the rules is implemented as OCaml channels, prf_output and prf_input. The one departure is that auth_prover and unauth_prover additionally take a function shallow that is invoked to perform the shallow projection operation. This operation is needed because OCaml does not provide a generic method for folding/mapping over the structure of a term. As such, our compiler generates type-specific shallow projection functions where needed, and includes them in the replaced calls to $auth$ and $unauth$. The type of the shallow projection operator is determined by the concrete type inferred at each $auth/unauth$ call. For example, the $unauth$ in **let** x : int = $unauth$ y is inferred to have type $\bullet$int $\rightarrow$ int. Therefore, we need a shallow projection operation of type int $\rightarrow$ int (which is just the identity). The generated code will refer to the ADS module's shallow_$\bullet$ function, shown at the bottom of the figure, for handling (nested) authenticated values.

Figure 12 shows the result of compiling a variant of authenticated binary search trees. The top of the figure is the code provided by the programmer. Compilation will replace the call to $auth$ with a call to auth_bst1 and the call to $unauth$ with a call to unauth_bst. These functions are defined at the bottom of the figure, and employ the needed, type-specific shallow projection operations.

The hash function referenced in Figure 11 is polymorphic, having type $\forall\alpha.\alpha \rightarrow$ string. It is implemented by first serializing the argument and then hashing it using SHA1 (which is widely considered to be collision-resistant). For serialization, we use OCaml's default serializer implemented in the Marshal module. This choice has an implication for security: the worst-case cost to compute the hash of a malicious string is bounded only by the representation of an integer in OCaml, either 32 or 64 bits, depending on the OS.

5.2 Optimizations

Our compiler implements two optimizations that reduce the size of the proof stream. The first optimization implements a *reuse buffer*. The idea is that we can reduce the size of the proof stream, and speed up proving/verification, when we anticipate that the same elements may appear in it more than once. For example, if a client submits a batch of operations to the server, it may end up traversing the same nodes of the ADS multiple times, and the client could cache these elements locally instead of reading them from the proof stream each time.

This optimization requires modifying the $unauth$ and $auth$ functionality; the modification is simlar for both prover and verifier. A counter is incremented each time $auth$ or $unauth$ is called. Each party maintains two data structures: *LRU-Map*, a mapping from $auth/unauth$ counter values to shallow projections, indicating the least-recently-used ordering, and *Digest-Map*, a mapping

```
type •α = | Shallow of string (* the digest *)
          | Merkle of string × α

let auth_prover (shallow: α → α) (v:α) : •α =
  Merkle(hash (shallow v), v)

let unauth_prover (shallow: α → α) (v:•α) : α =
  let Merkle(_,x) = v in
  to_channel !prf_output (shallow x);
  x

let auth_verifier (v:α) : •α = Shallow(hash v)

let unauth_verifier (v:•α) : α =
  let Shallow(h) = v in
  let y = from_channel !prf_input in
  assert h = hash y;
  y

let shallow_• (Merkle(h,_): •α) : •α = Shallow(h)
```

Figure 11. The implementation of type constructor $\bullet$ and the Prover and Verifier's $unauth$ and $auth$ coercions.

```
(* User-provided code *)
type bst = Tip
  | Bin of •bst × int × •bst
  | AuthBin of •(bst × int × bst)
let is_empty (t:•bst) : bool = (unauth t = Tip)
let mk_leaf (x:int) : •bst = AuthBin(auth(Tip, x, Tip))

(* Generated Prover code *)
let rec shallow_bst : bst → bst = function
  | Tip → Tip
  | Bin (x, y, z) → Bin(shallow_• x, y, shallow_• z)
  | AuthBin (x) → AuthBin (shallow_bst1 x)
and shallow_bst1 : bst × int × bst → bst × int × bst
= function (x, y, z) → (shallow_bst x, y, shallow_bst z)

let unauth_bst = unauth_prover shallow_bst
let auth_bst1 = auth_prover shallow_bst1
```

Figure 12. Example types and generated code (for prover)

from digests to $auth/unauth$ counter values; collectively *Digest-Map* and *LRU-Map* are referred to as "the cache," as they contain corresponding elements. When $unauth$ is called, if the digest exists in *Digest-Map*, the prover omits (resp., verifier fetches from the cache) the corresponding shallow projection, then updates the counter value associated with it in the cache; otherwise, the prover appends it to (resp., verifier reads it from) the proof stream and adds it to the cache. If the size of the cache exceeds the parameter, then the least recently used element (the smallest key in *LRU-Map*) is removed. The proof generation/checking/size benefits come at the cost of having to store the cache.

The second optimization we call *suspending disbelief*. It eliminates redundancy in shallow projections that contain hashes computable from nested shallow projections appearing subsequently in the proof stream. This optimization is implicit in the verification procedure for Merkle trees given in Section 2.1. We can approximate the optimization for arbitrary data structures in $\lambda\bullet$ by using a buffer to "suspend disbelief." This optimization requires modifications to $unauth$ for prover and verifier.

المنارة للاستشارات
www.manaraa.com

For the prover, the goal is to decide which digests in a shallow projection can safely be omitted, which are those that correspond to nodes that will be visited during subsequent calls to $unauth$. To achieve this, we extend the representation for $\bullet\alpha$ values:

type $\bullet\alpha$ = ... | MerkleSusp of string $\times$ α $\times$ bool ref
| Sentinel

Each time $unauth$ is called, the shallow projection is computed differently: Merkle(d,a) that would ordinarily be replaced with Shallow(d) is instead replaced with MerkleSusp(d,a,flag), where flag is a fresh mutable flag (initially **false**) that indicates whether the digest can be omitted. When one of the immediate children is accessed by $unauth$ the flag is set. Rather than writing the shallow projection immediately to the proof stream, it is appended to a queue. The queue is flushed when execution reaches a programmer-inserted insist annotation. This is a hint that belief need no longer be suspended; its placement has no effect on security but it is best used at the end of a distinct operation. Before a queue element is added to the proof stream, occurrences of MerkleSusp(d,a,flag) are (functionally) replaced with Sentinel when !flag is **true**, and replaced with Shallow(d) when !flag is **false**.

When the verifier encounters a shallow projection object in the proof stream containing Sentinel values, its digest cannot be computed immediately so the object is stored in a set referred to as the *suspended-disbelief* buffer. If $unauth$ is subsequently called on an immediate child corresponding to a Sentinel, the root hash is unknown so it cannot be immediately validated either. Thus we extend the $\bullet\alpha$ representation again with a new tag, Suspension. For every object placed in the buffer, each immediate Sentinel is replaced with a Suspension containing a callback closure and a mutable reference (initially empty) optionally containing a hash; when a leaf node is accessed, the Suspension reference is updated with the actual digest of the leaf, and the callback is invoked. The callback encapsulates a pointer to the parent node, and checks if all immediate Suspension children have been populated with digests; if so, the callback removes the node from the buffer, and either validates the node against digest, or recursively invokes the callback closure associated with the parent. Thus validation propagates upwards from the leaves to the root.

type $\bullet\alpha$ = ... | Suspension **of** string ref $\times$ (unit $\rightarrow$ unit)

A caveat of this optimization is that it exposes the verifier to potential resource exhaustion attacks, as (potentially infinite) computation is performed on untrusted data before it is validated. A solution would be to bound the number of steps the program should take before the buffer is cleared. Another caveat is that this optimization requires additional storage on the verifier in contrast to the original Merkle tree optimization.

5.3 Supporting full OCaml

Our compiler prototype supports authenticating the OCaml equivalent of the type language given in Figure 5 with the exception of function types. This support has been sufficient to program a variety of interesting data structures, as described in the next section.

To implement authenticated functions requires that we be able to perform the shallow projection of a lambda term. Our formalism does this by folding over the syntax of the lambda term's body to find authenticated values $\langle h, v\rangle$ and replace them with values h. In an implementation this operation is tantamount to transforming a closure's environment. We could do this quite naturally using Siskind and Pearlmutter's map-closure operator [26], but unfortunately (as they point out) it is not clear how to implement this operator in a statically typed language, since the compiler cannot, in general, know the types of a given closure's environment variables. We could imagine storing type information with the closure (e.g., Crary et al's [3] term representations for types) in support of a generic, run-time shallow projection operation as per the formalism. In the meantime, the most natural use of authenticated closures we have found is to support shallow CPS transformations; we can use an explicit stack to the same effect, as shown in Figure 4.

Among other OCaml features not yet supported, the most desirable is authenticated polymorphic types. Similarly to closure environments, the compiler cannot know types needed to perform shallow projection—if a value given to $auth$ and $unauth$ is polymorphic, then its type is determined by how type variables are instantiated at a particular call-site. Once again, a generic shallow projection operator as per our formalism (and easily implemented in a dynamically typed language) would fit the bill. We could imagine requiring that $auth$ and $unauth$ each take an additional type parameter; in most cases it could be statically determined, but to support polymorphism it could be passed as an argument, e.g., to the polymorphic function containing the $auth$/$unauth$ call.

6. Evaluation

To demonstrate the effectiveness and generality of our language and compiler, we have implemented a variety of ADSs. We analyze their performance and confirm it empirically with benchmarks for selected algorithms. Our benchmarks were conducted using an Amazon EC2 "m1.xlarge" instance (an Intel E5645 2.4GHz processor, with 16GB of RAM). All data structures were stored in RAM. Complete code is given in the Appendix.

Merkle trees. Our version of Merkle trees was given in Figure 2. As Merkle trees are the most common authenticated data structure, and are typically implemented by hand, we compared the running time of our compiled verifier routine to hand-written implementations in OCaml and in C (see Figure 13(c)). The benchmark consists of 10,000 random accesses to trees of height h for $h \in [5, 19]$. Each array element is a 1024 byte string; thus the largest tree contains approximately $250k$ elements and stores approximately $250MB$ of data in total. Compared to the hand-optimized version in C, the program generated by our compiler runs is slower by only a factor of two; the hand-written OCaml code is about 30% slower than the C program. Profiling with gprof reveals that substantial overhead is due to the Marshal serialization routine used by our compiler; the hand-written OCaml code avoids this serialization by concatenating the child digests directly.

Red-black+ Trees. A red-black+ tree is a self-balancing binary tree—the + indicates that internal nodes only store keys, and the values are stored only in the leaves. This data structure is appropriate for a mutable dictionary. We consider authenticated search trees to be the second oldest authenticated data structure, proposed by Naor and Nissim [20] to implement certificate revocation lists. Our results are asymptotically equivalent: the storage cost to the prover for the entire data structure is $O(n)$, while the computation cost per operation for both prover and verifier is $O(log\, n)$; the size of the proof stream is also $O(log\, n)$. Note that we also implement an authenticated version of normal binary search trees, too, and these also have the expected performance.

In Figure 13(a), we show the empirical runtime performance of the authenticated red-black+ tree for both the prover (P) and verifier (V) modes, as well the Ideal mode with the $\bullet$ annotations and $unauth$/$auth$ commands erased, to show the overall computational overhead of authentication. The benchmark consists of $100,000$ random insertions into a random tree containing 2^k elements, for each $k \in [4, 21]$. P runs approximately 25% faster than V; this is because V computes a hash during both $unauth$ and $auth$ instructions whereas P only computes a hash during $auth$. According to profiler analysis (using gprof), V spends 55% of its time in the SHA1 hash routine and 30% in (de)serialization routines;

المنارة للاستشارات
www.manaraa.com

P spends 28% of its time computing SHA1, 30% performing serialization and 22% in garbage collection. The overhead of P is approximately a factor of 100 compared to the ordinary data structure.

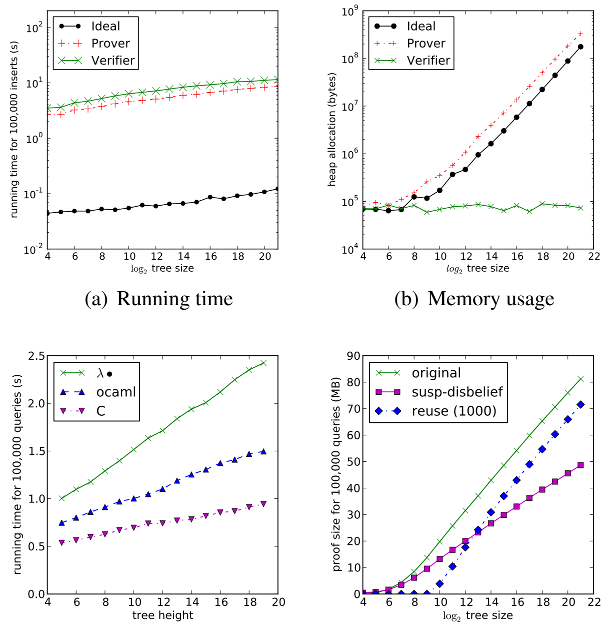
We measured the largest amount of memory allocated by OCaml during this benchmark as shown in Figure 13(b). This illustrates the key advantage of an authenticated data structure scheme: while P incurs space overhead by a factor of 3 vs the Ideal mode, the space requirement of the V is effectively constant.

Finally, we used this benchmark to evaluate the effectiveness of our two compiler optimizations on proof size (see Figure 13(d)). For a binary tree, the suspended-disbelief buffer results in a proof-size reduction of almost 50%, since only one of the left or right child digests must be transmitted for each node. The reuse-cache is most effective when the tree is small and mostly fits in the cache; thus since the cache-size parameter in our benchmark is 1000, the proof is nearly empty up to trees of height 9. However, because the benchmark consists of random queries, the leaves and nodes toward the bottom are accessed almost uniformly at random, so only a constant number of nodes near the root are read from the cache each query. The cache would be more beneficial in applications where some nodes were accessed much more frequently.

Skip Lists. Skip lists [25] are randomized data structures providing similar performance (in expectation) to binary search trees. Random algorithms can be used in $\lambda\bullet$ as long as P and V both use the same pseudorandom function and seed. Our results are asymptotically equivalent to previous work on authenticated skip lists [10]: the storage cost to P for the entire data structure is $O(\log n)$, where n is the number of elements inserted, and the *expected* computational cost to P and V as well as the size of the proof stream is $O(\log n)$ while the *worst-case* in time and space is $O(n)$.

Planar Separator Trees. Planar separator trees are data structures that can be used to efficiently query the distance of the shortest path between two vertices in a planar graph (e.g., many road maps) [5]. A separator of a graph is a collection of vertices inducing a binary partition on the graph, such that any path from a vertex in one partition to a vertex in the other partition must pass through a vertex in the separator. A consequence of them planar separator theorem is that every planar graph has a separator that is *small* ($O(\sqrt{n})$, where n is the number of vertices), yet induces a balanced partition (both partitions have at least $\frac{n}{3}$ elements); a search structure can be built over the graph by recursively constructing separators for each partition. While the naïve solution of storing the shortest distance between every pair of vertices requires $O(n^2)$ storage, the planar separator tree requires only $O(n^{\frac{3}{2}})$ space. The shortest distance between any two points can be computed in $O(\sqrt{n}\log\, n)$ time using this data structure. A potential application of an authenticated planar separator tree is for a mobile device user to query a service provider for travel directions, along with a proof that the response is actually the shortest path (rather than, e.g., one that routes the user out of their way past, past billboards for which the service provider might receive a profit). Using our authenticated compiler, the proof size and computation cost for P and V is also $O(\sqrt{n}\log\, n)$. We include this primarily as an example of a data structure that has not been treated as an authenticated data structure in prior work, but is straightforward to authenticate using our framework.

Bitcoin. Bitcoin [18] is a peer-to-peer network implementing a virtual currency; it features an authenticated data structure called the *blockchain*, which represents a history of transactions. A global ledger can be computed from the blockchain, which, abstractly speaking, contains a set of currently valid coins. A valid transaction removes a past coin from the ledger and adds a new one (assigned to the recipient of the coin).



(a) Running time

(b) Memory usage

(c) our $\lambda\bullet$ compiler vs hand-optimized code

(d) proof size with optional optimizations

Figure 13. Empirical performance evaluation results. For 100,000 insertions into a red-black+ tree, (a) running time for Prover, Verifier, and Ideal; (b) memory usage (heap size). (c) Running time for fetch operation in a balanced merkle tree, for the program generated by our compiler, hand-written OCaml code, and hand-written C code; For 100,000 random insertions in a binary search tree, (d) the effect of two optional optimizations on proof stream size, compared with two optional optimizations: a reuse-cache (of size 1000) and the belief-suspension buffer.

The current Bitcoin data structure suffers from two drawbacks: 1) a miner (verifier) needs to maintain an entire local copy of the global ledger to efficiently perform validation of transactions, which may become unscalable as the number of coins increases; and 2) at install time, a new client needs to download the entire transaction history and perform linear computation (in the size of the block chain) to construct the ledger even when it trusts the root digest.

These drawbacks can be resolved if we build the ledger (as an authenticated set) into the block chain data structure. We have done this in our language by composing the blockchain data structure with a ledger consisting of our authenticated red-black+ tree, thereby reducing client storage to $O(\log\ M)$, where M bounds the number of coins in the ledger. Full details are given in the Appendix

7. Related Work

As mentioned earlier, authenticated data structure research has had a flurry of results [16, 19, 13, 1, 10, 12] from the cryptography community, proposing authenticated versions of set (non)-membership, dictionaries, range queries, certain graph queries, B-trees, etc.

To the best of our knowledge, no one has offered a general authenticated data structure implementation for generic programs. Those closest work is by Martel *et. al.* [13], which proposes a method for designing authenticated data structures for a class of data structures referred to as “search DAGs.” Their model is limited to *static* data structures, i.e., does not support updates. Our

المنارة للاستشارات
www.manaraa.com

approach is also easier to use, since the programmer writes largely standard (purely functional) implementations of data structures and their operations, and the compiler generates the prover/verifier code; in their approach, the task of designing and implementing authenticated data structures still must be done by humans directly.

Our programmatically generated ADS implementations have performance competitive with known customized ADS constructions based on collision-resistant hashes. We note, however, that while most are, not all known ADS constructions are based on collision resistant hash functions. For example, bilinear-group based ADS constructions exist for set operations [23]. Using alternative algebraic primitives other than collision resistant hashes can sometimes yield asymptotically better ADS constructions.

Beyond the optimizations we have implemented (see Section 5.2), there are other known optimizations that we have not an included. An example is the technique of "commutative hashing" due to Goodrich et al. [10] which reduces the proof size when it is irrelevant which (of possibly several) child nodes are traversed. We believe it is likely that optimizations for specific data structures could be incorporated generally into our compiler. Other optimizations lie further outside our model; an example is a line of work beginning with Merkle's original paper [15, 2] in which the stored data is not arbitrary, but is instead generated algorithmically (e.g., using a pseudo-random number generator). In this case, the prover's can be significantly reduced by recomputing data on the fly rather than storing it. In our performance evaluation we consider memory usage and running time; however some work in authenticated data structures [17, 13] considers I/O efficiency, another practical characteristic.

Verified computation [7] and succint non-interactive arguments of knowledge (SNARKs) [8, 24] can also yield asymptotically better protocols for ensuring integrity of outsourced computation (e.g., with $O(1)$ amount of client computation other than reading the input and ouput). However, while theoretically attractive, known verified computation and SNARKs schemes are orders of magnitude more expensive than constructions based on collision-resistant hashes in practice [24] — partly due to the use of heavy-weight cryptographic primitives such as Fully Homomorphic Encryption.

8. Conclusions

We have presented $\lambda\bullet$, the first programming language for authenticated data structures (ADS). We have formally proven that every well-typed $\lambda\bullet$ program compiles to a secure protocol, and we have implemented $\lambda\bullet$ as a simple compiler extension to OCaml so that a programmer can easily derive an authenticated data structure from any ordinary one. The protocols generated by our compiler are competitive with the state-of-the-art in ADS based on collision-resistant hash functions.

We believe this work is long past-due. ADS are a 30-year-old technique in cryptography, and yet researchers have overlooked the simple connection between ADSs encapsulated in our notion of authenticated types. We plan to extend our language to be more expressive, and to include more efficient techniques based on advanced cryptographic primitives. We hope our work encourages ADS adoption in future secure cloud computing infrastructure.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *Proc. ISC*, pages 379–393, London, UK, UK, 2001. Springer-Verlag.

[2] P. Berman, M. Karpinski, and Y. Nekrich. Optimal trade-off for Merkle tree traversal. *Theor. Comput. Sci.*, 372(1):26–36, Mar. 2007.

[3] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP*, 1998.

[4] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *Data and Application Security*, pages 101–112. Springer, 2002.

[5] R. Duan. Planar separator theorem and its applications. lecture slides, Advanced Graph Algorithms, Summer 2012. max planck institut informatik. `http://www.mpi-inf.mpg.de/departments/d1/teaching/ss12/AdvancedGraphAlgorithms/Slides10.pdf`.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PLDI*, 1993.

[7] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO 2010*, pages 465–482. Springer, 2010.

[8] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Cryptology ePrint Archive, Report 2012/215, 2012. `http://eprint.iacr.org/`.

[9] M. T. Goodrich, C. Papamanthou, and R. Tamassia. On the cost of persistence and authentication in skip lists. In *Proc. Intl. Workshop on Experimental Algorithms*, volume 4525 of *LNCS*, pages 94–107. Springer, 2007.

[10] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proc. DARPA Information Survivability Conference and Exposition II (DISCEX II)*, pages 68–82, 2001.

[11] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[12] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios. Proof-infused streams: Enabling authentication of sliding window queries on streams. In *VLDB*, pages 147–158, 2007.

[13] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authentic data publication, 2001. Available from `http://www.cs.ucdavis.edu/~devanbu/files/model-paper.pdf`.

[14] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A General Model for Authenticated Data Structures. *Algorithmica*, 39(1):21–41, Jan. 2004.

[15] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, Apr. 1978.

[16] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.

[17] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, 2006.

[18] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, unpublished, 2009.

[19] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. USENIX*, pages 217–228, Berkeley, 1998.

[20] M. Naor and K. Nissim. Certificate revocation and certificate update. *IEEE J. on Sel. Areas Commun.*, 18(4):561–570, 2000.

[21] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 437–448. ACM, October 2008.

[22] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal authenticated data structures with multilinear forms. In *Proc. Int. Conference on Pairing-Based Cryptography (PAIRING)*, pages 246–264, 2010.

[23] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

[24] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proc. IEEE SSP*, 2013.

[25] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[26] J. M. Siskind and B. A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *POPL*, 2007.

[27] R. Tamassia. Authenticated data structures. In *11th Annual European Symposium on Algorithms*, Sept. 2003.

المنارة للاستشارات
www.manaraa.com

```
type tree =
  | Tip
  | Bin of •tree × int × •tree

let rec member (t:•tree) (x:int) : bool =
    match unauth t with
     Tip → false
   | Bin (l,y,r) →
      if y = x then true
      else if x < y then member l x
      else member r x

let rec insert (t:•tree) (x:int) : •tree =
    match unauth t with
     Tip → auth (Bin(auth Tip,x, auth Tip))
   | Bin (l,y,r) →
      if y = x then t
      else if x < y then auth (Bin(insert l x,y,r))
      else auth (Bin(l,y,insert r x))
```

Figure 14. Binary Search Trees (implementing a set of ints)

A. Code Examples

Here we present the complete code for the ADSs we have implemented, as discussed in Section 6. The next section presents an exploration on possible improvements to ADSs used in Bitcoin [18].

In particular, we implement three datastructures providing a set semantics, supporting membership queries and insertions:

- FIgure 14 presents (unbalanced) binary search trees.
- Figure 15 presents red-black+ trees.
- Figure 16 presents skip lists.

Figure 17 presents a *planar-separator tree* for finding the distance of the shortest path between two points in a planar graph.

B. Bitcoin case study

Bitcoin [18] is a peer-to-peer network that implements virtual currency as a replicated ledger. Bitcoin's core data structure, the *blockchain*, is essentially an authenticated structure, and as such we can model it in our language; however, the key operation, *blockchain validation*, is suboptimal, and we can easily define an improvement in our language via auxiliary authenticated data structures.

Bitcoin participants (called *miners*) vote to establish a canonical, linearized sequence of updates (called *transactions*) to the ledger. Transactions are grouped into batches and stored in *blocks*. Each block additionally stores a collision-resistant hash of the previous block, thus forming a *blockchain*; the hash of the most recent block can thus be seen as the root digest of an authenticated list containing all the transactions so far. A key requirement is that participants only vote on updates that are *valid* according to certain rules depending on the current state of the ledger. Therefore the blockchain must support the following query operation: does the root digest of a blockchain represent a *valid* sequence of transactions? When a new Bitcoin node joins the network, or rejoins after some interruption, it downloads the sequence of blocks and replays each update; both the transaction history and the ledger are stored locally. We are interested in reducing the client's local space requirements; for example, a user may wish to validate the blockchain using a trusted hardware device (e.g., a smart card) that only has a small amount of integrated storage. At the time of this writing, the entire transaction history is approximately 8 gigabytes and the ledger takes up 250 megabytes.

```
type color = R | B
type voption = Non | Som of string
type tree = Tip
      | Bin of color × •tree × (int × voption) × •tree

let lookup x =
  let rec look t = match unauth t with
  | Tip → None
  | Bin (_,_,(y, Som v),_) → if y = x then Some v else None
  | Bin (_,l,(y, Non ),r) → if x <= y then look l else look r
  in look

let balanceL t = match unauth t with
| Bin (B, l, a, r) → begin match unauth l with
  | Bin (R, l1, a1, r1) → begin match unauth l1 with
    | Bin (R, l2, a2, r2) → auth(Bin(R,
        auth(Bin(B,l2,a2,r2)),a1,
        auth(Bin(B,r1,a,r))))
    | _ → begin match unauth r1 with
      | Bin (R, l2, a2, r2) → auth(Bin(R,
        auth(Bin(B,l1,a1,l2)),a2,
        auth(Bin(B,r2,a,r))))
      | _ → t
    end
  end
  | _ → t
end
| _ → t

(* balanceR (omitted) is analogous *)

let insert x v t =
  let leaf = auth(Bin(B,auth Tip,(x,Som v),auth Tip)) in
  let blacken Bin(_, l, x, r) = Bin(B, l, x, r) in
  let rec ins t = match unauth t with
  | Tip → leaf
  | Bin(c,l,(y,Som _),r) →
    if x = y then failwith "duplicate insert" else
    if x < y then auth(Bin(R,leaf,(x,Non),t)) else
    if x > y then auth(Bin(R,t,(y,Non),leaf)) else
    assert false
  | Bin(c,l,(y,Non as yv),r) →
    if x = y then t else
    if x < y then balanceL(auth(Bin(c,ins l,yv,r))) else
    if x > y then balanceR(auth(Bin(c,l,yv,ins r)))
    else assert false
  in auth(blacken(unauth(ins t)))
```

Figure 15. Red-black+ tree (keys are ints, values are strings)

www.manaraa.com

```
type ord = Sentinel | Key of int
type skiplist =
  | Nil    | Node of •(skiplist × ord × skiplist)
let empty = Node(auth(Nil, Sentinel, Nil))

let rec member t x =
  match t with
    | Nil → false
    | Node a → let (down,y,right) = unauth a in
      if Key x = y then true else
      if Key x < y then false else
      if member right x then true else
      member down x

let flip() = Random.bool()

let insert t x =
  let rec ins t : skiplist * skiplist option = match t with
  | Nil → failwith "uninitialized skiplist"
  | Node a → let (down, y, right) = unauth a in
    (* invariant: x >= y, because of Sentinel *)
    let godown() = match down with
    | Nil → let leaf = Node (auth(Nil, Key x, right)) in
      Node (auth(down, y, leaf)), Some leaf
    | Node _ → match ins down with
      | down1, None → Node (auth(down1, y, right)), None
      | down1, Some leaf → if flip()
        then Node (auth(down1, y, right)), None
        else let leaf1 = Node (auth(leaf, Key x, right)) in
          Node (auth(down1, y, leaf1)), Some leaf
    in match right with
      | Nil → godown()
      | Node a → let (_,z,_) = unauth a in
        if Key x = z then failwith "duplicate insert"
        else if Key x ¡ z then godown()
        else let right1, fin = ins right in
          Node (auth(down, y, right1)), fin
        in let rec grow t leaf = if flip() then t
        else let leaf1 = Node (auth(leaf, Key x, Nil)) in
          grow (Node(auth(t, Sentinel, leaf1))) leaf1
  in match ins t with
  | t1, None → t1
  | t1, Some leaf → grow t1 leaf
```

Figure 16. Skiplist

```
type distance = float

type separator =
  | Tip ×
  | Bin of separator × int × distance list × separator

type sep_tree =
  | STip
  | SBin of sep_tree × separator × sep_tree

let rec member key tree = match tree with
  | Tip → false
  | Bin (l, k, _, r) →
    if (key = k) then true
    else if (key ¡ k) then member key l
    else member key r

let rec lookup key tree = match tree with
  | Bin (l, k, a, r) →
    if (key = k) then a
    else if (key ¡ k) then lookup key l
    else lookup key r

let rec smember w tree = match tree with
  | STip → false
  | SBin (_, sep , _) → member w sep

let shortest_across u v sep =
  let dists = List.map2 (fun x y → x+.y)
    (lookup u sep) (lookup v sep) in
  List.fold_right min dists infinity

let rec shortest u v tree = match tree with
  | SBin (l, sep, r) →
    let withinL = if smember u l && smember v l
     then shortest u v l else infinity in
    let withinR = if smember u r && smember v r
     then shortest u v r else infinity in
    let across = shortest_across u v sep in
    min across (min withinL withinR)
```

Figure 17. Planar-Separator Tree (for finding the distance of the shortest path between two points in a planar graph).

In our simplified Bitcoin model (see Figure 18), the ledger is simply a set of integers (in actuality, the ledger associates quantities of currency units to cryptographic public keys, entitling the owner of the corresponding private key to spend that currency). A transaction consists of a collection of elements to remove, and a collection of elements to insert. We assume that each block contains only a single transaction, the number of updates per transaction is constant, the size of the ledger at any time is bounded by m, and the number of blocks is n. We consider the operation of validating the entire blockchain since the initial "genesis" block (the ledger is initially an empty set). Our first validation routine (see Figure 18, Variation 0) first traverses the blockchain from the most recent block to the initial block, and then replays each transaction in temporal order, at each step updating a local copy of the ledger. This operation requires $O(m + n)$ space for the V: m for the local copy of the ledger and n because the recursive operation is not tail recursive, and so the stack grows to size n. The computational time for V is $O(n \log m)$ because the cost of each update to the ledger (implemented as a search tree) is $O(\log m)$. We can modify the algorithm to be tail recursive in our framework by adding an authenticated explicit stack (Variation 1), thus reducing the V's memory requirement to $O(m)$. [6] Finally, by replacing the ledger with an authenticated set data structure (e.g., as a red-black tree, as in (Variation 2)), V's space requirement is reduced to only $O(\log m)$. [7] The computational cost for P and V is asymptotically the same ($O(n \log m)$) for each of our variations. However, the size of the proof stream for Variation 2 ($O(n \log m)$) is larger than for Variations 0 and 1 ($O(n)$).

[6] An ordinary Bitcoin client does not actually validate the blockchain using a recursive function requiring an $O(n)$ size stack; however its storage cost is asymptotically equal to our Variation 0 because it stores the entire blockchain so that it can act as prover for other nodes. The ordinary Bitcoin client could be trivially modified to match the performance of our Variation 1 by discarding each block after processing it.

[7] If necessary, this could be further reduced to $O(1)$ by using another explicit stack for set update operations.

المنارة للاستشارات
www.manaraa.com

```
(* Variation 0: Ordinary blockchain validation *)
type coin = int
type transaction =
      coin list (* coins to remove *) ×
      coin list (* coins to insert *)
type ledger = IntSet.t (* Built-in set *)
type block = Genesis | Block of • block × • transaction
let apply tx ldgr =
   let after_remove = List.fold_right
      (IntSet.remove) (fst tx) ldgr in
   let after_insert = List.fold_right
      (IntSet.add) (snd tx) after_remove in
   after_insert
let rec validate (blk : •block) : ledger =
   match unauth blk with
   | Genesis → IntSet.empty
   | Block(prevblk, tx) →
      let ldgr = validate prevblk in
      apply (unauth tx) ldgr

(* Variation 1: tail recursive with authenticated stack *)
type stack = E | S of •(•transaction × stack)
let rec build s blk = match unauth blk with
   | Genesis → s
   | Block(prevblk, tx) → build (S(auth(tx,s))) prevblk
let validate_tr blk =
   let stk = build E blk in
   let rec _validate s ldgr =
      match s with
      | E → ldgr
      | S txs → let tx, s = unauth txs in
       _validate s (apply (unauth tx) ldgr)
      in _validate stk IntSet.empty

(* Variation 2: authenticated ledger *)
type ledger2 = • Redblack.tree
let apply2 tx ldgr : ledger2 =
   let after_remove =
      List.fold_right (Redblack.delete) (fst tx) ldgr
   in let after_insert =
      List.fold_right (Redblack.add) (snd tx) after_remove
   in after_insert let rec validate_auth newblk =
   let stk = build E newblk
   let rec _validate s dgr =
      match s with
      | E → ldgr
      | S txs → let tx, s = unauth txs in
       _validate s (apply2 (unauth tx) ldgr)
   in _validate stk Redblack.empty
```

Figure 18. Bitcoin blockchain validation model

www.manaraa.com