Logical Relations for Formally Verified Authenticated Data Structures

Simon Oddershede Gregersen
New York University
New York, NY, USA
s.gregersen@nyu.edu

Chaitanya Agarwal
New York University
New York, NY, USA
ca2719@nyu.edu

Joseph Tassarotti
New York University
New York, NY, USA
jt4767@nyu.edu

Abstract

Authenticated data structures allow untrusted third parties to carry out operations which produce proofs that can be used to verify an operation's output. Such data structures are challenging to develop and implement correctly. This paper gives a formal proof of security and correctness for a library that generates authenticated versions of data structures automatically. The proof is based on a new relational separation logic for reasoning about programs that use collision-resistant cryptographic hash functions. This logic provides a basis for constructing two semantic models of a type system, which are used to justify how the library makes use of type abstraction to enforce security and correctness. Using these models, we also prove the correctness of several optimizations to the library and then show how optimized, hand-written implementations of authenticated data structures can be soundly linked with automatically generated code. All of the results in this paper have been mechanized in the Rocq prover using the Iris framework.

CCS Concepts

• **Security and privacy → Logic and verification**; • **Theory of computation → Separation logic**.

Keywords

Authenticated data structures; security; hash functions; Merkle trees; logical relations; separation logic

ACM Reference Format:
Simon Oddershede Gregersen, Chaitanya Agarwal, and Joseph Tassarotti. 2025. Logical Relations for Formally Verified Authenticated Data Structures. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744801

1 Introduction

An *authenticated data structure* (ADS) [43] is a type of data structure in which operations produce cryptographic proofs that can be used to check that the results of the operation are valid. ADSs can be used in scenarios where computations are outsourced to untrusted third parties, where the results of the computation can be verified by checking the proofs that are produced. For example, Merkle trees [29] are a binary tree data structure in which internal nodes of the tree contain hashes of their children. A retrieve operation on a Merkle tree produces a proof consisting of a list of hashes of nodes. A verifier that only knows the hash of the root node of the tree is able to check the result of the retrieve by re-computing what the root hash would be from the proof and comparing the result to the actual root hash.

Because ADSs are used for security-critical applications, it is essential that they are correctly implemented. In particular, adversaries must not be able to construct false proofs that trick a verifier into accepting an invalid result. However, correctly developing and implementing ADSs is challenging. Moreover, it is difficult to test ADS implementations to ensure that adversarially-constructed invalid proofs are rejected.

To ease the task of developing ADSs, Miller et al. [30] develop $\lambda\bullet$, an extension to the OCaml programming language for implementing ADSs that use *cryptographic hash functions* to construct proofs. The $\lambda\bullet$ language extends OCaml with a new class of *authenticated types*, which describe data that comes with an accompanying cryptographic proof. Using $\lambda\bullet$, a developer writes a standard implementation of a data structure, and then the compiler automatically generates code for an authenticated version. Specifically, the $\lambda\bullet$ compiler converts a program into two different executables, called the *prover* and *verifier* executable. The prover performs operations and generates cryptographic proofs corresponding to those operations. Meanwhile, the verifier takes a proof as input and checks whether it is valid or not. Experimental evaluations show that the $\lambda\bullet$ compiler can generate authenticated data structures with the same asymptotic running time as manually-written versions.

To justify the correctness of $\lambda\bullet$'s approach to generating authenticated data structures, Miller et al. [30] present a core calculus language for a subset of $\lambda\bullet$ and prove security and correctness properties of programs written in this language. This core calculus has separate formal operational semantics describing how the prover and verifier versions of a program should run. In these different semantics, primitive operations in the language generate or produce proofs. In addition, there is a third operational semantics for *ideal* execution, in which there are no proof objects and programs execute like a standard, simplified OCaml-like program. This ideal execution behavior is not intended to be run directly, but instead is used to specify how the prover and verifier should behave. In particular, all well-typed programs in this language satisfy two important properties, called security and correctness, which can be informally summarized as:

- **Security**: If the verifier version of a program accepts a proof p and returns a value v, then either the ideal execution of the program would have also returned v, or a hash collision must have occurred during the execution of the verifier.

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744801

- **Correctness**: If the prover version generates a proof p and returns a value v, then the ideal version would also return v, and the verifier will accept p and return v as well.

In particular, if the cryptographic hash function used in the authenticated data structure operations has (strong) collision resistance, then this security property ensures that it is hard for an attacker to construct a malicious proof that deceives a verifier.

However, there are three important limitations of the $\lambda\bullet$ approach. First, maintaining a custom compiler frontend imposes development burden. Second, to achieve efficient performance, the $\lambda\bullet$ compiler implements several optimizations related to how proof objects are stored and checked, but these optimizations are not covered by the security and correctness theorems, because the core calculus only models a simple, unoptimized version of proof object operations. Third, even with these compiler optimizations, the data structures generated by $\lambda\bullet$ are not always as efficient as hand-written versions. For example, Miller et al. report that the $\lambda\bullet$-generated verifier for a binary tree structure takes twice as long to check proofs as a hand-written C version.

In subsequent work, Atkey [5] addressed the first of these limitations by showing that $\lambda\bullet$'s authenticated types could be directly encoded in OCaml's existing type system, without needing to extend the language or use a custom compiler frontend. With this approach, a programmer uses OCaml's module system to write an implementation of a data structure that is *parameterized* by functions that handle proof generation and checking. Then, by linking the data structure code with three different libraries implementing these functions, one obtains the prover, verifier, and ideal versions of an authenticated data structure. Although this approach eliminates the need for a custom compiler, it raises its own challenges. Whereas Miller et al. were able to prove security and correctness of $\lambda\bullet$'s core calculus by directly analyzing its new authenticated types, Atkey's approach achieves security and correctness by using a *parametricity* property [34] of OCaml's module system. Atkey hints at how existing proofs of parametricity for type systems related to OCaml's module system could be used to prove security and correctness, but gives no formal proof. Moreover, existing parametricity theorems cannot be directly applied, since establishing security and correctness properties require reasoning about hash collisions, which is not covered.

In this paper, we provide for the first time a complete proof of security and correctness for generating authenticated data structures using a typed module system. Moreover, we also address the second and third limitations of the $\lambda\bullet$ approach described above. In particular, we prove the correctness of several optimizations for proof objects supported by $\lambda\bullet$'s compiler. Additionally, we show how to prove that hand-written, optimized implementations of operations on an authenticated data structure can be safely linked with code that is generated automatically. This allows a developer to combine the ease of automatic generation for most operations, while still being able to manually optimize certain operations that need to be as efficient as possible.

Our proof uses the technique of program-logic-based logical relations [11] which has been used in recent years to prove properties about strong guarantees provided by a range of sophisticated type systems, including data race freedom provided by Rust [24], type soundness of an extension of Scala's core type system [16], expressive information-flow control types [12, 21], and program refinement [13, 20, 44, 46–48]. With this technique, to prove that a type system guarantees a particular property, one starts by constructing a program logic, typically a variant of Hoare logic [23], that is expressive enough for proving that programs have the property in question. Next, one defines a *semantic model* of the type system, in which the meaning of a type is defined in terms of Hoare triples in the program logic. Using the rules of the logic, one proves that this model is sound by showing that whenever a program e is well-typed according to the rules of the type system, a corresponding Hoare triple holds for e. Such triples then guarantee that all well-typed programs have the desired property.

In our application of this technique, we construct a program logic called CF-SL for relational reasoning about programs that make use of collision-resistant hash functions (§3). Then, we construct two semantic models of a type system that can encode the module-based construction proposed by Atkey. The first model captures the security property of authenticated data structures (§4), while the second captures the correctness property (§5). Using these models, we then prove the correctness of several of the optimizations implemented in the $\lambda\bullet$ compiler (§6). Next, we show how an optimized, manual implementation of retrieval operations for a Merkle tree can be soundly linked with automatically generated code for other operations on this data structure (§7). Finally, we compare our approach to prior work on verifying the correctness of authenticated data structures (§9).

All of the results in this paper have machine-checked proofs of correctness carried out with the Rocq prover [45] using the Iris program logic framework [25]. The complete proof development is available as part of our artifact [19] and on GitHub at https://github.com/jtassarotti/veri-auth.

2 Background

This section begins by describing Atkey's module-based construction of authenticated data types. Next, we define the formal calculus and type system that we use to encode this construction.

2.1 Authentikit

Atkey makes the observation that it is possible to implement the functionality offered by the $\lambda\bullet$ programming language as a library *within* a general-purpose programming language with sufficiently powerful abstraction facilities, such as OCaml. The key idea is to express authenticated computations using an abstract monad that, depending on its instantiation, will either construct or consume proofs alongside the computation. The abstract interface of Atkey's library, called Authentikit, is given by the OCaml module signature shown in Figure 1a.

First, the signature postulates the existence of a type constructor `auth` that represents the type of authenticated values.[1] Note that `auth` is abstract and the interface does not commit to a particular choice for how authenticated types are represented.

Second, the interface requires programmers to structure authenticated computations using an *authenticated computation monad* given by the (abstract) type constructor `m` and the two operations

[1] This constructor corresponds to the authenticated type constructor "$\bullet$" found in $\lambda\bullet$.

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a m
  val return : 'a -> 'a m
  val bind   : 'a m -> ('a -> 'b m) -> 'b m

  module Authenticatable : sig
    type 'a evi
    val auth   : 'a auth evi
    val pair   : 'a evi -> 'b evi -> ('a * 'b) evi
    val sum    : 'a evi -> 'b evi ->
                 [`left of 'a | `right of 'b] evi
    val string : string evi
    val int    : int evi
  end

  val auth   : 'a Authenticatable.evi ->
               'a -> 'a auth
  val unauth : 'a Authenticatable.evi ->
               'a auth -> 'a m
end
```

(a) Module signature in OCaml.

$$
\begin{aligned}
\mathsf{AUTHENTIKIT} &\triangleq \exists \mathsf{auth}, \mathsf{m} : \star \Rightarrow \star.\ \mathsf{Authentikit}\ \mathsf{auth}\ \mathsf{m} \\
\mathsf{Authentikit} &\triangleq \lambda \mathsf{auth}, \mathsf{m} : \star \Rightarrow \star. \\
&\quad (\forall \alpha : \star.\ \alpha \to \mathsf{m}\ \alpha) \times \\
&\quad (\forall \alpha, \beta : \star.\ \mathsf{m}\ \alpha \to (\alpha \to \mathsf{m}\ \beta) \to \mathsf{m}\ \beta) \times \\
&\quad \mathsf{Authenticatable} \\
\mathsf{Authenticatable} &\triangleq \exists \mathsf{evi} : \star \Rightarrow \star. \\
&\quad (\forall \alpha : \star.\ \mathsf{evi}\ (\mathsf{auth}\ \alpha)) \times \\
&\quad (\forall \alpha, \beta : \star.\ \mathsf{evi}\ \alpha \to \mathsf{evi}\ \beta \to \mathsf{evi}\ (\alpha \times \beta)) \times \\
&\quad (\forall \alpha, \beta : \star.\ \mathsf{evi}\ \alpha \to \mathsf{evi}\ \beta \to \mathsf{evi}\ (\alpha + \beta)) \times \\
&\quad (\mathsf{evi}\ \mathsf{string}) \times \\
&\quad (\mathsf{evi}\ \mathsf{int}) \times \\
&\quad (\forall f : \star \Rightarrow \star.\ \mathsf{evi}\ (f(\mu_\star f)) \to \mathsf{evi}\ (\mu_\star f)) \times \\
&\quad (\forall \alpha : \star.\ \mathsf{evi}\ \alpha \to \alpha \to \mathsf{auth}\ \alpha) \times \\
&\quad (\forall \alpha : \star.\ \mathsf{evi}\ \alpha \to \mathsf{auth}\ \alpha \to \mathsf{m}\ \alpha)
\end{aligned}
$$

(b) Module signature as a type in $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$.

Figure 1: Authentikit type signatures.

`return` and `bind`. As usual with monadic programming, the `bind` operation allows multiple steps of authenticated computation to be sequenced together. As we will see, forcing programs to be structured with these primitives allows the library to perform operations for handling and checking proofs between steps of execution.

Third, as the implementation will use hash functions for authenticating data, the interface requires a way to show that the data we want to authenticate can be hashed. In particular, there must be a way to serialize the value to a string. For example, OCaml has first class functions as values, but there is no way to serialize these functions and deserialize them reliably across processes, so we cannot use hashes to authenticate a function. Authentikit requires the programmer to build explicit serialization functions using the combinators in the `Authenticatable` submodule. A term of type `t evi` is a serialization function for data of type `t`.

Finally, there are two functions `auth` and `unauth`. The `auth` function produces authenticated values, and, given an authenticated value, the `unauth` function returns an authenticated computation that "unwraps" the authenticated value. The two functions correspond directly to two built-in operations of the same names found in $\lambda\bullet$. As we will see, the behavior of these two functions differs significantly between the verifier and prover implementations of the Authentikit interface.

Merkle Trees. Before we discuss how the Authentikit interface can be implemented, consider an implementation of basic Merke trees using Authentikit shown in Figure 2.

The `Merkle` module is parameterized by an implementation of the Authentikit interface. It defines a type for paths and a type for (authenticated) trees as well as operations for constructing, querying, and updating trees. Notice how the type signatures matches the interface one would expect for ordinary binary trees, with the addition of authentication annotations using the `auth` and `m` constructors. For example, rather than returning a tree, `make_leaf` returns an authenticated tree; and rather than taking a path and a tree as input and returning the result of a query, `retrieve` takes a path and an authenticated tree as input and returns the result of the query in the authenticated computation monad.

```
module Merkle = functor (K : AUTHENTIKIT) -> struct
  open K

  type path = [`L | `R] list
  type tree = [`left of string |
               `right of tree auth * tree auth]

  let tree_evi : tree Authenticatable.evi =
    Authenticatable.(sum string (pair auth auth))

  let make_leaf (s : string) : tree auth =
    auth tree_evi (`left s)
  let make_branch (l r : tree auth) : tree auth =
    auth tree_evi (`right (l,r))

  let rec retrieve (p : path) (t : tree auth)
    : string option m =
    bind (unauth tree_evi t) (fun t ->
      match p, t with
      | [], `left s -> return (Some s)
      | `L::p, `right (l,_) -> retrieve p l
      | `R::p, `right (_,r) -> retrieve p r
      | _, _ -> return None)

  let rec update (* ... *) = (* ... *)
end
```

Figure 2: Merkle trees implemented using Authentikit.

The function `tree_evi` defines a serializer for trees using the serialization primitives from the `Authenticatable` submodule. The `make_leaf` and `make_branch` functions are implemented using `auth` and the corresponding data constructors. Finally, the `retrieve` operation is just an ordinary binary tree traversal but with a few annotations: trees are first unwrapped using `unauth` and the computation is tracked using `bind` and `return`. The `update` function is defined in a similar way.

Because the `Merkle` module is parameterized by `Authentikit`, to run a computation, we first have to instantiate the module with an implementation of `Authentikit`. Figure 3 shows three such implementations that will give rise to the prover, verifier, and "ideal" semantics of the data structure.

Prover Implementation. The `Prover` module shown in Figure 3a implements the Authentikit module signature. The prover views authenticated values as a pair of an underlying value and a hash of its string representation. Authenticated computations are thunks that return a pair of a proof and a result.[2] Proofs are represented as lists of strings. The `bind` operation appends together the proofs that are generated by each step of the computation.

Authenticatable values are, from the perspective of the prover, values for which there exist a serialization function. The combinators are implemented following a consistent serialization scheme; we omit the details but discuss the requirements in §4. In the case of `auth`, the serialization of an authenticated value is its hash code, the "digest", not the underlying value. This means that, for example, the string representation of an authenticated tree is the root hash. Finally, the `auth` function pairs an input with its hashed string representation. The `unauth` function returns the underlying value and produces a proof containing the serialization of the value.

When we instantiate the `Merkle` module with the `Prover` module, each recursive call in `retrieve` will generate a singleton proof from the `unauth` invocation used to access the current node. These proofs are then appended together through the `bind` operations to produce an overall proof for the complete call to `retrieve`. Figure 4 depicts a Merkle tree from the perspective of the prover. Calling `retrieve` on the authenticated tree (t_0, h_0) and the path $[\mathsf{R}, \mathsf{L}]$ produces the proof $[(h_1, h_2), (h_5, h_6), s_5]$. There is some redundancy in the proof that is generated during a call to `retrieve`, as compared to a typical, manual implementation of a Merkle tree; Miller et al. [30] noted a similar redundancy in the code generated for `retrieve` in $\lambda\bullet$. We will revisit this in §6, where we discuss general optimizations that may reduce the proof size, and in §7, where we verify a manual implementation of `retrieve` that generates the optimal proof.

Verifier Implementation. The module `Verifier` in Figure 3c also implements the Authentikit signature. The verifier views authenticated values as strings, *i.e.*, just the hash code of the corresponding value. Authenticated computations, meanwhile, are functions that take proofs as input and either (1) return a value and a left-over proof, or (2) indicate failure for an invalid proof.

Authenticatable values are, from the perspective of the verifier, values for which there exists both a serialization function and a deserialization function. The combinators are implemented following the same serialization scheme as the prover so that serialization followed by deserialization for authenticated values is the identity.

To create authenticated values, the `auth` function serializes and hashes the value. For instance, the Merkle tree shown in Figure 4 is from the verifier's perspective just the root hash h_0. The `unauth` function, however, is the primary workhorse where the proof checking happens. It receives a hash code and proof as input and checks that the hash code of the first item in the proof list matches. If so, the proof item is deserialized and the result is returned together with the remaining proof. If the proof list is empty, if the hash code does not match, or if the deserialization fails, then the verifier returns `` `ProofFailure ``.

Instantiating the `Merkle` module with the `Verifier` module yields an implementation of `retrieve` that, in symmetry with the prover, consumes and checks a proof item against a hash code for every node it encounters as it descends.

Ideal Implementation. The module `Ideal` shown in Figure 3b implements the Authentikit module signature with the type constructors and operations as the identity. The module is not intended to be executed but serves as a specification device to say how the data structure functionally should behave. In particular, instantiating `Merkle` with the `Ideal` module yields an implementation of ordinary (non-authenticated) binary trees and, *e.g.*, the ideal perspective of the Merkle tree shown in Figure 4 is the corresponding binary tree where all the hash codes are erased.

Security and Correctness. Now that we have seen the implementation of the Authentikit library, let us consider at a high level why this library has the security and correctness properties described in §1. Consider the security property and look at what happens when `Merkle` is instantiated with the `Verifier` module and the `Ideal` module. We can compare how the `Verifier` version of `retrieve` runs when its input tree argument is a hash corresponding to the root of a tree given as input to the `Ideal` version of `retrieve`. If the `unauth` function in the `Verifier` code does not return `` `ProofFailure ``, then the value it returns must have the same hash as the value the `Ideal` version is using. Thus, either these values are equal, or we have exhibited a hash collision. Since the implementation of `retrieve` is the same outside of the code from the Authentikit module, this means that, absent a hash collision or a proof failure, the overall result of the retrieve operations matches. Similarly, for the correctness property, when the verifier is given as input the proof produced by the prover, each call to `unauth` in `retrieve` on the verifier side should return the same node value that the prover used in the corresponding step.

In the end, it is not so challenging to prove that one *particular* client of the Authentikit library, like `Merkle`, has the intended security and correctness properties. However, our goal is to prove that *any* well-typed client of the library has these properties. This is much more challenging to show because it requires somehow exploiting the fact that any well-typed client must use the underlying Authentikit operations in a "generic way", as enforced by the type system, so that we can relate what happens when the different implementations of these operations are plugged into a client.

2.2 The $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$ Language

As a first step toward proving that authenticated data structures implemented using Authentikit satisfy the intended security and correctness properties, we need a formal model of how the module type system works in a language like OCaml. Although there have been many proposals for models of Standard ML and OCaml-like module systems (*e.g.*, [9]), in this work we use an approach based on translating modules into terms in a variant of System F_ω [17]. It

[2] We deviate slightly from Atkey's original presentation by implementating authenticated computations as thunks. This is necessary for the correctness theorems to hold in the presence of side effects.

```
type proof = string list

module Prover : AUTHENTIKIT =
  type 'a auth = 'a * string
  type 'a m = () -> proof * 'a
  let return a () = ([], a)
  let bind c f =
    let (prf,  a) = c () in
    let (prf', b) = f a () in
    (prf @ prf', b)

  module Authenticatable = struct
    type 'a evidence = 'a -> string
    let auth (_, h) = h
    (* ... *)
  end

  let auth evi a = (a, hash (evi a))
  let unauth evi (a, _) () = ([evi a], a)
  let run m = m ()
end
```

(a) Prover. *Authentikit*$_P$ **denotes the corresponding term in** $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$.

```
module Ideal : AUTHENTIKIT = struct
  type 'a auth = 'a
  type 'a m = () -> 'a

  let return a () = a
  let bind a f () = f (a ()) ()
  (* .. *)
  let auth _ a = a
  let unauth _ a () = a
end
```

(b) Ideal. *Authentikit*$_I$ **denotes the corresponding term in** $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$.

```
type proof = string list

module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a m =
    proof -> [`Ok of proof * 'a | `ProofFailure ]

  let return a prf = `Ok (prf, a)

  let bind c f prf =
    match c prf with
    | `ProofFailure -> `ProofFailure
    | `Ok (prf', a) -> f a prf'

  module Authenticatable = struct
    type 'a evidence =
      { serialize : 'a -> string;
        deserialize : string -> 'a option }
    (* ... *)
  end

  let auth evi a = hash (evi.serialize a)

  let unauth evi h prf =
    match prf with
    | p :: ps when hash p = h ->
      (match evi.deserialize p with
       | None   -> 'ProofFailure
       | Some a -> `Ok (ps, a))
    | _ -> 'ProofFailure

  let run cf prf =
    match c prf with
    | `Ok (_, a) -> a
    | _ -> failwith "Proof failure"
end
```

(c) Verifier. *Authentikit*$_V$ **denotes the corresponding term in** $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$.

Figure 3: Three realizations of the Authentikit interface in OCaml..

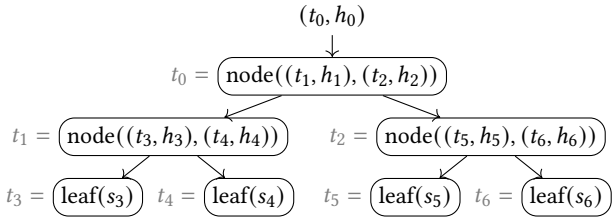

Figure 4: The prover view of a Merkle tree where h_i is the hash of t_i. The hash of a node is uniquely determined by the hashes of its children, *e.g.*, h_2 is derived from h_5 and h_6.

is part of the trusted computing base that this translation faithfully captures the semantics of compiled OCaml programs. Specifically, we use $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$, a higher-order programming language with higher-order references and a type system with polymorphism, abstract data types, iso-recursive types, and type abstraction in the style of System F_ω. The syntax and typing rules of $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$ is given in Figure 5.

The term language is mostly standard, with the addition of a primitive hash operation. Note that there are no types in terms: we write Λe for type abstraction, $e\ \langle\rangle$ for type application, and pack v and unpack e_1 as x in e_2 for formation and elimination of abstract data types. The operations fold e and unfold e are the special term constructs for iso-recursive types. ref e allocates a new reference, $!\,e$ dereferences the location e evaluates to, and $e_1 \leftarrow e_2$ assigns the result of evaluating e_2 to the location that e_1 evaluates to. We write $\lambda x.\, e$ to mean rec $\_\ x = e$, let $x = e_1$ in e_2 to mean $(\lambda x.\, e_2)\ e_1$, and $e_1; e_2$ to mean let $\_ = e_1$ in e_2.

Types have a standard kind structure: the kind $\star$ for the kind of proper types and $\kappa_1 \Rightarrow \kappa_2$ for constructors that given a type of kind κ_1 produce a type of kind κ_2. Types are formed from type variables α, type abstractions $\lambda\alpha : \kappa.\, \tau$, type applications $\tau_1\ \tau_2$, and constructors c. Constructors include unit, Booleans, integers, strings, products, disjoint sums, arrows, and references as well as universal, existential, and recursive quantifiers. We write binary constructors using infix notation, *e.g.*, $\tau \to \sigma$ to mean $\to\ \tau\ \sigma$, and we write $\forall\alpha : \kappa.\, \tau$ to mean $\forall_\kappa\ (\lambda\alpha : \kappa.\, \tau)$, and similarly for existential and recursive types. The typing judgment $\Theta \mid \Gamma \vdash e : \tau$ is mostly standard and assigns a type of kind $\star$ to a term in contexts Θ and Γ. The context Θ assigns kinds to type variables and Γ assigns types of kind $\star$ to term-level variables.

Figure 1b shows the Authentikit OCaml module signature from Figure 1a translated into the type AUTHENTIKIT in $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$. The type is derived by applying the translation described by Rossberg et al. [37]. Note that when defining the AUTHENTIKIT type, we introduce an intermediate definition Authentikit which will be useful for stating our security and correctness theorems.

Syntax

(values) $v ::= () \mid b \in \mathbb{B} \mid z \in \mathbb{Z} \mid s \in \mathit{String} \mid \ell \in \mathit{Loc} \mid (v, v) \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \mathsf{rec}\, f\, x = e \mid \Lambda e \mid \mathsf{pack}\, v$

(terms) $e ::= v \mid x \in \mathit{Var} \mid \circledcirc_1\, e \mid e \circledcirc_2 e \mid e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \mid \mathsf{case}\ e\, e\, e \mid \mathsf{ref}\, e \mid !\, e \mid e \leftarrow e \mid \mathsf{fold}\, e \mid \mathsf{unfold}\, e \mid e\, \langle\rangle \mid \mathsf{pack}\, e \mid \mathsf{unpack}\, e\, \mathsf{as}\, x\, \mathsf{in}\, e \mid \mathsf{hash}\, e$

(unary operators) $\circledcirc_1 ::= - \mid ! \mid \mathsf{intOfString} \mid \mathsf{stringOfInt} \mid \mathsf{length}$

(binary operators) $\circledcirc_2 ::= + \mid - \mid \cdot \mid = \mid \ldots$

(evaluation contexts) $K ::= - \mid \circledcirc_1\, K \mid e \circledcirc_2 K \mid K \circledcirc_2 v \mid e\, K \mid K\, v \mid \ldots$

(type elim. contexts) $T ::= - \mid T\, \tau$

(types) $\tau ::= \alpha \in \mathit{TyVar} \mid \lambda\alpha : \kappa.\, \tau \mid \tau\, \tau \mid c$

(kinds) $\kappa ::= \star \mid \kappa \Rightarrow \kappa$

(constructors) $c ::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{string} \mid \times \mid + \mid \to \mid \mathsf{ref} \mid \forall_\kappa \mid \exists_\kappa \mid \mu_\kappa$

Type Formation

$$\frac{}{\mathsf{unit}, \mathsf{bool}, \mathsf{int}, \mathsf{string} : \star} \qquad \frac{}{\times, +, \to : \star \Rightarrow \star \Rightarrow \star} \qquad \frac{}{\mathsf{ref} : \star \Rightarrow \star} \qquad \frac{}{\forall_\kappa, \exists_\kappa : (\kappa \Rightarrow \star) \Rightarrow \star} \qquad \frac{}{\mu_\kappa : (\kappa \Rightarrow \kappa) \Rightarrow \kappa}$$

$$\frac{\alpha : \kappa \in \Theta}{\Theta \vdash \alpha : \kappa} \qquad \frac{c : \kappa}{\Theta \vdash c : \kappa} \qquad \frac{\Theta, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Theta \vdash \lambda\alpha : \kappa_1.\tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Theta \vdash \sigma : \kappa_1 \Rightarrow \kappa_2 \qquad \Theta \vdash \tau : \kappa_1}{\Theta \vdash \sigma\, \tau : \kappa_2}$$

Type Elimination Context Formation

$$\frac{}{\Theta \vdash - : \kappa \hookrightarrow \kappa} \qquad \frac{\Theta \vdash T : \kappa_1 \hookrightarrow (\kappa \Rightarrow \kappa_2) \qquad \Theta \vdash \tau : \kappa}{\Theta \vdash T\, \tau : \kappa_1 \hookrightarrow \kappa_2}$$

Type Equivalence

$$\frac{\Theta \vdash \tau : \kappa}{\Theta \vdash \tau \equiv \tau : \kappa} \qquad \frac{\Theta \vdash \tau' \equiv \tau : \kappa}{\Theta \vdash \tau \equiv \tau' : \kappa} \qquad \frac{\Theta \vdash \tau \equiv \tau' : \kappa \qquad \Theta \vdash \tau' \equiv \tau'' : \kappa}{\Theta \vdash \tau \equiv \tau'' : \kappa} \qquad \frac{\Theta, \alpha : \kappa_1 \vdash \tau \equiv \tau' : \kappa_2}{\Theta \vdash \lambda\alpha : \kappa_1.\, \tau \equiv \lambda\alpha : \kappa_1.\, \tau' : \kappa_1 \Rightarrow \kappa_2}$$

$$\frac{\Theta \vdash \tau \equiv \sigma : \kappa_1 \Rightarrow \kappa_2 \qquad \Theta \vdash \tau' \equiv \sigma' : \kappa_1}{\Theta \vdash \tau\, \tau' \equiv \sigma\, \sigma' : \kappa_2} \qquad \frac{\Theta, \alpha : \kappa_1 \vdash \tau : \kappa_2 \qquad \Theta \vdash \sigma : \kappa_1}{\Theta \vdash (\lambda\alpha : \kappa_1.\, \tau)\sigma \equiv \tau[\sigma/\alpha] : \kappa_2} \qquad \frac{\Theta \vdash \tau : \kappa_1 \Rightarrow \kappa_2 \qquad \alpha \notin \Theta}{\Theta \vdash \tau \equiv \lambda\alpha : \kappa_1.\, \tau\, \alpha : \kappa_1 \Rightarrow \kappa_2}$$

Term Formation (excerpt)

$$\frac{\Gamma(x) = \tau}{\Theta \mid \Gamma \vdash x : \tau} \qquad \frac{\Theta \mid \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2 \qquad \Theta \vdash \tau_1, \tau_2 : \star}{\Theta \mid \Gamma \vdash \mathsf{rec}\, f\, x = e : \tau_1 \to \tau_2} \qquad \frac{\Theta \mid \Gamma \vdash e_2 : \tau_1 \to \tau_2 \qquad \Theta \mid \Gamma \vdash e_1 : \tau_1}{\Theta \mid \Gamma \vdash e_2\, e_1 : \tau_2}$$

$$\frac{\Theta, \alpha : \kappa \mid \Gamma \vdash e : \tau}{\Theta \mid \Gamma \vdash \Lambda e : \forall\alpha : \kappa.\, \tau} \qquad \frac{\Theta \mid \Gamma \vdash e : \forall\alpha : \kappa.\, \tau \qquad \Theta, \alpha : \kappa \vdash \tau : \star \qquad \Theta \vdash \sigma : \kappa}{\Theta \mid \Gamma \vdash e\, \langle\rangle : \tau[\sigma/\alpha]} \qquad \frac{\Theta \mid \Gamma \vdash e : \tau[\sigma/\alpha] \qquad \Theta, \alpha : \kappa \vdash \tau : \star \qquad \Theta \vdash \sigma : \kappa}{\Theta \mid \Gamma \vdash \mathsf{pack}\, e : \exists\alpha : \kappa.\, \tau}$$

$$\frac{\Theta \mid \Gamma \vdash e_1 : \exists\alpha : \kappa.\, \tau_1 \qquad \Theta, \alpha : \kappa \mid \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \Theta, \alpha : \kappa \vdash \tau_1 : \star \qquad \Theta \vdash \tau_2 : \star \qquad \alpha \notin \Theta, \tau_2}{\Theta \mid \Gamma \vdash \mathsf{unpack}\, e_1\, \mathsf{as}\, x\, \mathsf{in}\, e_2 : \tau_2}$$

$$\frac{\Theta \mid \Gamma \vdash e : T[\tau[\mu\alpha : \kappa.\, \tau/\alpha]] \qquad \Theta \vdash T : \kappa \hookrightarrow \star \qquad \Theta, \alpha : \kappa \vdash \tau : \kappa}{\Theta \mid \Gamma \vdash \mathsf{fold}\, e : T[\mu\alpha : \kappa.\, \tau]} \qquad \frac{\Theta \mid \Gamma \vdash e : T[\mu\alpha : \kappa.\, \tau] \qquad \Theta \vdash T : \kappa \hookrightarrow \star \qquad \Theta, \alpha : \kappa \vdash \tau : \kappa}{\Theta \mid \Gamma \vdash \mathsf{unfold}\, e : T[\tau[\mu\alpha : \kappa.\, \tau/\alpha]]}$$

$$\frac{\Theta \mid \Gamma : e : \tau}{\Theta \mid \Gamma \vdash \mathsf{ref}\, e : \mathsf{ref}\, \tau} \qquad \frac{\Theta \mid \Gamma : e_1 : \mathsf{ref}\, \tau \qquad \Theta \mid \Gamma : e_2 : \tau}{\Theta \mid \Gamma \vdash e_1 \leftarrow e_2 : \mathsf{unit}} \qquad \frac{\Theta \mid \Gamma : e : \mathsf{ref}\, \tau}{\Theta \mid \Gamma \vdash\, ! e : \tau} \qquad \frac{\Theta \mid \Gamma \vdash e : \mathsf{string}}{\Theta \mid \Gamma \vdash \mathsf{hash}\, e : \mathsf{string}} \qquad \frac{\Theta \vdash \tau \equiv \sigma : \star \qquad \Theta \mid \Gamma \vdash e : \sigma}{\Theta \mid \Gamma \vdash e : \tau}$$

Figure 5: Syntax and type system of $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$.

While OCaml uses iso-recursion for nominal types such as variants and records, the polymorphic variants used in Authentikit are equi-recursive. To avoid modeling both iso-recursive and equi-recursive types (which adds considerable complexity [8]), we use iso-recursive types throughout and instead add an explicit operation (shown in gray) for creating evidence of recursive types.

Throughout the rest of the paper, when presenting extended code snippets, we will continue to use OCaml syntax for readability as in Figure 1a, however, in each case, all of our formal proofs are stated in terms of translations of these programs into $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$.

3 Collision-Free Reasoning in Separation Logic

As outlined in §1, our first step in constructing a model of $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$'s type system is a program logic, *Collision-Free Separation Logic* (CF-SL), that is expressive enough to state and prove our desired security and correctness properties. CF-SL is built as an extension to the Iris program logic [25], which is a modern variant of separation logic [35]. The key extension we add is a rule that internalizes the collision-resistance property of the cryptographic hash functions we use, allowing us to only consider execution traces that are *collision-free* when proving a specification.

$$e_1 \overset{\text{pure}}{\rightsquigarrow} e_2 * \mathsf{wp}\ e_2\ \{\Phi\} \vdash \mathsf{wp}\ e_1\ \{\Phi\} \quad \textsc{wp-pure}$$

$$\mathsf{True} \vdash \mathsf{wp}\ \mathsf{ref}\ v\ \{\ell.\ \ell \mapsto v\} \quad \textsc{wp-alloc}$$

$$\ell \mapsto v \vdash \mathsf{wp}\ !\ell\ \{v.\ \ell \mapsto v\} \quad \textsc{wp-load}$$

$$\ell \mapsto v \vdash \mathsf{wp}\ \ell \leftarrow w\ \{\_.\ \ell \mapsto w\} \quad \textsc{wp-store}$$

$$\Phi(v) \vdash \mathsf{wp}\ v\ \{\Phi\} \quad \textsc{wp-val}$$

$$\mathsf{wp}\ e\ \{v.\ \mathsf{wp}\ K[v]\ \{\Phi\}\} \vdash \mathsf{wp}\ K[e]\ \{\Phi\} \quad \textsc{wp-bind}$$

$$(\forall v.\ \Psi(v) \mathrel{-\!\!*} \Phi(v)) * \mathsf{wp}\ e\ \{\Psi\} \vdash \mathsf{wp}\ e\ \{\Phi\} \quad \textsc{wp-wand}$$

Figure 6: Standard weakest precondition rules.

CF-SL's main program specification construct is a *weakest precondition* assertion of the form $\mathsf{wp}\ e\ \{\Phi\}$. In most separation logics with weakest preconditions, $\mathsf{wp}\ e\ \{\Phi\}$ holds in a state σ if, when executing e in σ, execution of e is *safe* (meaning that the program never gets stuck or triggers an assertion failure), and upon termination, the resulting state will satisfy the postcondition Φ. In CF-SL, we weaken the meaning of $\mathsf{wp}\ e\ \{\Phi\}$ to only require safety and postcondition-satisfaction for executions in which e does not compute a hash collision.

To state this precisely, we augment $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$'s semantics to track a history of all hashes computed during execution. Assume the existence of a hash function $\mathcal{H}$. We consider an operational semantics with program states consisting of a heap (modeled as a finite map from locations to values) and a *history* of the strings that have been hashed during execution.

$$\sigma \in \mathit{State} \triangleq (\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}) \times \mathit{Set}(\mathit{String})$$

The operational semantics $(\cdot \to \cdot) \in (\mathit{Expr} \times \mathit{State}) \times (\mathit{Expr} \times \mathit{State})$ extends the history when a program performs a hash operation but is otherwise standard.

$$\langle \mathsf{hash}\ s, (m, h) \rangle \to \langle \mathcal{H}(s), (m, h \cup \{s\}) \rangle$$

We define a *collision-free step*

$$\langle e, \sigma \rangle \to_{\mathrm{cf}} \langle e', \sigma' \rangle \triangleq \langle e, \sigma \rangle \to \langle e', \sigma' \rangle \wedge \mathit{collisionFree}(\sigma')$$

where

$$\mathit{collisionFree}(m, h) \triangleq \nexists s_1, s_2 \in h.\ \mathit{collision}(s_1, s_2)$$
$$\mathit{collision}(s_1, s_2) \triangleq s_1 \neq s_2 \wedge \mathcal{H}(s_1) = \mathcal{H}(s_2).$$

We use $\to^*_{\mathrm{cf}}$ to denote the reflexive transitive closure of $\to_{\mathrm{cf}}$ and take the predicate $\mathit{safe}_{cf}(e)$ to mean that all collision-free executions of e are safe. The soundness theorem of CF-SL shown below only considers collision-free execution traces.

Theorem 3.1 (Soundness). *Let φ be a first-order predicate. If*

$$\mathsf{True} \vdash \mathsf{wp}\ e\ \{\varphi\}$$

is derivable then $\mathit{safe}_{cf}(e)$ and for all collision-free states σ, if

$$\langle e, \sigma \rangle \to^*_{\mathrm{cf}} \langle v, \sigma' \rangle$$

then $\varphi(v)$ holds at the meta level.

CF-SL satisfies all the standard separation logic rules (see Figure 6 for an excerpt) from the Iris program logic. The rules use the separating conjunction connective, $P * Q$, which holds in some program state σ if it is possible to decompose σ into two disjoint pieces, σ_1 and σ_2, which satisfy P and Q respectively. The separating implication $P \mathrel{-\!\!*} Q$ is a form of implication that is the right adjoint of the separation conjunction, in the sense that $P * (P \mathrel{-\!\!*} Q) \vdash Q$. The points-to assertion $\ell \mapsto v$ holds in a state σ if location ℓ in σ stores the value v. In contrast to standard conjunction, $P \not\vdash P * P$ in general, since it may not be possible to split a state into two sub-pieces that each satisfy P. Because separation logic assertions delineate a part of program state, assertions are often called *resources*.

To reason in a collision-free compositional manner, CF-SL introduces a new resource $\mathsf{hashed}(s)$ that captures that s can be found in the hash history and thus has been hashed using $\mathcal{H}$ at some point during execution. The rule WP-HASH shown below reflects the operational behavior of the hash operation. In the postcondition the $\mathsf{hashed}(s)$ resource is obtained.

$$\textsc{wp-hash}\quad \mathsf{True} \vdash \mathsf{wp}\ \mathsf{hash}\ s\ \{v.\ v = \mathcal{H}(s) * \mathsf{hashed}(s)\}$$

The $\mathsf{hashed}(s)$ resource is duplicable, *i.e.*,

$$\mathsf{hashed}(s) \dashv\vdash \mathsf{hashed}(s) * \mathsf{hashed}(s)$$

and satisfies the rule HASH-VALIDITY which embodies reasoning without collisions: if two hashed strings witness a hash collision then the goal can trivially be discharged.

$$\textsc{hash-validity}\quad \frac{\mathit{collision}(s_1, s_2)}{\mathsf{hashed}(s_1) * \mathsf{hashed}(s_2) \vdash \mathsf{False}}$$

Relational Collision-Free Reasoning. So far, what we have seen is a *unary* logic that allows us to prove properties of a single program. However, our goal is to relate the behaviors of a prover, verifier, and ideal version of a program. To do so, we need a *relational* logic that will allow us to prove a correspondence between the behaviours of multiple programs. We follow CaReSL [49] and construct a relational variant of CF-SL by encoding a second program as a resource $\mathsf{spec}(e)$. The resource tracks a "specification" program which can be updated and progressed according to the operational semantics. For example, the resource $\mathsf{spec}((\lambda x.\ e_1)\ v_2)$ can be updated to $\mathsf{spec}(e_1[v_2/x])$, reflecting execution of a beta reduction. Formally, this is specified as a *view shift* implication [25] $\mathsf{spec}((\lambda x.\ e_1)\ v_2) \Rrightarrow \mathsf{spec}(e_1[x/v_2])$ in Iris. A view shift $P \Rrightarrow Q$ intuitively says that, given resources satisfying P, we can update our resources and the "logical state" to obtain resources satisfying Q. In particular,

$$\frac{Q \vdash \mathsf{wp}\ e\ \{\Phi\}}{(P \Rrightarrow Q) * P \vdash \mathsf{wp}\ e\ \{\Phi\}}$$

The relational logic also comes with a points-to connective $\ell \mapsto_{\mathsf{s}} v$ that denotes ownership of the location ℓ and its contents v on the heap of the specification program. For example, storing a value to a location in the specification program requires ownership of the points-to connective, as captured by the following rule:

$$\mathsf{spec}(\ell \leftarrow w) * \ell \mapsto_{\mathsf{s}} v \Rrightarrow \mathsf{spec}(()) * \ell \mapsto_{\mathsf{s}} w$$

in which, on the right hand side, we have $\mathsf{spec}(())$ (reflecting that the store returned the unit value $()$), and the points-to assertion is updated to reflect the updated value of the location. We refer to Frumin et al. [13] and our Rocq formalization [19] for a detailed discussion on how the specification resources are defined.

A relational variant of the soundness theorem of CF-SL follows as a consequence from the resource construction and Theorem 3.1.

Corollary 3.2 (Relational Soundness). *Let φ be a first-order relation. If*

$$\mathsf{spec}(e_2) \vdash \mathsf{wp}\ e_1\ \{v_1.\ \exists v_2.\ \mathsf{spec}(v_2) * \varphi(v_1, v_2)\}$$

is derivable then $\mathit{safe}_{cf}(e_1)$ and for all collision-free states σ, if

$$\langle e_1, \sigma\rangle \to^*_{\mathrm{cf}} \langle v_1, \sigma_1\rangle$$

then there exists a value v_2 and state σ_2 such that $\langle e_2, \sigma\rangle \to^* \langle v_2, \sigma_2\rangle$ and $\varphi(v_1, v_2)$ holds at the meta level.

4 Security

In this section, we show that ADSs implemented using Authentikit have the security property described in §1. Formally stated, the security theorem looks as follows. We say a type τ is a *primitive type* if it is either unit, bool, int, string, $\tau_1 \times \tau_2$, or $\tau_1 + \tau_2$ where τ_1 and τ_2 are primitive types.

Theorem 4.1 (Security). *Let τ be a primitive type. If*

$$\emptyset \mid \emptyset \vdash e : \forall \mathit{auth}, m : \star \Rightarrow \star.\ \mathit{Authentikit}\ \mathit{auth}\ m \to m\ \tau$$

then for all proofs p (a list of strings) and collision-free states σ, if

$$\langle \mathrm{run}_V\ (e\ \langle\rangle\ \langle\rangle\ \mathit{Authentikit}_V)\ p, \sigma\rangle \to^*_{\mathrm{cf}} \langle \mathsf{Some}\, v, \sigma_1\rangle$$

then there exists a state σ_2 such that

$$\langle \mathrm{run}_I\ (e\ \langle\rangle\ \langle\rangle\ \mathit{Authentikit}_I), \sigma\rangle \to^* \langle v, \sigma_2\rangle$$

In prose, the theorem requires that e is a well-typed function that takes an Authentikit implementation as an argument and returns an authenticated computation. Then it says that if we instantiate e with the verifier implementation $\mathit{Authentikit}_V$ and it accepts the proof p, it will return the same value as the ideal semantics given by instantiating and running e with $\mathit{Authentikit}_I$.

The challenge in proving this theorem is that all we know about e is that it has the type stated in the premise. Intuitively, since e has this type, it must use the operations provided by the Authentikit interface in a generic way and cannot violate the abstractions of the interface. To prove the theorem formally, we must reason about these abstraction guarantees enforced by the type system. A standard approach for reasoning about type abstraction is to construct a model using logical relations. In this section, we construct such a model using CF-SL and use it to prove the theorem.

4.1 Logical Relation for Security

A logical-relations model provides an interpretation for a type system by describing the behaviors that all programs with a given type τ should have. Constructing such a model "directly" for a highly expressive type system like that of $\mathsf{F}^{\mathrm{ref}}_{\omega,\mu}$ is challenging, but in recent years, the so-called "logical" approach to logical relations [11] has made this easier by defining the logical relation in terms of a highly expressive program logic. This approach has been used with the Iris program logic for a wide range of languages with System F-like type systems, *e.g.*, to prove program refinement [13].

To define a model of $\mathsf{F}^{\mathrm{ref}}_{\omega,\mu}$, we take a similar approach using the relational collision-free logic from §3, and adapt techniques from recent work of Sieczkowski et al. [40] to incorporate the higher kinds found in $\mathsf{F}^{\mathrm{ref}}_{\omega,\mu}$. For completeness, the full model is defined in Figure 7, though many of the technical details of the construction are not needed to understand the rest of our results.

First, the model defines an interpretation of types, $[\![\Theta \vdash \tau : \kappa]\!]_\Delta$ where $\Delta \in [\![\Theta]\!]$ interprets the free type variables in τ. Due to the higher-kinded nature of $\mathsf{F}^{\mathrm{ref}}_{\omega,\mu}$, the co-domain of this interpretation depends on the kind κ of τ. The kind $\star$ of proper types is interpreted as binary relations in the logic. Intuitively, $[\![\Theta \vdash \tau : \star]\!]_\Delta$ characterizes the set of pairs of closed values (v_1, v_2) of type τ such that v_1 *refines* v_2.[3] The kind $\kappa_1 \Rightarrow \kappa_2$ of constructors is interpreted as maps[4] from $[\![\kappa_1]\!]$ to $[\![\kappa_2]\!]$.

The interpretation of types is defined by structural recursion on the type: type variables are interpreted by lookup in Δ, type abstractions as maps, and type application as application of maps. Constructors are interpreted using corresponding connectives in the logic in a standard way: *e.g.*, functions are interpreted using (separating) implication taking related inputs to related outputs, universal types are interpreted using universal quantification, reference types are interpreted using the points-to connectives, and recursive types are interpreted using a guarded fixed point [32].

Next, we define a term interpretation, $\mathcal{E}(R)(e_1, e_2)$, as a relational assertion in CF-SL. This interpretation assertion is a separating implication that takes the specification program executing e_2 as assumptions, and has as a conclusion a weakest precondition asserting that if e_1 reduces to some value v_1, then there exists a corresponding execution of e_2 to a value v_2 such that $R(v_1, v_2)$ holds. Finally, the logical relation $\Theta \mid \Gamma \vDash e_1 \preceq e_2 : \tau$ extends the interpretation to typed open terms by requiring that, after substituting in related values for each free variable, the resulting closed terms should be related according to $\mathcal{E}$.

Using the rules of CF-SL, we next prove that the typing rules of $\mathsf{F}^{\mathrm{ref}}_{\omega,\mu}$ are *compatible* with the logical relation: for every typing rule, if we have a pair of related terms for every premise, then we also have pair of related terms for the conclusion, *e.g.*, in the case of function application, we have

$$\frac{\Theta \mid \Gamma \vDash e_2 \preceq e_2' : \tau_1 \to \tau_2 \qquad \Theta \mid \Gamma \vDash e_1 \preceq e_1' : \tau_1}{\Theta \mid \Gamma \vDash e_2\ e_1 \preceq e_2'\ e_1' : \tau_2}$$

Because such a compatibility lemma holds for every typing rule, the fundamental theorem of logical relations follows by induction on the typing derivation:

Theorem 4.2. *If $\Theta \mid \Gamma \vdash e : \tau$ then $\Theta \mid \Gamma \vDash e \preceq e : \tau$.*

That is, if e is a well-typed term in $\mathsf{F}^{\mathrm{ref}}_{\omega,\mu}$, then e is related to itself according to the logical relation. In particular, if e is a closed term, then the relational weakest precondition assertion given by $\mathcal{E}$ holds for e. Thus, just from knowing the type of e we can deduce a "free theorem" automatically about e.

Taking a step back, there is nothing specific to authenticated data structures in this model. The main changes, as compared to program-logic-based logical-relations models in prior work, is that

[3] CF-SL is substructural, while the $\mathsf{F}^{\mathrm{ref}}_{\omega,\mu}$ type system is not. To account for this, we consider relations defined as functions into $\mathit{iProp}_\square$, the type of *persistent* propositions in CF-SL. We say P is persistent if $P \vdash \square P$ where $\square$ is the Iris *persistence modality*.

[4] More specifically, because the ambient logic of Iris is step-indexed, the maps also have to be *non-expansive*, meaning that they map n-equivalent arguments to n-equivalent results. However, this is not important for an intuitive understanding of the model.

Kind interpretation

$$[\![\star]\!] \triangleq Val \times Val \to iProp_{\square}$$
$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \xrightarrow{\text{ne}} [\![\kappa_2]\!]$$

Type interpretation

$$[\![\Theta \vdash \tau : \kappa]\!]_{(\cdot)} : [\![\Theta]\!] \xrightarrow{\text{ne}} [\![\kappa]\!]$$
$$[\![\Theta \vdash \alpha : \kappa]\!]_{\Delta} \triangleq \Delta(\alpha)$$
$$[\![\Theta \vdash \lambda\alpha.\, \tau : \kappa_1 \Rightarrow \kappa_2]\!]_{\Delta} \triangleq \lambda R : [\![\kappa_1]\!].\, [\![\Theta, \alpha : \kappa_1 \vdash \tau : \kappa_2]\!]_{\Delta, R}$$
$$[\![\Theta \vdash \sigma\ \tau : \kappa_2]\!]_{\Delta} \triangleq [\![\Theta \vdash \sigma : \kappa_1 \Rightarrow \kappa_2]\!]_{\Delta}\,([\![\Theta \vdash \tau : \kappa_1]\!]_{\Delta})$$
$$[\![\Theta \vdash c : \kappa]\!]_{\Delta} \triangleq [\![c : \kappa]\!]$$

Term interpretation

$$\mathcal{E}(R)(e_1, e_2) \triangleq \forall i, K.\, \mathrm{spec}(K[e_2]) \mathrel{-\!\!*} \mathrm{wp}\ e_1\ \{v_1.\, \exists v_2.\, \mathrm{spec}(K[v_2]) * R(v_1, v_2)\}$$

Context interpretations

$$[\![\Theta]\!] \triangleq \Pi_{\alpha:\kappa \in \Theta} [\![\kappa]\!]$$
$$[\![\Theta \vdash \Gamma]\!]_{\Delta}(\vec{v}, \vec{w}) \triangleq \forall (x_i, \tau_i) \in \Gamma.\, [\![\Theta \vdash \tau_i : \star]\!]_{\Delta}(v_i, w_i)$$

Logical relation

$$\Theta \mid \Gamma \vDash e_1 \preceq e_2 : \tau \triangleq \forall \Delta \in [\![\Theta]\!], \vec{v_1}, \vec{v_2}.\, [\![\Theta \vdash \Gamma]\!]_{\Delta}(\vec{v_1}, \vec{v_2}) \mathrel{-\!\!*} \mathcal{E}\big([\![\Theta \vdash \tau : \star]\!]_{\Delta}\big)(e_1[\vec{v_1}/\Gamma], e_2[\vec{v_2}/\Gamma])$$

Constructor interpretation

$$[\![\mathsf{unit} : \star]\!] \triangleq \lambda(v_1, v_2).\, v_1 = v_2 = ()$$
$$[\![\mathsf{int} : \star]\!] \triangleq \lambda(v_1, v_2).\, \exists z \in \mathbb{Z}.\, v_1 = v_2 = z$$
$$[\![\mathsf{bool} : \star]\!] \triangleq \lambda(v_1, v_2).\, \exists b \in \mathbb{B}.\, v_1 = v_2 = b$$
$$[\![\mathsf{string} : \star]\!] \triangleq \lambda(v_1, v_2).\, \exists s \in String.\, v_1 = v_2 = s$$
$$[\![\times : \star \Rightarrow \star \Rightarrow \star]\!] \triangleq \lambda R, S : [\![\star]\!].\, \lambda(v_1, v_2).\, \exists w_1, w_2, u_1, u_2.\, v_1 = (w_1, u_1) * v_2 = (w_2, u_2) * R(w_1, w_2) * S(u_1, u_2)$$
$$[\![+ : \star \Rightarrow \star \Rightarrow \star]\!] \triangleq \lambda R, S : [\![\star]\!].\, \lambda(v_1, v_2).\, \exists w_1, w_2.\ (v_1 = \mathsf{inl}\ w_1 * v_2 = \mathsf{inl}\ w_2 * R(w_1, w_2)) \vee (v_1 = \mathsf{inr}\ w_1 * v_2 = \mathsf{inr}\ w_2 * S(w_1, w_2))$$
$$[\![\to : \star \Rightarrow \star \Rightarrow \star]\!] \triangleq \lambda R, S : [\![\star]\!].\, \lambda(v_1, v_2).\, \square \forall w_1, w_2.\, R(w_1, w_2) \mathrel{-\!\!*} \mathcal{E}(S)(v_1\ w_1, v_2\ w_2)$$
$$[\![\mathsf{ref} : \star \Rightarrow \star]\!] \triangleq \lambda R : [\![\star]\!].\, \lambda(v_1, v_2).\, \exists \ell_1, \ell_2 \in Loc.\, v_1 = \ell_1 * v_2 = \ell_2 * \boxed{\exists w_1, w_2.\, \ell_1 \mapsto w_1 * \ell_2 \mapsto_s w_2 * R(w_1, w_2)}$$
$$[\![\forall_\kappa : (\kappa \Rightarrow \star) \Rightarrow \star]\!] \triangleq \lambda R : [\![\kappa \Rightarrow \star]\!].\, \lambda(v_1, v_2).\, \forall S \in [\![\kappa]\!].\, \mathcal{E}(R(S))(v_1\ \langle\rangle, v_2\ \langle\rangle)$$
$$[\![\exists_\kappa : (\kappa \Rightarrow \star) \Rightarrow \star]\!] \triangleq \lambda R : [\![\kappa \Rightarrow \star]\!].\, \lambda(v_1, v_2).\, \exists S \in [\![\kappa]\!].\, R(S)(v_1, v_2)$$
$$[\![\mu_\kappa : (\kappa \Rightarrow \kappa) \Rightarrow \kappa]\!] \triangleq \lambda R : [\![\kappa \Rightarrow \kappa]\!].\, gfix(F_\kappa(R)) \qquad \text{where } F_\kappa(R)(f) \triangleq \begin{cases} \lambda(v_1, v_2).\, \triangleright R(f)(v_1, v_2) & \text{if } \kappa = \star \\ \lambda S \in [\![\kappa_1]\!].\, F_{\kappa_2}(id)(R(f)(S)) & \text{if } \kappa = \kappa_1 \Rightarrow \kappa_2 \end{cases}$$

Figure 7: Binary logical-relations model of $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$.

(1) it includes support for higher kinds, and (2) the term interpretation $\mathcal{E}$ uses CF-SL as the underlying program logic. The fact that there is nothing specific to authenticated data types in the model is to be expected, since the idea behind the Authentikit library is that, unlike in $\lambda\bullet$, there is no need to extend OCaml with new special types for representing authenticated data. Instead, security is ensured by the way that the library uses standard type system abstraction guarantees. Thus, the "security-specific" work in showing Theorem 4.1 lies in proving something about the implementations of Authentikit, which we turn to now.

4.2 Security of Authentikit

Let e be a client of Authentikit satisfying the premises of Theorem 4.1. Our goal in this section is to show the conclusion of this theorem by proving that when we instantiate e with $Authentikit_V$ and $Authentikit_I$, the two different instantiations of e are logically related according to the model of the previous section. This will then imply that a relational weakest precondition in CF-SL holds between these two programs, so that our result will follow by applying Corollary 3.2, the relational soundness theorem of CF-SL.

We start by applying the fundamental theorem of the logical relation to e to get that it is related to itself at the type $\forall \mathsf{auth}, \mathsf{m} : \star \Rightarrow \star.\, \mathsf{Authentikit\ auth\ m} \to \mathsf{m}\ \tau$. Unfolding the definition of the logical relation as given in Figure 7, it says that for any choice of relational interpretation of the quantified constructors auth and m, if we apply e to arguments that are related according to $[\![\mathsf{Authentikit\ auth\ m}]\!]_\Delta$, the results will be related according to $\mathcal{E}([\![\mathsf{m}\ \tau]\!]_\Delta)$ (where Δ maps auth and m to the selected interpretations). In particular, we define interpretations of auth and m so that we can show the verifier and ideal implementation of Authentikit are related, *i.e.*, we show

$$[\![\mathsf{Authentikit\ auth\ m}]\!]_\Delta(Authentikit_V, Authentikit_I)$$

For the interpretation $[\![\mathsf{auth}]\!] : [\![\star]\!] \to [\![\star]\!]$ of the constructor for authenticated values, we choose the following:

$$[\![\mathsf{auth}]\!](R)(v_1, v_2) \triangleq \exists a, t.\, v_1 = \mathcal{H}(\mathrm{serialize}_t(a)) * R(a, v_2) * \mathrm{hashed}(\mathrm{serialize}_t(a))$$

A pair of values (v_1, v_2) inhabits this relation if v_1 (the verifier side) is the hash of the serialization of some value a such that $R(a, v_2)$. where R is the relation that characterizes the type of a. The hashed resource records that the serialization of a was hashed during execution.

The (meta-level) partial function $\mathrm{serialize}_t : Val \rightharpoonup String$ maps values of type $t \in \{\mathsf{string}, \mathsf{int}, t \times t, t + t\}$ to strings according to some serialization scheme for t. Two crucial properties of the serialization strategy applied throughout the Authentikit library is injectivity and uniqueness: if $\mathrm{serialize}_{t_1}(v_1) = \mathrm{serialize}_{t_2}(v_2)$ then $v_1 = v_2$; and $\mathrm{serialize}_{t_1}(v) = \mathrm{serialize}_{t_2}(v)$ for all t_1 and t_2.

The interpretation $[\![\mathsf{m}]\!] : [\![\star]\!] \to [\![\star]\!]$ of the constructor for authenticated computations looks as follows.

$$[\![\mathsf{m}]\!](R)(v_1, v_2) \triangleq \forall K, w, p.\ \mathsf{spec}(K[v_2\ ()]) * \mathsf{isProof}(w, p) \mathrel{-\!\!*} \mathsf{wp}\ v_1\ w \left\{ \begin{pmatrix} u_1.\ u_1 = \mathsf{None} \lor \\ \left(\begin{array}{l} \exists a_1, a_2, w', p'.\ u_1 = \mathsf{Some}\ (w', a_1) * \\ \mathsf{spec}(K[a_2]) * \mathsf{isProof}(w', p') * R(a_1, a_2) \end{array}\right) \end{pmatrix} \right\}$$

The specification says that the verifier v_1, when applied to a value w corresponding to the proof stream p, returns an option value indicating whether the proof was accepted or not. If the proof is accepted, the verifier and the ideal return (w', a_1) and a_2, respectively, where $R(a_1, a_2)$ holds and w' is a value corresponding to the remaining unconsumed proof stream p'.

We continue the proof by setting $\Delta = [\mathsf{auth} \mapsto [\![\mathsf{auth}]\!], \mathsf{m} \mapsto [\![\mathsf{m}]\!]]$ and show that $\mathit{Authentikit}_V$ and $\mathit{Authentikit}_I$ are related component by component. For the *Authenticatable* sub-module we pick an interpretation of the evi constructor where $[\![\mathsf{evi}]\!](R)(v_1, v_2)$ requires v_1 to be a pair of a serializer and a deserializer that behaves as expected when applied to values inhabiting R according to the serialization scheme.

The most interesting case is unauth, where proof checking happens and all parts of the model come together, *i.e.*, we show

$$[\![\forall \alpha : \star.\ \mathsf{evi}\ \alpha \to \mathsf{auth}\ \alpha \to \mathsf{m}\ \alpha]\!]_{\Delta'}(\mathsf{unauth}_V, \mathsf{unauth}_I)$$

where $\Delta' = \Delta[\mathsf{evi} \mapsto [\![\mathsf{evi}]\!]]$. By unfolding the definition, we are to show that given $[\![\mathsf{evi}]\!](R)(v_1, v_2)$ and $[\![\mathsf{auth}]\!](R)(w_1, w_2)$ then $\mathcal{E}([\![\mathsf{m}]\!](R))(\mathsf{unauth}_V\ v_1\ w_1, \mathsf{unauth}_I\ v_2\ w_2)$. By the interpretation of m, there is some proof p which corresponds to w_1. If p is empty, the verifier returns None and we are done. If $p = s :: p'$ then we continue using WP-HASH and obtain the resource $\mathsf{hashed}(s)$. Since $[\![\mathsf{auth}]\!](R)(w_1, w_2)$ we know $w_1 = \mathcal{H}(\mathsf{serialize}_t(a))$ and we have $\mathsf{hashed}(\mathsf{serialize}_t(a))$ and $R(a, w_2)$ for some t and a. The next step of the verifier is thus the conditional test $\mathcal{H}(s) = \mathcal{H}(\mathsf{serialize}_t(a))$. If the test fails, the verifier returns None and we are done. If the test succeeds then either (1) we have encountered a collision, in which case we conclude using HASH-VALIDITY, or (2) s is equal to $\mathsf{serialize}_t(a)$ which means s is a valid serialization of a and deserialization succeeds, so we are done.

At this point, we have shown

$$\mathcal{E}([\![\mathsf{m}\ \tau]\!]_\Delta)(e\ \langle\rangle\ \langle\rangle\ \mathit{Authentikit}_V, e\ \langle\rangle\ \langle\rangle\ \mathit{Authentikit}_I)$$

which gives us a relational weakest precondition for the two instantiations of the client e. The last step of our security theorem is to prove, using this weakest precondition, that when we execute the verifier and prover with run_V and run_I, the results match the conclusion of Theorem 4.1. We prove this as a separate lemma about run_V and run_I by establishing a weakest precondition in CF-SL, and then applying Corollary 3.2, to get

Lemma 4.3 (Security, semantic). *Let φ be a first-order relation. If $\mathcal{E}([\![\mathsf{m}]\!]\ \varphi)(e_1, e_2)$ then for all proofs p and collision-free σ, if $\langle \mathsf{run}_V\ e_1\ p, \sigma\rangle \to^*_{\mathsf{cf}} \langle \mathsf{Some}\, v_1, \sigma_1\rangle$ then $\langle \mathsf{run}_I\ e_2, \sigma\rangle \to^* \langle v_2, \sigma_2\rangle$ such that $\varphi(v_1, v_2)$.*

Intuitively, this lemma says that if e_1 and e_2 are authenticated computations, then for any proof p, executing $\mathsf{run}_V\ e_1\ p$ and $\mathsf{run}_I\ e_2$ results in φ-related values. When combined with our earlier results, Theorem 4.1 follows as a corollary.

5 Correctness

Correctness of ADSs implemented against the Authentikit module type can, like security, be established using a logical relation.

Theorem 5.1 (Correctness). *Let τ be a primitive type. If*

$$\emptyset \mid \emptyset \vdash e : \forall \mathit{auth}, m : \star \Rightarrow \star.\ \mathit{Authentikit}\ \mathit{auth}\ m \to m\ \tau$$

then for all collision-free states σ, if

$$\langle \mathsf{run}_P\ (e\ \langle\rangle\ \langle\rangle\ \mathit{Authentikit}_P), \sigma\rangle \to^*_{\mathsf{cf}} \langle (p, v), \sigma_1\rangle$$

then there exists states σ_2 and σ_3 such that

$$\langle \mathsf{run}_V\ (e\ \langle\rangle\ \langle\rangle\ \mathit{Authentikit}_V)\ p, \sigma\rangle \to^* \langle \mathsf{Some}\, v, \sigma_2\rangle \text{ and}$$
$$\langle \mathsf{run}_I\ (e\ \langle\rangle\ \langle\rangle\ \mathit{Authentikit}_I), \sigma\rangle \to^* \langle v, \sigma_3\rangle$$

The theorem requires that e can be syntactically typed as taking any Authentikit implementation and returning an authenticated computation. In return, if we instantiate and run e with the prover implementation $\mathit{Authentikit}_P$, producing some proof p, then e instantiated with the verifier implementation $\mathit{Authentikit}_V$ must accept p. Moreover, both the prover and the verifier yield the same output as the ideal semantics given by instantiating and running e with $\mathit{Authentikit}_I$.

Proving this theorem raises two additional challenges beyond what was needed for the previous security theorem. The first less serious challenge is that this theorem relates together *three* programs (the prover, verifier, and ideal), instead of just two. Thus, we need to generalize the relational logic to support reasoning about three programs at once, and also generalize the logical-relations model to a ternary relation.

The second more difficult challenge is that the theorem considers executions in which the proof generated by the prover is supplied as input to the verifier. In the relational logic we have presented so far (and all other relational program logics we are aware of), there is no natural way to prove a relational specification where the output of one program should be the input to the other. At best, one can instead essentially prove two, separate *unary* specifications about the prover and verifier by first showing that all proofs generated by the prover satisfy some property P, and then showing that, if the property P is assumed as a precondition of the input to the verifier, the verifier will succeed and return the same result.

It might be possible to make this unary approach work, but doing so gives up many of the benefits that relational program logics bring. It has long been known in the relational program logic literature that relational proofs are usually simplified when the two programs can be *aligned* or *synchronized* [2, 6, 50], so that the proof reasons about similar steps in the two programs at the same time. To address this second challenge while still retaining aligned, relational reasoning, we develop a novel use of a proof technique called *prophecy variables* [1].

Before proceeding to the proof, we note that the formal statement of correctness for $\lambda\bullet$ given by Miller et al. [30] is slightly different from Theorem 5.1. In particular, their theorem says that if the ideal execution returns a value v, then there is an execution of the prover returning v and a proof p that the verifier will accept, resulting in the same value v. Besides taking the execution of the ideal as a premise instead of an execution of the prover, their version of the theorem considers normal executions of the prover,

whereas our version only considers collision-free traces. However, while the formulation of correctness given by Miller et al. holds for the $\lambda\bullet$ core calculus, it *does not* hold when applying some of the optimizations implemented in the $\lambda\bullet$ compiler. In contrast, as we will see in §6, when those corresponding optimizations are applied to an implementation of Authentikit, the formulation given in Theorem 5.1 does hold.

5.1 Logical Relation for Correctness

Our ternary logical relation $\Theta \mid \Gamma \vDash e_1 \preceq e_2 \preceq e_3 : \tau$ is defined using a ternary variant of the collision-free logic that encodes the verifier and ideal program as two separation logic resources $\mathsf{spec}_V(e)$ and $\mathsf{spec}_I(e)$. These resources can be updated and progressed according to the operational semantics just as for the $\mathsf{spec}(e)$ resource introduced in §3. Equipped with the ternary logic, we define a logical relation similar to the binary relation from §4.1 by generalization to the ternary case in the obvious way. For instance, the term interpretation looks as follows.

$$\begin{aligned}&\mathcal{E}(R)(e_1,e_2,e_3) \triangleq \forall i, K_2, K_3.\\ &\quad \mathsf{spec}_V(K_2[e_2]) * \mathsf{spec}_I(K_3[e_3]) \mathrel{-\!\!*}\\ &\quad \mathsf{wp}\ e_1 \left\{ v_1.\ \exists v_2, v_3.\ \begin{array}{l}\mathsf{spec}_V(K_2[v_2]) * \mathsf{spec}_I(K_3[v_3]) *\\ R(v_1,v_2,v_3)\end{array} \right\}\end{aligned}$$

The fundamental theorem follows in a similar way.

Theorem 5.2. *If* $\Theta \mid \Gamma \vdash e : \tau$ *then* $\Theta \mid \Gamma \vDash e \preceq e \preceq e : \tau$.

5.2 Correctness of Authentikit

The correctness proof follows the same recipe as the security proof. We start by defining a suitable interpretation of auth and m, and then show that $\mathit{Authentikit}_P$, $\mathit{Authentikit}_V$, and $\mathit{Authentikit}_I$ are related according to the ternary logical relation. The key difference is the interpretation of m, the type variable for authenticated computations. This interpretation uses prophecy variables to "predict" what the proof generated by the prover in the future will be.

A prophecy variable is a ghost variable that supplies information about what will happen later on in execution. In the Iris realization [26], a prophecy variable manifests as a separation logic resource $\mathsf{isProph}(\alpha, p)$, where α is a variable identifier and p is a sequence of values. The assertion tells us that we own the exclusive right to resolve (*i.e.*, assign values to) α, and that the predicated values that it will be resolved to are given by the sequence p.

Prophecy variables are resolved using a (ghost) programming language construct resolve α s using the rule below.

WP-PROPH-RESOLVE

$$\mathsf{isProph}(\alpha, p) \vdash \mathsf{wp}\ \ \mathsf{resolve}\ \alpha\ s \left\{ \_.\ \exists p'.\ p = s :: p' * \mathsf{isProph}(\alpha, p') \right\}$$

By resolving α to s, the rule reveals that $p = s :: p'$, *i.e.* we may reason *as if* s was equal to the head of the prophecy sequence p, even though the value s may have been the result of a computation and not statically known beforehand. The prophecy variable, however, allows us to *refer* to this future resolved value earlier on without knowing the exact value yet.

In order to prophesy the prover's output, we add a resolve command for a designated prophecy variable α in the $\mathit{Authentikit}_P$ implementation of the unauth procedure.

$$\mathsf{unauth}_P \triangleq \lambda \mathit{evi}, (a, \_), ().\ \mathsf{let}\ s = \mathit{evi}\ a\ \mathsf{in}\ \mathsf{resolve}\ \alpha\ s; ([s], a)$$

Thus, the prophecy sequence p for α will correspond to the list of values that make up the proof that is produced. We define $\mathsf{isProphProof}(\alpha, v, p) \triangleq \mathsf{isProph}(\alpha, p) * \mathsf{isProof}(v, p)$, as a representation predicate for this prophesied proof stream.

Then, the ternary interpretation of authenticated computations looks as follows.

$$\begin{aligned}&[\![\mathsf{m}]\!](R)(v_1,v_2,v_3) \triangleq \forall K_2, K_3, \alpha, p, w.\\ &\quad \mathsf{spec}_V(K_2[v_2\ w]) * \mathsf{spec}_I(K_3[v_3\ ()]) * \mathsf{isProphProof}(\alpha, w, p) \mathrel{-\!\!*}\\ &\quad \mathsf{wp}\ v_1() \left\{ u. \begin{array}{l}\exists p_1, p_2, w_1, w_2, a_1, a_2, a_3.\\ u = (w_1, a_1) * p = p_1 +\!\!+ p_2 * R(a_1,a_2,a_3) *\\ \mathsf{isProof}(w_1, p_1) * \mathsf{isProphProof}(\alpha, w_2, p_2) *\\ \mathsf{spec}_V(K_2[\,\mathsf{Some}\ (w_2, a_2)]) * \mathsf{spec}_I(K_3[a_3])\end{array} \right\}\end{aligned}$$

In the above, the input w to the verifier corresponds to the (prophesied) proof stream p. In the postcondition of v_1, it is revealed that p is the concatenation of two proof streams p_1 and p_2 where p_1 corresponds to the value w_1 produced by the prover during execution of v_1, and p_2 corresponds to the value w_2 returned by the verifier, which represents the remaining proof stream that the verifier has not yet consumed.

The definition satisfies the correctness statement shown below. Theorem 5.1 then follows by the same procedure as for Theorem 4.1.

Lemma 5.3 (Correctness, semantic). *Let* φ *be a first-order ternary relation. If* $\mathcal{E}([\![m]\!]\ \varphi)(e_1,e_2,e_3)$ *then for all* σ, *if* $\langle \mathsf{run}_P\ e_1, \sigma\rangle \to^*_{\mathrm{cf}} \langle (p, v_1), \sigma_1\rangle$ *then* $\langle \mathsf{run}_V\ e_2\ p, \sigma\rangle \to^* \langle \mathsf{Some}\ v_2, \sigma_2\rangle$ $\langle \mathsf{run}_I\ e_3, \sigma\rangle \to^* \langle v_3, \sigma_3\rangle$, *and* $\varphi(v_1,v_2,v_3)$.

6 Optimizations

This section discusses four optimizations to the Authentikit library that reduce the size of the proof stream and/or speed up proving or verification. We show using our logical relation that each of them are secure and correct by adapting the proofs from §4 and §5.

Proof Accumulator. The prover implementation from Figure 3 uses a functional list append in the bind function to combine the proof streams produced by successive authenticated computations. Functional list append is linear in the length of its first argument, and so repeatedly appending to the end of a list in this way leads to quadratic running time. A standard functional programming optimization to avoid this quadratic behavior is to instead *prepend* new proof items to an accumulator that gets reversed before the final proof stream is emitted. An excerpt of the optimization is shown in Figure 8. The verifier remains unchanged and it is therefore only the correctness proof and in turn the interpretation of authenticated computations that needs to be adapted to take the accumulator and list reversal into account.

Reuse Buffering. The $\lambda\bullet$ compiler implements an optimization to reduce the size of the proof stream in cases where the same items may appear in the proof stream multiple times. For example, a client may perform a batch of operations that end up re-traversing many of the same nodes of the ADS, in which case the hashes of these nodes do not need to be repeated multiple times.

```
module Prover =
  type 'a m = proof -> proof * 'a
  (* ... *)
  let unauth evi (a, _) pf = (evi a :: pf, a)
  let run m =
    let pf, res = m [] in (List.rev pf, res)
end
```

Figure 8: Excerpt of the proof accumulator optimization.

We implement a similar optimization by modifying the prover and verifier unauth procedures. An excerpt of the verifier modification is found in Figure 9. Both the prover and the verifier maintains a cache of previously seen hashes. When the unauth function is invoked on an authenticated value, both parties will first consult the cache. If a hash is found, the prover omits adding the proof to the proof stream (because it is not needed), and the verifier returns the deserialization of the value found in the cache. If the hash is not found, a proof will be emitted/consumed and the verifier deserializes and checks the proof. If the check is successful, the result is added to the cache.

By adding proof reuse, correctness of the prover now relies on the collision-free property of the hash function: the prover may return a wrong result if a hash collides with a previously seen hash found in the cache. As such, the correctness theorem only applies to collision-free prover execution traces. While Miller et al. [30] implement the optimization in the $\lambda\bullet$ compiler, their formal model only considers the unoptimized version. In particular, their statement of the correctness theorem does not appear to hold for this optimization, as we alluded to in §5.

To account for this optimization, our security and correctness proofs need to be adapted but the changes are minimal. The main effort involves verifying the cache data structures and adapting the interpretation of authenticated computations to account for the cache and its contents. For example, the verifier cache maps hashes $\mathcal{H}(s)$ to s where s is the serialization of some authenticated value. We record that s was hashed during execution using the hashed(s) resource to account for hash collisions using the collision-free logic.

Heterogeneous Reuse Buffering. The verifier implementation for the previous optimization in Figure 9 performs deserialization when inserting an item into the cache but also when it is retrieved. The second deserialization can be avoided by caching the authenticated value itself rather than its serialization. However, this requires a heterogeneous cache since different authenticated values have different types. In OCaml, this would not be syntactically well typed and in fact Miller et al. [30] does not consider this optimization in their implementation of $\lambda\bullet$. To implement the optimization in OCaml, we have to resort to the `Obj.magic` featue of OCaml that bypasses the typechecker for the cache operations. However, even though this heterogeneous cache is not syntactically well-typed, we prove that it is safe and show that $\mathit{Authentikit}_V$ is still related to the prover and ideal versions in the logical relations at the appropriate type interface. The optimization only requires minimal changes to the security and correctness proof.

Stateful Buffering. The caching mechanisms considered in the previous sections are implemented using purely functional data

```
module Verifier =
  type 'a m = pfstate ->
              [`Ok of pfstate * 'a | `ProofFailure]
  (* ... *)
  let unauth evi h pf =
    match Map.find_opt h pf.cache with
    | None ->
       match pf.pf_stream with
       | [] -> `ProofFailure
       | p :: ps when hash p = h ->
          match evi.deserialize p with
          | None -> `ProofFailure
          | Some a ->
            `Ok ({pf_stream = ps;
                  cache = Map.add h p pf.cache}, a)
       | _ -> `ProofFailure
    | Some p ->
       match evi.deserialize p with
       | None -> `ProofFailure
       | Some a -> `Ok (pf, a)
  (* ... *)
end
```

Figure 9: Excerpt of verifier version of proof-reuse buffering.

structures. For instance, the cache in Figure 9 is passed as an argument to the unauth function. Both OCaml and $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$ are expressive enough to implement and use heap-allocated data structures and so we have also implemented and verified the aforementioned optimizations using a heap-allocated cache.

7 Manual Security and Correctness Proofs

Lemma 4.3 and Lemma 5.3 hold for *any* terms e_1, e_2, and e_3 that satisfy our interpretations of authenticated computations, *i.e.*, terms for which we can derive the associated relational weakest preconditions. These relational weakest preconditions hold automatically for all well-typed clients of the Authentikit library by the fundamental lemmas for the logical relations, which is how we deduced Theorem 4.1 and Theorem 5.1. However, alternatively, if we have manually-written prover and verifier implementations of an operation on an authenticated data structure that does *not* use the Authentikit library, we can instead directly prove that the implementation inhabits the logical relation and therefore satisfies the relational weakest precondition. As the logical relation is *compatible* (*i.e.*, satisfies the compatibility lemmas), this also means that such implementations can be soundly linked with automatically generated code for other operations on the same data structure.

We apply this methodology to an optimized `retrieve` operation on Merkle trees. The optimization addresses a redundancy in the proof objects that `retrieve` generates, which also occurs in the corresponding $\lambda\bullet$ implementation, as noted by Miller et al. [30].[5] For example, calling `retrieve` on the tree from Figure 4 with the path [R, L] produces the proof $[(h_1, h_2), (h_5, h_6), s_5]$ but h_2 and h_5 can be derived from the other proof items and are therefore unnecessary—the minimal proof generated by a standard, manual implementation of Merkle trees only contains h_1, h_6 and s_5. To achieve such minimal proofs, an implementation of retrieve has to break the syntactic typing abstractions of the Authentikit library, and so we cannot

[5] Miller et al. [30] describe a compiler optimization to automatically eliminate such redundancies for $\lambda\bullet$, but the optimization does not have a correctness proof.

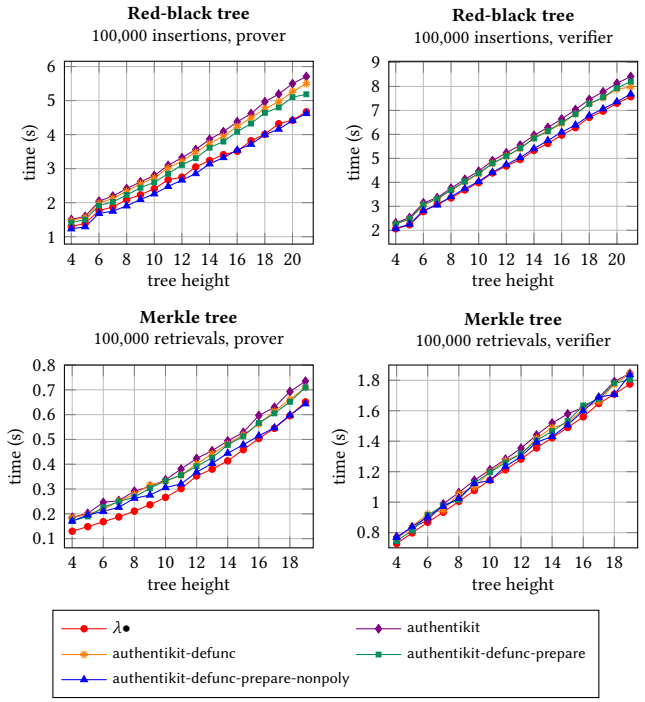

Figure 10: Ablation study of prover and verifier implementations: insertions in red-black trees, retrievals in Merkle trees.

automatically apply Theorem 4.1 and Theorem 5.1 to it. Instead, we prove that such an implementation directly inhabits the logical relation at the appropriate type, *i.e.*, for security we show:

$$[\![\mathsf{path} \to \mathsf{auth\ tree} \to \mathsf{m\ (option\ \ string)}]\!]_\Delta(\mathsf{retrieve}'_V, \mathsf{retrieve}_I)$$

where Δ maps auth, m, and path to their respective interpretations; $\mathsf{retrieve}'_V$ is the optimized verifier implementation of the retrieve operation; and $\mathsf{retrieve}_I$ is an implementation of ordinary binary tree retrieval. We show a similar statement to establish correctness. Both the security and correctness proofs follow by induction on the path and rely on finding a suitable inductive invariant. We refer to the appendix of an extended version of this paper [18] for a description of the implementation and our artifact [19] for details on the proof.

8 Performance Evaluation

Previous sections of this paper have shown how the module-based encoding of Authentikit can be verified for correctness, giving it similar security guarantees as the original custom language approach used by Miller et al. [30] for λ•. In this section, we evaluate how the module approach's performance compares to the custom compiler frontend that λ• uses. In particular, there are at least two potential sources of overhead with the Authentikit approach. First, with Authentikit, the program is written in a functorized, monadic style, whereas the λ• compiler directly inserts calls to the appropriate operations. Second, Authentikit builds up serializers by composing the Authenticatable combinators, while the λ• compiler can statically generate appropriate serialization code for a given datatype. We aim to assess how much overhead (if any) can be attributed to each of these factors.

Experimental Setup. We compare our Authentikit implementation's performance against the λ• compiler on two benchmarks: insertions into authenticated red-black trees and retrievals from Merkle trees. We conduct our experiments using a machine with an Intel i7-4770 3.40GHz CPU and 16 GB RAM. For these experiments, we do not enable any of the special optimizations discussed in §6, or their corresponding analogues in λ•.

Because our goal is to assess just the overhead intrinsic to the module-based encoding, we make several changes to Authentikit to isolate just this factor for comparison. First, we compile Authentikit with OCaml 4.01, which is the version of OCaml that the λ• compiler is forked from. Second, because λ• reads and writes proof streams from a file incrementally, instead of storing the entire proof stream as a list, for these experiments we change the implementation in Authentikit to similarly read/write proofs from files. Finally, since λ• uses OCaml's built-in Marshal library for serialization, we also change Authentikit to call this library from the combinators in the Authenticatable module, instead of the version of serialization verified in Rocq.[6]

Results. In Figure 10, we see that for 100,000 insertions in red-black trees and 100,000 retrievals in Merkle trees, λ• prover and verifier running times are shorter than for Authentikit. To understand the source of this performance gap, we also benchmark several transformations to the Authentikit code that successively bring it closer to the implementation generated by the λ• compiler.

- **authentikit-defunc**: this version de-functorizes the implementations by manually inlining the prover and verifier code respectively into the operation code.
- **authentikit-defunc-prepare**: this version removes the use of the Authenticatable combinators for serializing, and instead directly implements a serialization function for the specific tree data type which calls the Marshal library.
- **authentikit-defunc-prepare-nonpoly**: this version replaces the use of polymorphic variants with non-polymorphic variants for representing the tree datatypes.

These changes essentially entirely remove the performance gap between Authentikit and λ•. Some of these transformations could potentially be done automatically in different versions of the OCaml compiler. For example, the `flambda` optimizer in newer versions of OCaml supports hints for inlining.

The use of polymorphic variants also makes up a substantial part of the performance gap. Polymorphic variants come with some overhead compared to non-polymorphic variants, and in particular have a larger in-memory representation and a larger serialized format when using the Marshal library. But, by default in OCaml, polymorphic variants are needed for the recursive typing required to use the `evi` type combinators with recursive data structures like trees. However, with the `-rectypes` OCaml compiler option enabled, non-polymorphic variants could also be used, thereby avoiding this overhead.

[6]The Marshal serializer cannot be called directly on data that contains `auth` types as sub-components in the prover, since this would serialize both the underlying data and the hash. Instead, a preprocessing step drops the non-hash components before calling Marshal's serialize, similarly to line 14 of Figure 3a.

9 Related Work

Verification of Authenticated Data Structures. Merkle trees are the canonical ADS but Miller et al. [30] also use $\lambda\bullet$ to implement Red-black+ trees, skip lists, and planar-separator trees, among others. All of these data structures are directly portable to Authentikit. Miraldo et al. [31] verify a particular notion of authenticated append-only skip lists in Agda and indicate that it does not seem possible to encode their implementation in $\lambda\bullet$ because of the type-directed hashing discipline. It would be interesting future work to apply our semantic approach to show that their implementation can be safely linked with code that is generated using Authentikit. Brun and Traytel [7] mechanize the proofs of security and correctness for the core calculus of $\lambda\bullet$ given by Miller et al. in Isabelle/HOL. They identify and resolve several minor technical issues in the original proofs. As discussed in §1, the $\lambda\bullet$ approach—and therefore also Brun and Traytel's formalization —has three important limitations as compared to our work: (1) a custom compiler frontend is needed, (2) optimizations are not covered by the security/correctness theorems, and (3) hand-written optimizations cannot be verified and integrated with automatically generated code.

Lochbihler and Maric [28] show how to systematically and modularly derive ADSs as data types in Isabelle/HOL using so-called Merkle functors. The construction comes with no formal security or correctness guarantees. They point out that HOL's lack of abstraction over type constructors (which are supported by OCaml and $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$) hinders expressing their process in its full generality.

Sato et al. [39] implement and verify a variant of Merkle Patricia Trees in F*. They show that each of their tree-manipulating functions are functionally correct and that the hashing scheme is collision resistant by a reduction to collision resistance of the hash function being used. Arasu et al. [3] use a sparse, incremental Merkle tree to authenticate the state of a verified key-value store and show sequential consistency up to hash collision. In contrast to both of these efforts, our proof guarantees security and correctness for *all* syntactically well-typed functions.

Logical Relations. Relational parametricity and free theorems for languages with higher kinds have been studied using several different approaches [4, 22, 36, 51, 52]. Our logical-relations model takes inspiration from recent work [40] that developed a model of a type system with generalized algebraic data types. In addition, our Rocq formalization builds on their formalization of intrinsically well-kinded types that uses a notion of functorial syntax [33].

Logical relations have been used to establish a variety of different security properties of expressive type systems. Frumin et al. [12] develop a relation to establish a timing-sensitive notion of noninterference for an information-flow control type system. Gregersen et al. [21] similarly develop a model for a termination-insensitive notion of noninterference. Georges et al. [14, 15], Strydonck et al. [41] and Swasey et al. [42] use logical relations to establish so-called robust safety properties that formalize the security guarantee offered by a capability machine. Legoupil et al. [27] use a logical relation to show robust capability safety of a WebAssembly extension and Sammler et al. [38] consider a sandboxing mechanism.

Prophecy Variables. Prophecy variables were originally developed by Abadi and Lamport [1] to establish certain program refinements that require speculative reasoning about future events. They are commonly used in concurrent program verification for reasoning about future-dependent linearization points but have since then also found use in relational reasoning that requires alignment [2, 50]. Frumin et al. [13] show how to integrate prophecy variables in a logical relation for contextual refinement of concurrent programs. de Vilhena et al. [10] consider several (unary) examples where prophecy variables are seemingly required in verifying deterministic and sequential code, including a proof of a structural infinitary conjunction rule for separation logic. To our knowledge, our work is the first to use prophecy variables to show free theorems.

10 Conclusion

Authenticated data structures allow untrusted third parties to carry out operations which produce proofs that can be used to check that the results of the operation are valid. In this work, we showed how the Authentikit library generates secure and correct authenticated data structures automatically.

Our proof uses a new relational separation logic for reasoning about programs that use collision-resistant cryptographic hash functions. We use the logic as a basis for constructing two logical-relations models of $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$ that are expressive enough to show security and correctness of authenticated data structures generated using Authentikit. The correctness proof, in particular, relies on a novel use of prophecy variables. Finally, we also showed how to use our models to prove security and correctness of four optimizations to the library and how optimized, hand-written implementations of authenticated data structures can be soundly linked with automatically generated code.

Acknowledgments

This work was supported in part by the National Science Foundation, grant no. 2225441 and the Carlsberg Foundation, grant no. CF23-0791.

References

[1] Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. 165–175. doi:10.1109/LICS.1988.5115

[2] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 2023. An Algebra of Alignment for Relational Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 573–603. doi:10.1145/3571213

[3] Arvind Arasu, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Aymeric Fromherz, Kesha Hietala, Bryan Parno, and Ravi Ramamurthy. 2023. FastVer2: A Provably Correct Monitor for Concurrent, Key-Value Stores. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*. 30–46. doi:10.1145/3573105.3575687

[4] Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. 46–61. doi:10.4230/LIPICS.CSL.2012.46

[5] Robert Atkey. 2016. Authenticated Data Structures, as a Library, for Free! https://bentnib.org/posts/2016-04-12-authenticated-data-structures-as-a-library.html. [Accessed 27-11-2024].

[6] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2016. Relational Logic with Framing and Hypotheses. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*. doi:10.4230/LIPICS.FSTTCS.2016.11

[7] Matthias Brun and Dmitriy Traytel. 2019. Generic Authenticated Data Structures, Formally. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. doi:10.4230/LIPICS.ITP.2019.10

[8] Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. 2016. System F-Omega with Equirecursive Types for Datatype-Generic Programming. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 30–43. doi:10.1145/2837614.2837660

[9] Karl Crary. 2017. Modules, abstraction, and parametric polymorphism. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 100–113. doi:10.1145/3009837.3009892

[10] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy game: verifying a local generic solver in Iris. *Proc. ACM Program. Lang.* 4, POPL (2020), 33:1–33:28. doi:10.1145/3371101

[11] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* 7, 2 (2011). doi:10.2168/LMCS-7(2:16)2011

[12] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. Compositional Non-Interference for Fine-Grained Concurrent Programs. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. 1416–1433. doi:10.1109/SP40001.2021.00003

[13] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). doi:10.46298/LMCS-17(3:9)2021

[14] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (2024), 3:1–3:59. doi:10.1145/3623510

[15] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. doi:10.1145/3527318

[16] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *Proc. ACM Program. Lang.* 4, ICFP (2020), 114:1–114:29. doi:10.1145/3408996

[17] Jean-Yves Girard. 1972. *Interprétation Functionelle et Élimination des Coupures Dans L'arithmétique D'ordre Supérieure.* Ph. D. Dissertation. Université Paris VII.

[18] Simon Oddershede Gregersen, Chaitanya Agarwal, and Joseph Tassarotti. 2025. Logical Relations for Formally Verified Authenticated Data Structures. arXiv:2501.10802 [cs.LO] https://arxiv.org/abs/2501.10802

[19] Simon Oddershede Gregersen, Chaitanya Agarwal, and Joseph Tassarotti. 2025. *Logical Relations for Formally Verified Authenticated Data Structures - Rocq and OCaml artifact.* doi:10.5281/zenodo.15551944

[20] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL (2024), 753–784. doi:10.1145/3632868

[21] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized Logical Relations for Termination-Insensitive Noninterference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. doi:10.1145/3434291

[22] Ryu Hasegawa. 1994. Relational Limits in General Polymorphism. *Publications of the Research Institute for Mathematical Sciecnes* 30 (1994), 535–576.

[23] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. doi:10.1145/363235.363259

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154

[25] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris From the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

[26] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. doi:10.1145/3371113

[27] Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. 2024. Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 304–332. doi:10.1145/3689722

[28] Andreas Lochbihler and Ognjen Maric. 2020. Authenticated Data Structures as Functors in Isabelle/HOL. In *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20-21, 2020, Los Angeles, California, USA (Virtual Conference)*. 6:1–6:15. doi:10.4230/OASICS.FMBC.2020.6

[29] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 293)*, Carl Pomerance (Ed.). Springer, 369–378. doi:10.1007/3-540-48184-2_32

[30] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated Data Structures, Generically. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 411–424. doi:10.1145/2535838.2535851

[31] Victor Cacciari Miraldo, Harold Carr, Mark Moir, Lisandra Silva, and Guy L. Steele Jr. 2021. Formal verification of authenticated, append-only skip lists in Agda. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. 122–136. doi:10.1145/3437992.3439924

[32] Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. 255–266. doi:10.1109/LICS.2000.855774

[33] Piotr Polesiuk and Filip Sieckowski. 2024. Functorial Syntax for All. (2024). https://popl24.sigplan.org/details/CoqPL-2024-papers/7/Functorial-Syntax-for-All The 10th International Workshop on Coq for Programming Languages.

[34] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. 513–523.

[35] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. doi:10.1109/LICS.2002.1029817

[36] Edmund P. Robinson and Giuseppe Rosolini. 1994. Reflexive Graphs and Parametric Polymorphism. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. 364–371. doi:10.1109/LICS.1994.316053

[37] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing Modules. *J. Funct. Program.* 24, 5 (2014), 529–607. doi:10.1017/S0956796814000264

[38] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The High-Level Benefits of Low-Level Sandboxing. *Proc. ACM Program. Lang.* 4, POPL (2020), 32:1–32:32. doi:10.1145/3371100

[39] Sota Sato, Ryotaro Banno, Jun Furuse, Kohei Suenaga, and Atsushi Igarashi. 2021. Verification of a Merkle Patricia Tree Library Using F. *CoRR* abs/2106.04826 (2021). arXiv:2106.04826 https://arxiv.org/abs/2106.04826

[40] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. 2024. The Essence of Generalized Algebraic Data Types. *Proc. ACM Program. Lang.* 8, POPL (2024), 695–723. doi:10.1145/3632866

[41] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. 2022. Proving full-system security properties under multiple attacker models on capability machines. In *35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022*. 80–95. doi:10.1109/CSF54842.2022.9919645

[42] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. doi:10.1145/3133913

[43] Roberto Tamassia. 2003. Authenticated Data Structures. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*. 2–5. doi:10.1007/978-3-540-39658-1_2

[44] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. doi:10.1007/978-3-662-54434-1_34

[45] The Coq Development Team. 2024. *The Coq Proof Assistant.* doi:10.5281/zenodo.11551307

[46] Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 3, ICFP (2019), 105:1–105:28. doi:10.1145/3341709

[47] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6, Article 40 (Nov. 2024), 75 pages. doi:10.1145/3676954

[48] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *Proc. ACM Program. Lang.* 2, POPL (2018), 64:1–64:28. doi:10.1145/3158152

[49] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 377–390. doi:10.1145/2500365.2500600

[50] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. 742–766. doi:10.1007/978-3-030-81685-8_35

[51] Janis Voigtländer. 2009. Free theorems involving type constructor classes: functional pearl. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. 173–184. doi:10.1145/1596550.1596577

[52] Dimitrios Vytiniotis and Stephanie Weirich. 2010. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.* 20, 2 (2010), 175–210. doi:10.1017/S0956796810000079