AuraConf: A Unified Approach to Authorization and Confidentiality

Jeffrey A. Vaughan
University of California, Los Angeles

Abstract

This paper introduces AuraConf, the first programming language with a unified means to specify access-control and confidentially policies. In concert with a proof-carrying access control mechanism, AuraConf allows confidentially policies to be specified declaratively using types and enforced via cryptography. Programs written in AuraConf enjoy a formal security guarantee via noninterference. Additionally, the language definition introduces a novel type system where the typechecker may use resources (i.e., private keys) and knowledge of an object's provenance (i.e., how a ciphertext was computed) to guide analysis.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Languages, Security, Theory

Keywords Dependent types, information flow, cryptography

1. Introduction

Language based security research has made great strides in addressing the complementary problems of enforcing access-control restrictions and preventing unintentional disclosure of confidential data. Indeed, ongoing research into access-control logics and proof-carrying access control [1, 9, 14, 20] have realized an expressive, flexible framework for describing and enforcing authorization policies. A largely orthogonal body of research uses information flow analysis to detect unsafe, in a precise sense, uses of confidential information within a program. Such techniques can ensure that secrets are appropriately encrypted—manually by the programmer [3, 15, 22, 23, 33] or automatically by the language framework [5, 39]—before (e.g.) transmission on an insecure network.

Information flow analysis (with encryption) and proof-carrying access control represent first-rate, language-based approaches to mitigating different security problems. Because programs may deal both with confidential data and access-controlled resources, it is natural to ask: *Can these techniques be combined?*

Yes. This paper introduces AuraConf, a programing language that provides direct support for both proof-carrying access control and information flow with automatic encryption. The specific goals of this design are as follows.

- To establish a natural connection between information-flow analysis for policy specification and cryptography for policy enforcement.
- To unify these confidentiality mechanisms with proof-carrying access-control techniques as realized in Aura.
- To provide technical mechanisms for anomaly detection, including audit of relevant security events and static discovery of security errors via typing.

AuraConf is built as an extension to the Aura [20, 37, 38] programming language. Plain Aura is a platform for programming with access control and audit. Programs construct proofs of their access-control rights at runtime, and such proofs are consumed and logged by procedures that perform secure operations. Dependent types provide a precise way to connect access-control predicates with specific data values. And, mutually distrusting principals may use digitally signed propositions to introduce new access-control policies.

Aura is a solid foundation for the present work because its design explicitly captures core notations of proof-carrying access control. Additionally, Aura's expressive type system provides many components, including weak dependent types and a monadic programing style that, as we will discuss, are useful for reasoning about confidentiality. Aura is, for these reasons, a practical framework to investigate the general question of how to unify language-based approaches to confidentiality and authorization.

Mixing an informative type system with encryption, as done in AuraConf, is an enabling technology for trustworthy application development. For instance, consider a distributed streaming-movie service. Developers of such a system must address a variety of security-focused issues.

- *Authorization Policy:* Who can access a movie? When can access-rights may be transferred between users?
- *Confidentiality Policy:* What data is secret? Movie ratings and reviews, or only the bits encoding videos?
- *Enforcement and Audit Mechanisms:* What data should be encrypted? When should encryption occur? When runtime decryption failures occur, what does this mean?

While Aura provides mechanisms for answering only the authorization-policy questions, AuraConf provides a framework in which all the issues may be addressed. Confidentiality policies are specified as types, and the language ensure that confidential information is automatically encrypted at system boundaries. Decryption failures are always accompanied by a proof-object indicating system components that may be faulty or malicious; note the mechanisms used to allow precise audit of such anomalous, confidentiality-related events are defined in terms of Aura's existing access-control features.

TLDI'11, January 25, 2011, Austin, Texas, USA.

The mixture of encryption with an expressive type system exposes a fundamental tension. Type systems gain power—the ability to prevent errors and uphold invariants—by exploiting precise information about a program's terms. In contrast, the point of encryption is to obscure information in certain contexts.

This tension has several technical manifestations. First, typing is relative; each principal has its own, local notion of what is well-typed. This is desirable because it accounts for the following real phenomenon. To Alice all arbitrary, unknown bit strings are plausibly encrypted messages for Bob—elements of the AuraConf type **int for** Bob. In contrast Bob can tell which bit strings are well-formed at that type, and which are garbage.

Second, typing exhibits a hysteretic, or path dependent, effect. When Alice creates a new ciphertext for Bob, she transforms a perfectly legible piece of abstract syntax into an opaque binary blob. In order for type preservation to hold during this process, Alice's computation must annotate the ciphertext and, as a side-effect, record information to validate the annotation in the future.

Third, resolving the above issues requires a precise treatment of public keys, both at compile time and at runtime. Discussing key availability at different hosts requires ideas from modal type theories [21, 27]. Ensuring that needed keys are available dynamically requires type-and-effect analysis [26, 36]. (In principle other techniques could be used, but the concepts involved are essential.)

The AuraConf language resolves the tensions indicated above and helps programmers both handle confidential data and manage access-control credentials. Meeting these goals requires three substantial technical contributions:

- The design of AuraConf including the confidentiality type constructor **for** and a sophisticated type system that enforces both key-management and code-mobility constraints.
- A mechanized proof that AuraConf satisfies type safety and that type checking is decidable.
- A mechanized proof showing AuraConf programs protect confidential data; more precisely, they satisfy noninterference.

The rest of this paper is structured as follows. Section 2 describes AuraConf's new constructs informally, and Section 3 presents a sample program exercising these features. Section 4 summarizes the formal language definition and metatheory. Section 5 discusses AuraConf from the perceptive of conventional information flow. Section 6 reviews related work and Section 7 concludes.

2. Overview of AuraConf

This section provides an informal introduction to AuraConf's new features.

Access control in Aura

The core Aura language [20, 38] is designed to add support for authorization policy specification, runtime enforcement, and audit to a functional programming language. Aura security policies are expressed as propositions in an authorization logic. Aura's type system cleanly integrates standard data types (like integers) with proofs of authorization logic propositions, and programs manipulate authorization proofs just as they might other values.

In Aura, the proposition A **says** P denotes "Principal A says (or endorses) proposition P." Aura authorization proofs serve as *evidence* of access-control decisions, and programs must present appropriate proofs in order to access resources. Evidence is composed of a mix of cryptographic signatures, which capture principals' "utterances," and standard rules of logical deduction.

Aura has ML-like evaluation semantics, characterized by call-by-value reduction and effectful operators. Its static semantics are substantially more novel and are based on *weak dependent types* that can express a variety of useful propositions. For instance, the proposition

```
Alice says ((P: Prop) → Bob says P → P)
```

means that principal Alice will endorse any proposition that Bob endorses. This proposition, which may be pronounced "Bob speaks for Alice," means that Alice is delegating *all* of her authority to Bob. In contrast, the proposition

```
Alice says ((s: Song) → isJazz s →
            Bob says (MayPlay Bob s) → MayPlay Bob s)
```

describes a more limited form of delegation, where Alice delegates some, but not all, of her authority to Bob (only her rights to play jazz songs). The latter proposition follows the principle of least privilege and represents a safer, more secure form of delegation than the former. The ability to express such restricted delegation is an advantage of Aura compared with simpler authorization logics containing only polymorphism.

The Aura runtime system automatically logs proofs used for access-grants, and logged evidence enables useful post-hoc analysis of the authorization decisions made during a system's execution.

Confidential Computations and the For-Monad

AuraConf integrates the mechanisms described above with features for specifying and automatically enforcing confidentiality constraints.

In AuraConf secrets are protected with an indexed confidentiality monad. A confidential integer intended only for Alice can be given type (**int for** Alice). As expected, values of this type are constructed using the following monadic return operator.

```
return Alice 42 : int for Alice
```

This expression evaluates by encrypting 42 with Alice's public key, yielding a blob of ciphertext, written $\mathcal{E}$(Alice, 42, 0x2b63), with an additional annotation that will be discussed shortly. The number 0x2b63 represents a random value inserted by the encryption algorithm to ensure that encrypting identical plaintexts does not yield identical ciphertexts. Code running on any host should be able to perform the **return** operation, as it uses only Alice's public key and needs no access to private keys. A program running with Alice's private key may decrypt and declassify the ciphertext as follows.

```
run (return Alice 42) : int
```

Additionally, when given a value of type **int for** Alice, AuraConf programs can use a bind operator to produce a new encrypted computation, also for Alice, based on the existing secret.

```
bind (int for Alice)
     (return Alice 42)
     (λ{Alice} x: int . return Alice (x * 2))
  : int for Alice
```

When Alice runs the resulting encrypted computation, she will decrypt the 42 before supplying it to the decryption of the function. For now, it's ok to ignore the {Alice} component of the λ; this is an effect annotation and will be defined and discussed later.

As illustrated above, for-monad operators treat their arguments lazily. Imagine for the moment that we could use homomorphic encryption [16, 31] to allow for-bound computations to be applied eagerly. (I am not aware, by the way, of any practically efficient homomorphic encryption scheme.) An eager for-bind would permit curious adversaries to probe encrypted objects using functions that diverge on known inputs. Giving the for-monad lazy semantics eliminates this timing channel.

While the dynamic semantics of encryption are straightforward, they pose a substantial problem for typechecking. Consider a machine running on Bob's behalf that performs the above encryption for Alice. A sound type system should satisfy subject reduction and be able to relate ciphertext $\mathcal{E}$(Alice, 42, 0x2b63) with type **int for** Alice. But the entire point of encryption is to ensure that users other than Alice cannot meaningfully inspect the ciphertext, and Bob cannot decompose or examine the newly created object.

AuraConf resolves this tension as follows. Ciphertexts may be annotated with one of two forms of typing metadata. First, the term

```
cast 𝓔(Alice, 42, 0x2b63) to (int for Alice)
  : int for Alice
```

is a *true cast*—a form of type coercion allowed only when semantic evidence indicates that the cast is "correct." A true cast typechecks when the ciphertext is a known value with known provenance. Whenever Bob's program creates a ciphertext, it records a *fact* associating the new ciphertext with the appropriate type. As evaluation proceeds, programs accumulate a context of facts which are used to typecheck known ciphertexts. We assume fact contexts are part of a host's local state and are not shared between different principals. True casts are also permitted when the typechecker can *statically* access an appropriate decryption key. Thus the above cast can be typechecked on Bob's machine, where it originated, as well as on Alice's machine, where it will be used. Evaluating a **return** yields a ciphertext annotated with a true cast.

True casts alone are insufficient for writing some protocols. Consider two programs, running with Bob and Charlie's authority respectively, jointly constructing an **int for** Alice using the **return** and **bind** operators. In particular, Charlie's program may need to **bind** a ciphertext previously created by Bob. Because facts are not shared between different principals, and because Charlie cannot access Alice's private key, there's no way Charlie will be able to typecheck the ciphertext annotated with a true cast. Instead, Charlie's program will need to work with a justified cast,

```
cast c to (int for Alice) blaming p : int for Alice
```

where p is a proof that ciphertext c has the correct form. Concretely,

```
p : (Bob says (c isa (int for Alice))).
```

Proposition constructor **isa** is a built-in constant with the job of witnessing these justified casts.

In combination, true and justified casts allow us to reason about ciphertexts, even those which cannot be decrypted in a particular context. Subject reduction ensures that (for suitable fact contexts) decryption never fails for true-cast ciphertexts. Furthermore, while justified casts may lead to decryption failures, such failures are accompanied by signed **isa** proofs that can be used to assign blame. Observe that the justified cast mechanism and the very notion of blame rest Aura's ability to capture principal intent via **says** types.

Casting allows the programmer to assign a precise type to ciphertext. Conversely, **asbits** strips a ciphertext's annotation, resulting in a term with the following less informative type.

```
asbits (cast(𝓔(Alice, 42, 0x2b63)) to (int for Alice))
  : bits
```

Type **bits** classifies naked ciphertexts, and this term reduces to

```
𝓔(Alice, 42, 0x2b63) : bits.
```

3. Examples

This section shows sample AuraConf programs. For illustration, we use modules—a feature that is not part of the formal language definition. Additionally, Figure 2 uses syntactic sugar for writing recursive functions; AuraConf supports general recursion (via a datatype-base encoding) but does not have this convenient syntax.

```
(* Interface providing a basic networking API *)
Signature NetIO

  assert OkToSend: prin → Type → Prop;
  val attempt_acquire_strong_credential :
        (b: prin) →
        Maybe ((a: prin) →
        (T: Type) →
        pf (b says OkToSend a T) →
        pf (Kernel says OkToSend a T))

  val recv: (T: Type) → T

  val send: (T: Type) → (a: prin) → T →
                pf (Kernel says (OkToSend a T)) → Unit
End Signature
```

Figure 1: A simple communications library.

Figure 1 defines a simple networking interface. The functions send and recv are intended to send and receive data values. In addition to data to transmit, send consumes a proof that the system (that is the principal Kernel) permits the operation. Concretely

```
send int Bob 42 p
```

sends the data value 42 to principal Bob when p is an appropriate access-control proof. Note that both confidential and non-confidential values may be transmitted over any channel.

Assertion OkToSend and function attempt_acquire_strong_credential define an access control policy for the send function. This function allows client b to request a proof object permitting arbitrary network writes.

Suppose that a program running with Alice's authority needs to build a secret message that will eventually be read by Bob after being processed, and possibly for-bound, by Charlie. We can write this program as follows.

```
Module Sender Of Alice
  open NetIOImp

  let msg at ⊥=
        let x at Bob = return 312 as (int for Bob) in
        let y at ⊥= asbits x in
          cast y to (int for Bob)
                blaming (say Alice (y isa (int for Bob)))
  in
  send (int for Bob) Charlie msg (get_cred msg)

End Sender
```

The program creates an annotated ciphertext for Bob—with the form of a true cast—strips its annotation with **asbits**, and creates a justified cast suitable for sending. The true cast is stored in x, whose **at** Bob annotation reflects that the true cast may only be typed in certain contexts—those where Bob's key or relevant facts are available. In contrast, y and msg may be interpreted anywhere; this is reflected by the **at** ⊥ annotations. Finally, the get_cred function is assumed to return a proof granting permission to send. Even this simple program relies on the harmonious interaction of several language features: both **cast**s, **asbits**, **return**, **isa**, and **let at**.

The Sender module is annotated with **Of** Alice, indicating that it defines code that will be typechecked and run on behalf of principal Alice. In the terminology of Section 4, the module's top-level terms must be typechecked with statically available key and effect label

```
Module NetworkedStore Of Server

(* Use the a module which implements the NetIO interface. *)
open NetIOImp: NetIO;

(* The type of network requests to this server *)
data request: Type {
| r_put: (a: prin) → (id: Nat) → String for a → request
| r_get: prin → Nat → request }

(* The Map datatype stores for each principal a natural-
   number-indexed set of confidential Strings. *)
data Map: Type {
| m_intro : ((a: prin) → Nat →
             Maybe (String for a)) → Map }

let empty_map: Map =
      m_intro (λa: prin. λid: Nat. nothing (String for a))

let lookup: Map → (a: prin) →
              Nat → Maybe (String for a) =
  λm: Map. λa: prin. λn: Nat.
  match m with Maybe (String for a) {
  | m_intro → λf: ((a: prin) → Nat →
                   Maybe (String for a)). f a n }

let insert: Map → (a: prin) → Nat →
                  (String for a) → Map =
  λm: Map. λa: prin. λid: Nat. λmsg: String for a.
    m_intro (λa′: prin. λ id′: Nat.
               if a = a′
                 then
                   match eqnat id id′ with
                             Maybe (String for a′) {
                   | true → ⟨just (String for a) msg:
                                  Maybe(String for a′)⟩
                   | false → lookup m a′ id′ }
                 else lookup m a′ id′)

(* A helper function that lets that lets us compose functions of
   type T → S with the for monad. Bind alone only works
   with T → (S for a) functions. *)
let for_lift : (T: Type) → (S: Type) → (a: prin) →
                 (T → S) → T for a → S for a =
  λT: Type. λS: Type. λa: prin. λf: T → S.
    λx: T for a.
      bind y = x
        in (return (f y) as (S for a)) as (S for a)

(* rewrite_cred uses say and a proof signed by the kernel to
 * produce a new, more useful proof access-control proof. *)
let rewrite_cred : ((a: prin) → (T: Type) →
                      pf (self says OkToSend a T) →
                      pf (Kernel says OkToSend a T)) −{self}→
                    ((a: prin) → (T: Type) →
                      pf (Kernel says OkToSend a T)) =
  λ{self} p1: ((a: prin) → (T: Type) →
                pf (self says OkToSend a T) →
                pf (Kernel says OkToSend a T)).
    let p2: pf (self says ((b: prin) →
                  (S: Type) → OkToSend b S)) =
              say ((b: prin) → (S: Type) → OkToSend b S) in
    let p3: (b: prin) → (S: Type) →
                  pf (self says (OkToSend b S)) =
              λb:prin. λS:Type.
                bind p2
                  (λp2′: self says (
                        (b: prin) → (S:Type) → OkToSend b S).
                     return (bind p2′
                                (λp2′′: (b: prin) → (S:Type) →
                                                    OkToSend b S.
                                   return self (p2′′ b S))))
      in
    λc: prin. λU: Type. p1 c U (p3 c U)

(* attempt_acquire_credential gets a proof allowing access to the send
   function and rewrites it into a useful form using rewrite_cred. *)
let attempt_acquire_credential : Unit −{self}→ Maybe ((a: prin) →
    (T: Type) → pf (Kernel says OkToSend a T)) =
      λ{self} x: Unit.
        match (attempt_acquire_strong_credential self) with
            Maybe ((a: prin) → (T: Type) →
                     pf (Kernel says OkToSend a T)) {
        | just → λ{self} p: (a: prin) → (T: Type) →
                              pf (self says OkToSend a T) →
                              pf (Kernel says OkToSend a T).
                    just ((a: prin) → (T: Type) →
                                       pf (Kernel says OkToSend a T))
                         (rewrite_cred p)
        | nothing → nothing ((a: prin) → (T: Type) →
                                  pf (Kernel says OkToSend a T)) }

(* The main server loop. This reads input requests from the network
   and stores or retrieves confidential values as needed. *)
let server_loop: ((a: prin) → (T: Type) →
                    pf (Kernel says OkToSend a T)) →
                    Map → Unit =
  λp: (a: prin) → (T: Type) →
        pf (Kernel says OkToSend a T).
  fun rec: Map → Unit =
    λm: Map. rec
      match recv request with Map {
      | r_put → λa: prin. λid: Nat.
                  λmsg: String for a. insert m a id msg
      | r_get → λa: prin. λid: Nat.
                  let u: Unit =
                    match lookup m a id with Unit {
                    | just → λmsg: String for a.
                                send (String for a) a
                                     ( for_lift String String
                                               a time_stamp msg)
                                     (p a (String for a))
                    | nothing → send (String for a) a
                                     (String a "not found")
                                     (p a (String for a)) }
                  in m }
    in rec end

(* This code starts the server loop after acquiring necessary
   credentials. If such credentials are not available, it fails. *)
match attempt_acquire_credential unit with Unit {
| just → λp: ((a: prin) → (T: Type) →
                pf (Kernel says OkToSend a T)).
            server_loop p empty_map
| nothing → unit }

End Module
```

Figure 2: Code for confidential storage server.

both equal to singleton world Alice. Furthermore, the code must be run with authority Alice.

Figure 2 illustrates a larger program that uses the NetIO interface to implement a storage server. Clients use the NetworkedStore program to store encrypted objects. Each principal has a storage area with a set of slots indexed by natural numbers. A request of form r_put Alice 3 v instructs the server to store value v (of type String **for** Alice) in principal Alice's third storage cell. This value can be retrieved with request r_get Alice 3. The server allows anyone to store or retrieve data from any storage location, even those belonging to another principal. Confidentiality of slot contents is maintained by the use of **for**-types and encryption. It is also possible to add a layer of proof-based access-control to limit access to ciphertexts.

At the heart of this example is the server_loop function. This reads incoming requests from the network and adds values to, or finds values in, the store. In the case of an r_get request, the loop adds a time stamp to the retrieved value (Line 111). Note that whether or not returned values are timestamped does not affect the type of the resulting object; in general the ability to compose computations with ciphertexts allows the server's behavior to change without breaking existing interfaces. This composition is made possible by function for_lift which is defined using **bind** (Lines 43–48).

The storage server must acquire access-control proofs and rewrite them into useful forms. Function attempt_acquire_credential (Lines 78–91) attempts to get a proof, which permits liberal use of send, from the network module's attempt_acquire_strong_credential function. Success yields a proof with the form of a delegation:

```
(a: prin) → (T: Type) → pf (Server says OkToSend a T) →
  pf (Kernel says OkToSend a T).
```

We read this proposition as delegation because it transforms an access control statement by Server into a statement by Kernel; that is, Server principal is "speaking for" Kernel. This is rewritten to a simpler form using rewrite_cred (Lines 52–52). Function rewrite_cred uses **say** to create a fresh (Server says ...) proof and compose it with the delegation above. Evaluating **say** requires Server's private key, and this fact is recorded as a latent effect in rewrite_cred's type. The following section discusses effects annotations is more detail.

4. Language Definition

This section describes the definition and metatheory of AuraConf.

In type-safe languages such as AuraConf, a conservative algorithm identifies and rejects programs that might go wrong—that is crash—at runtime. There are many ways that a program can crash, such as by accessing a memory location out of scope or jumping to an invalid instruction sequence. AuraConf's type system, like Aura's or ML's, rules out these particular errors. However, an AuraConf program could potentially go wrong in several other ways, and the type system must address the following two challenges just to ensure soundness.

Challenge 1
Ensure decryption failures—in which a ciphertext cannot be decrypted to a well-typed plaintext—only occur where a proof can be used to assign blame. Failures without such proofs constitute undefined behavior.

Challenge 2
Ensure that running programs only (attempt to) use private keys that are actually available at runtime. Programs that require unavailable keys for decryption or signing are stuck.

Worlds

W, V, U	::=	$\bot$	Bottom world (no keys)
	|	t	Singleton worlds
	|	$\top$	Top world (all keys)

Terms

t	::=	...	(Standard functional programming)
	|	$(x{:}t) \to_{\{W\}} t$	Implication, quantification, and function arrow
	|	$\lambda_{\{W\}} x{:}t.\ t$	Abstraction
	|	a **for** P	Type of encrypted data
	|	$\mathcal{E}(a, e, n)$	Ciphertext $(n \in \mathbb{N})$
	|	**return**$_{\mathbf{f}}$ e **as** t	Private data e with type t
	|	**bind**$_{\mathbf{f}}$ $x = e_1$ **in** e_2 **as** t	Private computation
	|	**run**$_{\mathbf{f}}$ e	Extract private data
	|	**cast** e **to** t **blaming** p	Cast using type-evidence
	|	**cast** e **to** t	Empirical cast
	|	**fail** p	Decryption-failed exception
	|	a **says** P	Proposition "a affirms P"
	|	**say** $a\ P$	Direct affirmation
	|	**return**$_s$ $a\ e$	Affirmation when given proof
	|	**bind**$_{\mathbf{S}}$ $e_1\ e_2$	Says composition

Figure 3: AuraConf Syntax

We address the first challenge by constraining the canonical forms of **for** types. Enforcing these constraints requires that types and terms have (loosely) consistent meanings to typecheckers with different capabilities, i.e. different access to private keys. AuraConf's type system accomplishes this using ideas based on modal type systems for distributed computing [21, 27, 28].

We address the second challenge by statically tracking the use of **say** and **run**, the only operators that use private keys, and ensuring that required keys will be accessible at runtime. To do so, we blend ideas from modal type systems with those from type-and-effect analysis [26, 36].

Types, propositions, and core Aura We pause to briefly summarize important features of core Aura's language design. Vaughan [37] describes this in detail.

Following the Curry-Howard correspondence [12, 19], Aura propositions are expressed as programming language types. For instance Aura's $\to$ type constructor can express several different concepts. Intuitively, (s: Song) → P → Q is analogous to the universally quantified formula $\forall s \in Song.\ P \Rightarrow Q$. Proofs of propositions are given a programming-language terms. For instance, λx: P. x is a proof of P → P.

At the same time $\to$ can be interpreted as an ML-style function arrow. Thus the factorial function might have type **int** → **int**. A kinding relation keeps track of whether $\to$ should be interpreted as logical implication or as function arrow. The kind **Prop** classifies propositions while the kind **Type** classifies computations. Aura supports general recursion for **Type**s, and the **Type**-**Prop** distinction is needed to prevent diverging computations from being unsoundly confused with valid proofs. Logical consistency of Aura's **Prop** fragment is a corollary of a strong normalization result by Jia and Zdancewic [22].

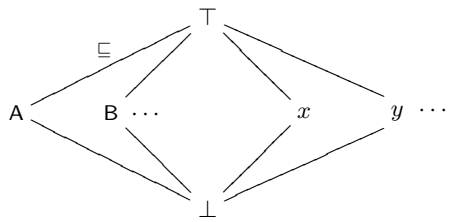
Syntax AuraConf's syntax is summarized in Figure 3, and includes the new operators introduced in Section 2. Not shown are standard functional-programming constructs, like pattern matching, and some access-control structures, like the Aura's **says** modality. While all terms are members of a single syntactic class, we

will use the metavariables p, P, e, T, and a to indicate places where proofs, propositions, computational expressions, computational types, and principals are expected. To enable the type-and-effect analysis described above, abstractions and arrows are labeled with *worlds* that summarize latent uses of private keys.

Syntactically, the set of worlds is the set of terms augmented with distinguished top and bottom elements. AuraConf's static semantics identify only some worlds as well formed: namely principal constants, variables of type **prin**, $\top$, and $\bot$. We define a partial order on worlds,

$$\frac{}{\bot \sqsubseteq W} \qquad \frac{}{W \sqsubseteq W} \qquad \frac{}{W \sqsubseteq \bot}$$

and can visualize the lattice of well-formed worlds as follows.



Intuitively $W \sqsubseteq U$ when U describes more private keys than W. World $\top$ represents the set of all private keys. Generalizing worlds to arbitrary principal sets would work formally, but is less appealing from an implementation perspective.

The special term **fail** p represents fatal exceptions caused by decryption failures. Argument p represents a proof to be blamed for the exception.

Static Semantics AuraConf's static semantics is based on Aura's, but with several substantial changes.

The static semantics are defined terms of several auxiliary judgments:

Term e_1 equals e_2 in E	*converts* $E\ e_1\ e_2$
e is a value	*value* e
W is a world constant	*simple* W
T has simple inhabitants	*atomic* $\Sigma\, T$
Approximate typing	$E \mathrel{|\!\sim} e : T$

The value relation holds for free variables, computations suspended by $\mathbf{return_f}$ or $\mathbf{bind_f}$, and usual values like lambda abstractions and constants. Judgment *atomic* $\Sigma\, T$ is intended to hold when T is **prin** or T is an inductive type that defines an enumeration, such as bool. The other auxiliary relations are discussed later in this section.

AuraConf typechecks programs using the following main judgments:

Well-formed/typed...	
...signature	$\Sigma \vdash \diamond$
...type environment	$\Sigma; \mathcal{F}; W \mid \vdash E$
...world (V)	$\Sigma; \mathcal{F}; W \mid E; V \vdash \diamond$
...worlds (V and U)	$\Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond$
...term	$\Sigma; \mathcal{F}; W \mid E; V; U \vdash t : s$
...match branches	$\Sigma; \mathcal{F}; W \mid E; V; U; s; \mathit{args} \vdash \mathit{brns} : t$

The typing relation (well-typed term) is complex. How do we read this judgment?

Facts, worlds, and the typing judgment Meta-variable $\mathcal{F}$ is a fact context as described in Section 2. It's formally defined by the grammar

Fact Contexts
$$\mathcal{F} \quad ::= \quad \cdot \mid \mathcal{F}, \mathcal{E}(a, e, n) : t.$$

Intuitively, typechecking uses the fact context to associate typing information with newly created ciphertexts. This is important, because ciphertexts are not generally amenable to inspection. Intuitively, F grows during evaluations and allows the runtime typechecker to to take advantage of information not statically available.

World W, the *statically available key*, describes which key is available for use by the typechecker. We will only consider singleton and bottom worlds here; typechecking a program with $W = \top$ corresponds to having all private keys available at once—a well-defined but unlikely scenario. Together the fact context and statically available key determine a hard limit on the typechecker's ability to reason about ciphertext.

World V, the *soft decryption limit*, is a formal upper limit describing which decryption keys or facts should be used when typechecking a particular term. Intuitively the keys used to check a term are $W \sqcap V$. The soft decryption limit is necessary to deal with mobile code. Consider what happens when Alice creates a string **for** Bob. She is building an object containing a subterm, say s, that Bob must decrypt and typecheck as string. However, Bob must check s without the benefit of Alice's private key and using a different fact context. (Because s might be a computation containing nested binds Bob's task is non-trivial.) To account for this, Alice's typing derivation uses $V =$ Bob when checking s, thus ensuring Bob can understand s without Alice's private information or state. Typechecking a top-level program takes places with $V = \top$, indicating no restriction on key or fact use.

The interaction between fact context, statically available key and soft decryption limit can be better understood by examining simplified versions of typing rules for true casts and return. (Unabridged versions of these rules may be found in figure 4; all rules including those elided here are given in author's thesis [37].) WF-TM-FORRET has form

$$\frac{\_; \mathcal{F}; W \mid \_; a; \_ \vdash e : t \qquad a \sqsubseteq V \qquad \cdots}{\_; \mathcal{F}; W \mid \_; V; \_ \vdash \mathbf{return_f}\ e\ \mathbf{as}\ (t\ \mathbf{for}\ a) : t\ \mathbf{for}\ a}\ .$$

This rule is packaging expression e for consumption by principal a. The first premise checks that e is classified by t under soft decryption limit a—this will ensure that the derivation will work even when a does not have access to the facts in $\mathcal{F}$ or a private key indicated by W. Having checked e under this restriction it's ok to conclude that $\mathbf{return_f}\ e\ \mathbf{as}\ (t\ \mathbf{for}\ a)$ has type $t\ \mathbf{for}\ a$ in a less restricted context, where soft decryption limit V is greater than a.

Observe that elements right of the vertical bar differed between WF-TM-FORRET's premise and conclusion, but symbols on the bar's left stayed the same. In general, symbols left of the bar are parameters of the relation and are held constant throughout an entire derivation tree. Symbols right of the bar are indices and may change within a derivation.

The statically available key is used directly in rule WF-TM-CASTDEC.

$$\frac{b \sqsubseteq W \qquad b \sqsubseteq V \qquad \_; \mathcal{F}; W \mid \_; V; \_ \vdash e : t}{\_; \mathcal{F}; W \mid \_; V; \_ \vdash \mathbf{cast}\ \mathcal{E}(b, e, n)\ \mathbf{to}\ (t\ \mathbf{for}\ b) : t\ \mathbf{for}\ b}$$

Here an annotated ciphertext encrypted for b is type checked by decrypting and recursively typechecking its contents. Premise $b \sqsubseteq W$ checks that the statically available key is sufficient to perform the decryption. WF-TM-CASTDEC may only be applied when current soft decryption limit V is greater than b. This last point is important. Together with setting the soft decryption limit to a in WF-TM-FORRET, it ensures that impossible decryptions are not required to later typecheck data packaged by correct programs.

Finally, WF-TM-CASTFACT has form

$$\frac{(\mathcal{E}(a, e, n) : t\ \mathbf{for}\ b) \in F \qquad b \sqsubseteq V}{\_; \mathcal{F}; W \mid \_; V; \_ \vdash \mathbf{cast}\ \mathcal{E}(a, e, n)\ \mathbf{to}\ (t\ \mathbf{for}\ b) : t\ \mathbf{for}\ b}\ .$$

This says that an annotated piece of ciphertext can have type $t\ \mathbf{for}\ b$ when a fact indicates the type. As above, soft decryption limit V must be greater than b. Critically, it's not necessary that $b \sqsubseteq W$—

$\boxed{\Sigma; \mathcal{F}; W|E; V; U \vdash e : t}$

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textsf{Type} : \textsf{Kind}} \text{ WF-TM-TYPE}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; \bot; \bot \vdash u_1 : k_1 \qquad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \qquad \Sigma; \mathcal{F}; W|E; V; U_0 \vdash \diamond \\ \Sigma; \mathcal{F}; W|E, x : u_1 \textsf{ at } \bot; \bot; \bot \vdash u_2 : k_2 \qquad k_2 \in \{\textsf{Type}, \textsf{Prop}, \textsf{Kind}\} \qquad k_1 \in \{\textsf{Type}, \textsf{Prop}\} \vee u_1 \in \{\textsf{Type}, \textsf{Prop}\}\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash (x{:}u_1) \to_{\{U_0\}} u_2 : k_2} \text{ WF-TM-ARR}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \qquad (x : t \textsf{ at } V_0) \in E \qquad V_0 \sqsubseteq V}{\Sigma; \mathcal{F}; W|E; V; U \vdash x : t} \text{ WF-TM-VAR}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; \bot; \bot \vdash u_1 : k_1 \qquad \Sigma; \mathcal{F}; W|E; V; U_0 \vdash \diamond \qquad \Sigma; \mathcal{F}; W|E, x : u_1 \textsf{ at } \bot; V; U_0 \vdash f : u_2 \\ \Sigma; \mathcal{F}; W|E; V; U \vdash (x{:}u_1) \to_{\{U_0\}} u_2 : k \qquad k \in \{\textsf{Type}, \textsf{Prop}\} \qquad k_1 \in \{\textsf{Type}, \textsf{Prop}\} \vee u_1 \in \{\textsf{Type}, \textsf{Prop}\}\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \lambda_{\{U_0\}} x{:}u_1.\ f : (x{:}u_1) \to_{\{U_0\}} u_2} \text{ WF-TM-ABS}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; V; U \vdash e_1 : (x{:}t_2) \to_{\{U_0\}} u \\ \Sigma; \mathcal{F}; W|E; \bot; U_2 \vdash e_2 : t_2 \qquad \Sigma; \mathcal{F}; W|E; V; U \vdash t_2 : k_2 \qquad \Sigma; \mathcal{F}; W|E; V; U \vdash \{e_2/x\}u : k_u \\ U_0 \sqsubseteq U \qquad \left(\begin{array}{c}(\textit{value } e_2 \wedge U_2 = \bot) \vee (k_u = \textsf{Type} \wedge x \notin \mathit{fv}(u) \wedge U_2 = U) \\ \vee\ (k_2 \in \{\textsf{Prop}, \textsf{Kind}\} \wedge x \notin \mathit{fv}(u) \wedge U_2 = U)\end{array}\right)\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash e_1\ e_2 : \{x/e_2\}u} \text{ WF-TM-APP}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; V_1; U \vdash e_1 : t_1 \qquad \Sigma; \mathcal{F}; W|E; V; V_1 \vdash \diamond \\ \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t_1 : k \qquad \Sigma; \mathcal{F}; W|E, x : t_1 \textsf{ at } V_1; V; U \vdash e_2 : t \qquad k \in \{\textsf{Type}, \textsf{Prop}\} \qquad V_1 \sqsubseteq V\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textsf{let } x \textsf{ at } V_1 = e_1 \textsf{ in } e_2 : t} \text{ WF-TM-LETAT}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; b; U \vdash e_1 : t_1 \textsf{ for } b \qquad \Sigma; \mathcal{F}; W|E; b \vdash \diamond \qquad \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t_1 : \textsf{Type} \\ \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t \textsf{ for } b : \textsf{Type} \qquad \Sigma; \mathcal{F}; W|E; V \vdash \diamond \qquad \Sigma; \mathcal{F}; W|E, x : t_1 \textsf{ at } b; b; b \vdash e_2 : t \textsf{ for } b \qquad b \sqsubseteq V\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textsf{bind}_\textsf{f}\ x = e_1 \textsf{ in } e_2 \textsf{ as } (t \textsf{ for } b) : t \textsf{ for } b} \text{ WF-TM-FORBIND}$$

$$\frac{\textit{value } a \qquad \Sigma; \mathcal{F}; W|\cdot; \bot; \bot \vdash a : \textsf{prin} \qquad \Sigma; \mathcal{F}; W|\cdot; \bot; \bot \vdash P : \textsf{Prop} \qquad \Sigma; \mathcal{F}; W|E; V; U \vdash P : \textsf{Prop}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textsf{sign}(a, P) : a \textsf{ says } P} \text{ WF-TM-SIGN}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textsf{prin} : \textsf{Type}} \text{ WF-TM-PRIN}$$

Figure 4: Selected typing rules for AuraConf

this rule enables typing checking of ciphertexts without decryption and without principal b's private key.

AuraConf typing contexts track a soft decryption limit for each bound variable. This is necessary to ensure that a substitution property—replacing variables with appropriate values maintains a term's type—holds. Formally, AuraConf environments are defined by the following grammar.

Environments
$E \quad ::= \quad \cdot \mid E, x{:}t \textsf{ at } W \mid E, x \sim (t_1 = t_2){:}u \textsf{ at } W$

Equalities in the environment enable type refinement as in core Aura [20].

The typing relation's final new metavariable, U, is the judgment's *effect label*. This summarizes the keys that are necessary to successfully execute a piece of code. Effect label $U = \bot$ indicates that an expression is *pure*—that it can execute with no private keys. For instance, **say** signs a proposition. It's typed as follows:

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; \bot; \bot \vdash P : \textsf{Prop} \\ \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash a : \textsf{prin} \qquad a \sqsubseteq U \qquad \cdots\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textsf{say}\ a\ P : \textsf{pf } a \textsf{ says } P}$$

Premise $a \sqsubseteq U$ records that **say** uses a's key. Additionally AuraConf typing maintains the invariant that type-level terms, such as a **says** P, are pure. Checking the rule's premises with bottom effect label helps to enforce this condition. For a comparison, consider the rule

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; \bot; \bot \vdash P : \textsf{Prop} \\ \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash a : \textsf{prin} \qquad \cdots\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textsf{sign}(a, P) : \textsf{pf } a \textsf{ says } P}\ ,$$

which types previously created signatures. Because no new signatures are built by **sign**, typing does not require $a \sqsubseteq U$, or otherwise constrain effect label U.

It's important to understand the distinction between a judgment's soft decryption limit and effect label. The soft decryption limit controls access to a private key used *statically* for type checking. In contrast, the effect label describes keys used *dynamically* for decryption and signing. It's appealing to attempt to conflate these, but my attempts to do so were imprecise, inelegant, or plain incorrect. The difficulties arise from several considerations. Consider the application $f(\lambda x.e)$ where f does not apply $\lambda x.e$. (Function f

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{bits} : \mathbf{Type}} \text{ WF-TM-BITS}$$

$$\frac{\Sigma; \mathcal{F}; W|E; \bot; \bot \vdash P : \mathbf{Prop} \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash a : \mathbf{prin} \quad a \sqsubseteq U \quad \mathit{value}\ a}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{say}\ a\ P : \mathbf{pf}\ (a\ \mathbf{says}\ P)} \text{ WF-TM-SAY}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V; \bot \vdash t : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; V; \bot \vdash a : \mathbf{prin} \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \mathit{value}\ t \quad \mathit{value}\ a}{\Sigma; \mathcal{F}; W|E; V; U \vdash t\ \mathbf{for}\ a : \mathbf{Type}} \text{ WF-TM-FOR}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathcal{E}(t_1, t_2, n) : \mathbf{bits}} \text{ WF-TM-ENC} \qquad \frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad E \vdash\!\sim e : t\ \mathbf{for}\ a}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{asbits}\ e : \mathbf{bits}} \text{ WF-TM-ASBITS}$$

$$\frac{\Sigma; \mathcal{F}; W|E; a; a \vdash e : t \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t\ \mathbf{for}\ a : \mathbf{Type} \quad a \sqsubseteq V}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{return_f}\ e\ \mathbf{as}\ (t\ \mathbf{for}\ a) : t\ \mathbf{for}\ a} \text{ WF-TM-FORRET}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash t\ \mathbf{for}\ b : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; V; U \vdash e : \mathbf{bits} \quad \mathit{value}\ e}{\Sigma; \mathcal{F}; W|E; V; U \vdash e\ \mathbf{isa}\ t\ \mathbf{for}\ b : \mathbf{Prop}} \text{ WF-TM-ISA}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash e : t\ \mathbf{for}\ a \quad a \sqsubseteq V \quad a \sqsubseteq U}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{run_f}\ e : t} \text{ WF-TM-FORRUN}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash e : t_1 \quad \Sigma; \mathcal{F}; W|E; V; U \vdash t_2 : \mathbf{Type} \quad \mathit{converts}\ E\ t_1\ t_2}{\Sigma; \mathcal{F}; W|E; V; U \vdash \langle e : t_2 \rangle : t_2} \text{ WF-TM-CASTCONV}$$

$$\frac{(\mathcal{E}(a, e, n) : t\ \mathbf{for}\ b) \in \mathcal{F} \quad \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t\ \mathbf{for}\ b : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad b \sqsubseteq V}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{cast}\ \mathcal{E}(a, e, n)\ \mathbf{to}\ (t\ \mathbf{for}\ b) : t\ \mathbf{for}\ b} \text{ WF-TM-CASTFACT}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|\cdot; b; b \vdash e : t \\ \Sigma; \mathcal{F}; W|\cdot; \bot; \bot \vdash t\ \mathbf{for}\ b : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \Sigma; \mathcal{F}; W|\cdot; b \vdash \diamond \quad b \sqsubseteq V \quad b \sqsubseteq W\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{cast}\ \mathcal{E}(b, e, n)\ \mathbf{to}\ (t\ \mathbf{for}\ b) : t\ \mathbf{for}\ b} \text{ WF-TM-CASTDEC}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E; V; U \vdash p : \mathbf{pf}\ (a\ \mathbf{says}\ (e\ \mathbf{isa}\ t\ \mathbf{for}\ b)) \\ \Sigma; \mathcal{F}; W|E; V; U \vdash e : \mathbf{bits} \quad \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t\ \mathbf{for}\ b : \mathbf{Type} \quad \mathit{value}\ e\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{cast}\ e\ \mathbf{to}\ (t\ \mathbf{for}\ b)\ \mathbf{blaming}\ p : t\ \mathbf{for}\ b} \text{ WF-TM-CASTJUST}$$

Fig. 4: Selected typing rules for AuraConf (cont.)

may, however, do other interesting things with $\lambda x.e$, such as store it in a data structure.) We want the type system to require a sufficient soft decryption limit to analyze e's embedded ciphertexts. In contrast, e's latent effects are not forced and we would like the application to check with $\bot$ effect label. It's unclear how a single annotation can accommodate both views; using a separate soft decryption limit and effect label resolves this. More generally, the type system treats soft decryption limits like Jia and Walker's [21] **at** modality, while the effect labels are inspired by standard type-and-effect systems. Technically, these analyses are quite different and it's unsurprising that to reap the benefits of both requires incorporating mechanisms inspired by each.

Signatures, worlds, environments, branches, and conversion. The judgments for type signatures and branches follow core Aura and are not reproduced here. Every branch of a pattern match must share the same effect label. Types declared in signatures must be pure and check with $W = V = \bot$ and $\mathcal{F} = \cdot$.

Definitions of environment, world, and worlds well-formedness are more novel and are detailed in Figure 5.

The well-formed environment relation checks that all world annotations are themselves well-formed. Additionally, type-level variables (i.e., those classified by **Type** or **Prop**) may only be annotated with world $\bot$. The well-formed environment relation also ensures that the statically available key is a *simple world*—either a principal constant or $\bot$. Intuitively this ensures statically available keys may be interpreted as key constants.

The well-formed world relation always accepts $\top$ and $\bot$. If the world wraps a term, it must be value of type **prin**. The well-formed worlds relation checks that two worlds, typically a soft decryption limit and effect label, are well-formed.

Finally, *converts* $E\ e_1\ e_2$ holds when e_1 and e_2 are equal according to constraints in environment E. This relation is of course reflexive, symmetric, and transitive; the key rule is

$$\frac{x{\sim}(s = t){:}k \in E}{\mathit{converts}\ E\ s\ t}\ ,$$

which uses equality assumptions in the environment. Such equalities are introduced by a conditional operator,

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W|E, x{\sim}(v_1 = v_2){:}k; V; U \vdash e_1 : t \\ \Sigma; \mathcal{F}; W|E; V; U \vdash e_2 : t \qquad \ldots\end{array}}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{if}\ v_1 = v_2\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t}\ .$$

$\boxed{\mathit{simple}\ W}$

$$\frac{}{\mathit{simple}\ \bot} \qquad \frac{a \in \{\mathsf{A}, \mathsf{B}, \mathsf{C} \ldots\}}{\mathit{simple}\ a}$$

$\boxed{\Sigma; \mathcal{F}; W| \vdash E}$

$$\frac{\mathit{simple}\ W}{\Sigma; \mathcal{F}; W| \vdash \cdot}$$

$$\frac{\Sigma; \mathcal{F}; W| \vdash E \qquad \Sigma; \mathcal{F}; W|E; V \vdash \diamond \qquad \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t : k \qquad x \text{ fresh} \qquad (k \in \{\mathsf{Type}, \mathsf{Prop}\}) \vee (t \in \{\mathsf{Type}, \mathsf{Prop}\} \wedge V = \bot)}{\Sigma; \mathcal{F}; W| \vdash E, x : t\ \mathsf{at}\ V}$$

$$\frac{\Sigma; \mathcal{F}; W| \vdash E \qquad \Sigma; \mathcal{F}; W|E; \bot; U \vdash e_1 : t \qquad \Sigma; \mathcal{F}; W|E; \bot; U \vdash e_2 : t \qquad \mathit{atomic}\ \Sigma\ t \qquad x \text{ fresh} \qquad \mathit{value}\ e_1 \qquad \mathit{value}\ e_2 \qquad \Sigma; \mathcal{F}; W|E; \bot; \bot \vdash t : \mathsf{Type} \qquad \Sigma; \mathcal{F}; W|E; V \vdash \diamond}{\Sigma; \mathcal{F}; W| \vdash E, x \sim (e_1 = e_2) : t\ \mathsf{at}\ V}$$

$\boxed{\Sigma; \mathcal{F}; W|E; V \vdash \diamond}$

$$\frac{\Sigma; \mathcal{F}; W| \vdash E}{\Sigma; \mathcal{F}; W|E; \bot \vdash \diamond} \qquad \frac{\Sigma; \mathcal{F}; W|E; \bot; \bot \vdash a : \mathsf{prin} \qquad \mathit{value}\ a}{\Sigma; \mathcal{F}; W|E; a \vdash \diamond}$$

$$\frac{\Sigma; \mathcal{F}; W| \vdash E}{\Sigma; \mathcal{F}; W|E; \top \vdash \diamond}$$

$\boxed{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond}$

$$\frac{\Sigma; \mathcal{F}; W|E; V \vdash \diamond \qquad \Sigma; \mathcal{F}; W|E; U \vdash \diamond}{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond}$$

Figure 5: Major auxiliary judgments for AuraConf's static semantics. The predicate *atomic* $\Sigma\, t$, used here but not defined, indicates that t is an inductive type whose elements can easily be tested for equality [37].

(The Coq definition of *converts* includes an extra argument to allow for later language extensions; this is elided.) Conversion also includes congruence rules. For instance, under assumption $x = \mathsf{self}$, term $x\ \mathsf{says}\ P$ converts to $\mathsf{self}\ \mathsf{says}\ P$. Equalities only mention atomic values, and conversion only alters the "value" parts of a type—convertible types always have the same shape up to embedded data values. Many standard presentations of dependently typed languages use implicit conversions, which may occur anywhere in a type derivation, but Aura requires an explicit source-code cast. This is appealing because it yields an algorithmic type system.

New and modified language constructs Moving from Aura to AuraConf requires broad changes to the static semantics. Here we will examine the most interesting aspects of the new static semantics, using simplified typing rules.

Variables and binding with soft decryption limits Application, abstraction, and variable expressions are changed when moving from Aura to AuraConf. This is necessary to work with soft decryption limits and effect labels.

The variable rule is

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \qquad (x : t\ \mathsf{at}\ V_0) \in E \qquad V_0 \sqsubseteq V}{\Sigma; \mathcal{F}; W|E; V; U \vdash x : t}\ .$$

From an AuraConf perspective the important part is the premise $V_0 \sqsubseteq V$. Elsewhere, we ensure that whenever some value v is substituted for x that value is well typed with soft decryption limit V_0.

A function's type is annotated with its body's suspended effects. The typing rule looks like

$$\frac{\Sigma; \mathcal{F}; W|E; V; U_0 \vdash \diamond \qquad \Sigma; \mathcal{F}; W|E, x : u_1\ \mathsf{at}\ \bot; V; U_0 \vdash f : u_2 \qquad \cdots}{\Sigma; \mathcal{F}; W|E; V; U \vdash \lambda_{\{U_0\}} x{:}u_1.\ f : (x{:}u_1) \to_{\{U_0\}} u_2}\ .$$

The rule could be generalized by allowing latent effect label U_0 to depend on x. This was omitted in the interest of simplicity. Dependent effects can still be written; they must reference variables quantified at a surrounding abstraction. To avoid annotating every abstraction with a soft decryption limit, this rule binds x at bottom.

Lambda abstractions are used at applications. The essence of application typing is as follows.

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash e_1 : (x{:}t_2) \to_{\{U_0\}} u \qquad \Sigma; \mathcal{F}; W|E; \bot; U_2 \vdash e_2 : t_2 \qquad (\mathit{value}\ e_2 \wedge U_2 = \bot) \vee (x \notin \mathit{fv}(u) \wedge U_2 = U) \qquad U_0 \sqsubseteq U \qquad \cdots}{\Sigma; \mathcal{F}; W|E; V; U \vdash e_1\ e_2 : \{x/e_2\}u}$$

Application ensures that argument e_2 is typeable with bottom soft decryption limit; this matches with abstraction typing. Because evaluating the abstraction may trigger latent effect U_0, we require $U_0 \sqsubseteq U$. When e_2 is not a value—which implies e_1's type is not dependent—e_2 may also have have an effect label up to U.

So far we've only seen a way to introduce variables $\mathsf{at}\ \bot$. The $\mathsf{let\ at}$ construct allows us to reason about variables with different soft decryption limits. This construct's typing rule is summarized by

$$\frac{\Sigma; \mathcal{F}; W|E; V_1; U \vdash e_1 : t_1 \qquad \Sigma; \mathcal{F}; W|E, x : t_1\ \mathsf{at}\ V_1; V; U \vdash e_2 : t \qquad V_1 \sqsubseteq V \qquad \cdots}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathsf{let}\ x\ \mathsf{at}\ V_1 = e_1\ \mathsf{in}\ e_2 : t}\ .$$

Here e_1 is checked with soft decryption limit V_1 and is bound to x in e_2. In e_2's environment, x is typed $\mathsf{at}\ V_1$. The restriction $V_1 \sqsubseteq V$ is necessary to prevent $\mathsf{let\ at}$s from raising the soft decryption limit and allowing the unsafe use of facts or statically available keys. While $\mathsf{let\ at}$ could be defined as a derived form, based on an enhanced abstraction form, the independent construct simplifies function definition and breaks the language into simple, orthogonal pieces.

The ciphertext and the for monad The AuraConf type system always interprets unannotated ciphertexts as unintelligible blobs.

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathcal{E}(t_1, t_2, n) : \mathsf{bits}}$$

As discussed above, more precise typings may be given to ciphertexts annotated with true casts or justified casts.

The main operators for working with confidential values are return, run, and bind. The return operator packages an expression as a confidential computation and is typed as follows.

$$\frac{\Sigma; \mathcal{F}; W|E; a; a \vdash e : t \qquad a \sqsubseteq V \qquad \cdots}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathsf{return_f}\ e\ \mathsf{as}\ (t\ \mathsf{for}\ a) : t\ \mathsf{for}\ a}\ .$$

Because e will eventually be run with a's authority it is type checked with soft decryption limit and effect label a. Typically $W \not\sqsubseteq a$ so setting the soft decryption limit to a prevents statically available key W from being used when checking e—important because W will not be on hand when a's program needs to check e. Likewise effect label a rules out inappropriate occurrences of say or $\mathsf{run_f}$. Typing for $\mathsf{bind_f}$ works analogously; see rule WF-TM-FORBIND.

$\boxed{E \vdash\sim e : t}$

$$\frac{(x : t \textbf{ at } V) \in E}{E \vdash\sim x : t} \text{ GE-TM-VAR}$$

$$\frac{}{E \vdash\sim \mathcal{E}(a, e, n) : \textbf{bits}} \text{ GE-TM-ENC}$$

$$\frac{\textit{value} \textbf{ cast } e \textbf{ to } t \textbf{ for } a \,[\, \textbf{blaming } p \,] \qquad E \vdash\sim e : \textbf{bits} \qquad \forall x \in \text{vars}(p, t \textbf{ for } a). \exists t_x, V_x. (x : t_x \textbf{ at } V_x) \in E}{E \vdash\sim \textbf{cast } e \textbf{ to } t \textbf{ for } a \,[\, \textbf{blaming } p \,] : t \textbf{ for } a} \text{ GE-TM-CAST}$$

Figure 6: Approximate typing judgment used by WF-TM-ASBITS

The $\textbf{run}_\textbf{f}$ operator decrypts and evaluates annotated ciphertexts. It's typed by:

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash e : t \textbf{ for } a \qquad a \sqsubseteq V \qquad a \sqsubseteq U}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textbf{run}_\textbf{f} \; e : t}$$

The premise $a \sqsubseteq U$ forces effect label U to record that the $\textbf{run}_\textbf{f}$ uses a's private key. Premise $a \sqsubseteq V$ prevents problems with nested occurrences of $\textbf{run}_\textbf{f}$. For example when a evaluates $\textbf{run}_\textbf{f}\ (\textbf{run}_\textbf{f}\ e_1)$ to $\textbf{run}_\textbf{f}\ e_2$, term e_2 might be contain true-casts for a. Hence V must be greater than a to ensure preservation.

Finally, **asbits** transforms an annotated ciphertext with **for** type into a bare ciphertext with type **bits**. This operator is typed as follows.

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \qquad E \vdash\sim e : t \textbf{ for } a}{\Sigma; \mathcal{F}; W|E; V; U \vdash \textbf{asbits } e : \textbf{bits}}$$

The first premise maintains the invariant that the typing judgment's subjects are well-formed. The second premise uses a liberal over-approximation of typing to check that e is almost a t **for** a. The approximation, formalized in Figure 6, types variables and bare encryptions as usual, but always trusts the annotation on true or justified casts. (The square-bracket notation in GE-TM-CAST defines a rule that works for both flavors of cast.) It's sound to use the approximation here because **asbits** dynamically discards casts, returning the underlying ciphertexts; **asbits** launders bad **for**s into good **bits**. The typing rule is desirable because the typing of **asbits** e is independent of the facts context and statically available key, a useful property for defining mobile code.

Dynamic semantics The dynamic semantics for AuraConf makes precise the notion of a program's authority, realistically models the state necessary to perform (pseudo-)randomized cryptography, and enables reasoning about dynamically created ciphertexts.

The evaluation judgment is written

$$\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \;\mapsto\; \{\!|e', n'|\!\} \text{ learning } \mathcal{F}.$$

This says that an expression e running with W's authority—with the private keys described by world W—steps to e'. Expression e may, as described below, dynamically invoke the type checker, so the evaluation relation contains a signature Σ and fact context $\mathcal{F}_0$ for this purpose. Natural number n represents the initial seed of a randomization vector for encryption; the step updates it to n'.[1] Finally $\mathcal{F}$ is a fact context, with zero or one elements, containing facts about freshly created ciphertexts.

[1] Note that literally using a stream of sequential numbers as inputs to the encryption algorithm may not be secure for some protocols. Instead we should view such uses of n as actually looking up the nth number in a sequence of (pseudo-)random numbers.

In general AuraConf's evaluation relation subsumes Aura's. For intuition, when $e \mapsto e'$ in Aura,

$$\Sigma; \mathcal{F}_0; \textbf{self} \vdash \{\!|e, n|\!\} \;\mapsto\; \{\!|e', n|\!\} \text{ learning } \cdot$$

holds in AuraConf. Figure 7 lists the evaluation rules for new operators.

Rules STEP-FORRET and STEP-FORBIND introduce new ciphertexts. In each case the current randomization seed, n is inserted into the ciphertext and the seed is incremented. Additionally a fact describing the ciphertext is learned. While STEP-FORRET is simple, STEP-FORBIND looks more complicated. The latter builds an expression using **let at** that can be run by the destination machine and that performs necessary decryptions.

Rule STEP-FORRUN-OK, STEP-FORRUN-ILLTYPED, and STEP-FORRUN-JUNK attempt to decrypt and typecheck an annotated ciphertext, signaling an error as needed.

Figure 7 elides several congruence rules. They are all similar to STEP-APP-CONGL, which copies its premise's new facts and randomization seed.

Basic metatheory and soundness AuraConf satisfies two important properties: syntactic soundness and noninterference. Syntactic soundness guarantees that all well-typed programs have a well-defined evaluation semantics. Noninterference [4, 40], states that a program's outputs are not affected (up to a natural equivalence induced by cryptography) by inputs intended to be secret. AuraConf's type system has several non-standard aspects; consequently, the technical statements and proofs of these properties are novel.

All properties of AuraConf are formalized as constructive proofs in the Coq proof assistant.[2] Such formalization is particularly important for large languages and security-focused languages; AuraConf is both.

Stating preservation and progress requires defining when a term has reached an exceptional state. This is intended to occur only after a decryption failure, and identifies a proof to be used when diagnosing the failure. We write e blames p when $(\textbf{fail } p)$ is a subterm of e, not located under a $\textbf{return}_\textbf{f}$ or a $\textbf{bind}_\textbf{f}$. In Coq this is defined as an inductive predicate over the syntax of terms.

Preservation states that if a well-typed term steps the result has the same type, or else a decryption error has been detected and a proof identified for blame assignment.

LEMMA 1 (Preservation). *Assume* $\Sigma; \mathcal{F}_0; W|E; V; U \vdash e : t$ *and* $\Sigma \vdash \diamond$. *Then* $\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \;\mapsto\; \{\!|e', n'|\!\}$ *learning* $\mathcal{F}$ *implies either* $\Sigma; \mathcal{F}_0 +\!\!+ \mathcal{F}; W|E; V; U \vdash e' : t$ *or there exists* p *such that* e' blames p.

Notation $\mathcal{F}_0 +\!\!+ \mathcal{F}$ denotes a fact context containing the elements of $\mathcal{F}$ and $\mathcal{F}_0$.

The above lemma misses an important aspect of evaluation. Running an AuraConf program doesn't simply reduce an input term to a result; it also generates a sequence of new facts. There are terms that typecheck under bad fact contexts, but get stuck at evaluation. Thus we must ensure that newly generated facts are, in the following sense, semantically valid.

DEFINITION 1 ($\text{valid}_\Sigma\, \mathcal{F}$). *We write* $\text{valid}_\Sigma\, \mathcal{F}$ *when both the following hold. First,* $\Sigma \vdash \diamond$. *Second, for every* $\mathcal{E}(a, e, n) : t \textbf{ for } b$ *in* $\mathcal{F}$ *it is the case that* $a = b$ *and* $\Sigma; \cdot; b|\cdot; b; b \vdash e : t$.

Intuitively this predicate holds when decrypting each ciphertext in a fact context would validate the declared types. Certain bogus facts, say `"hello"` : **int**, aren't harmful to soundness, and are ignored. The empty fact context is trivially valid.

[2] Coq scripts are available from the author's webpage, `http://www.cs.ucla.edu/~jeff/`.

$\boxed{\Sigma; F_0; W \vdash \{\!|e_1, n_1|\!\} \mapsto \{\!|e_2, n_2|\!\} \text{ learning } F}$

$$\frac{\textit{value } v}{\Sigma; F_0; W \vdash \{\!|\textsf{let } x \textsf{ at } V = v \textsf{ in } e, n|\!\} \mapsto \{\!|\{v/x\}e, n|\!\} \text{ learning } \cdot}\ \textsc{step-LetAt}$$

$$\frac{}{\Sigma; F_0; W \vdash \{\!|\textsf{return}_\textsf{f}\ e \textsf{ as } (t \textsf{ for } a), n|\!\} \mapsto \{\!|\textsf{cast } \mathcal{E}(a, e, n) \textsf{ to } (t \textsf{ for } a), n+1|\!\} \text{ learning } \mathcal{E}(a, e, n) : t \textsf{ for } a}\ \textsc{step-ForRet}$$

$$\frac{\textit{value } v}{\begin{array}{l}\Sigma; F_0; W \vdash \{\!|\textsf{bind}_\textsf{f}\ x = v \textsf{ in } e \textsf{ as } t \textsf{ for } a, n|\!\} \\ \quad \mapsto \{\!|\textsf{cast } \mathcal{E}(a, \textsf{let } x \textsf{ at } a = (\textsf{run}_\textsf{f}\ v) \textsf{ in } (\textsf{run}_\textsf{f}\ e), n) \textsf{ to } (t \textsf{ for } a), n+1|\!\} \\ \quad \text{learning } \mathcal{E}(a, \textsf{let } x \textsf{ at } a = (\textsf{run}_\textsf{f}\ v) \textsf{ in } (\textsf{run}_\textsf{f}\ e), n) : t \textsf{ for } a\end{array}}\ \textsc{step-ForBind}$$

$$\frac{\textit{value}(\textsf{cast } \mathcal{E}(a, e, m) \textsf{ to } t\ [\ \textsf{blaming } p\ ])}{\Sigma; F_0; W \vdash \{\!|\textsf{asbits cast } \mathcal{E}(a, e, m) \textsf{ to } t\ [\ \textsf{blaming } p\ ], n|\!\} \mapsto \{\!|\mathcal{E}(a, e, m), n|\!\} \text{ learning } \cdot}\ \textsc{step-AsBits}$$

$$\frac{\textit{value}(\textsf{cast } \mathcal{E}(a, e, m) \textsf{ to } t\ [\ \textsf{blaming } p\ ]) \qquad \Sigma; F_0; W|\cdot; a; a \vdash e : t \qquad a \sqsubseteq W}{\Sigma; F_0; W \vdash \{\!|\textsf{run}_\textsf{f}\ (\textsf{cast } \mathcal{E}(a, e, m) \textsf{ to } t\ [\ \textsf{blaming } p\ ]), n|\!\} \mapsto \{\!|e, n|\!\} \text{ learning } \cdot}\ \textsc{step-ForRun-Ok}$$

$$\frac{\textit{value}(\textsf{cast } \mathcal{E}(a, e, m) \textsf{ to } t \textsf{ blaming } p) \qquad \Sigma; F_0; W|\cdot; a; a \not\vdash e : t \qquad a \sqsubseteq W}{\Sigma; F_0; W \vdash \{\!|\textsf{run}_\textsf{f}\ (\textsf{cast } \mathcal{E}(a, e, m) \textsf{ to } t \textsf{ blaming } p), n|\!\} \mapsto \{\!|\textsf{fail } p, n|\!\} \text{ learning } \cdot}\ \textsc{step-ForRun-IllTyped}$$

$$\frac{\textit{value}(\textsf{cast } \mathcal{E}(a, e, m) \textsf{ to } (t \textsf{ for } b) \textsf{ blaming } p) \qquad b \sqsubseteq W \qquad a \neq b}{\Sigma; F_0; W \vdash \{\!|\textsf{run}_\textsf{f}\ (\textsf{cast } \mathcal{E}(a, e, m) \textsf{ to } (t \textsf{ for } b) \textsf{ blaming } p), n|\!\} \mapsto \{\!|\textsf{fail } p, n|\!\} \text{ learning } \cdot}\ \textsc{step-ForRun-Junk}$$

$$\frac{\Sigma; F_0; W \vdash \{\!|e_1, n|\!\} \mapsto \{\!|e_1', n'|\!\} \text{ learning } F}{\Sigma; F_0; W \vdash \{\!|e_1\ e_2, n|\!\} \mapsto \{\!|e_1'\ e_2, n'|\!\} \text{ learning } F}\ \textsc{step-App-CongL}$$

Figure 7: Selected AuraConf evaluation rules.

Importantly $\text{valid}_\Sigma\ \mathcal{F}$ is not defined as a typing judgment because its truth, in general, may only be ascertained with access to every principal's private key. Such a property is useless when implementing a typechecker. Thus it is better to consider validity as a semantic property existing beside but distinct from AuraConf's type system.

The following lemma shows facts generated during reduction are valid.

LEMMA 2 (New Fact Validity). *Assume that* $\Sigma \vdash \diamond$ *holds and* $\Sigma; \mathcal{F}_0; W|E; V; U \vdash e : t$. *Then* $\text{valid}_\Sigma\ \mathcal{F}_0$ *and* $\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \mapsto \{\!|e', n'|\!\}$ *learning* $\mathcal{F}$ *implies* $\text{valid}_\Sigma\ \mathcal{F}$.

Additionally, AuraConf has a decidable typing relation. Decidability is of independent theoretic interest, but matters in particular because evaluating $\textsf{run}_\textsf{f}$ dynamically invokes the type checker. Were typing undecidable, $\textsf{run}_\textsf{f}$ could instead conservatively approximate; otherwise the progress lemma would not hold.

LEMMA 3 (Decidability). *Suppose* $\Sigma \vdash \diamond$*; then it is decidable if that* $\Sigma; \mathcal{F}; W|E; V; U \vdash e : t$. *Furthermore, it is also decidable if there exists any* S *such that* $\Sigma; \mathcal{F}; W|E; V; U \vdash e : t$.

The AuraConf statement of progress follows. Note that it describes the behavior of terms that are well-typed using a valid fact context. Additionally, any simple world greater than U and V—that is with the private keys specified by the soft decryption limit and effect label—has enough authority to step a program without getting stuck.

LEMMA 4 (Progress). *Assume Conjecture 3 holds. Assume also that* $\Sigma \vdash \diamond$, $\text{valid}_\Sigma\ \mathcal{F}_0$, *and* $\Sigma; \mathcal{F}_0; W_0|E; V; U \vdash e : t$. *Now suppose* W *is a simple world where* $U \sqsubseteq W$ *and* $V \sqsubseteq W$. *Then either* e *is a value, or there exist* e', n', *and* $\mathcal{F}$ *where* $\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \mapsto \{\!|e', n'|\!\}$ *learning* $\mathcal{F}$.

$\boxed{W \vdash t_1 \simeq t_2}$

$$\frac{}{W \vdash x \simeq x}\ \textsc{sim-Var}$$

$$\frac{W \vdash t_{11} \simeq t_{12} \qquad W \vdash t_{21} \simeq t_{22}}{W \vdash (t_{11}\ t_{21}) \simeq (t_{12}\ t_{22})}\ \textsc{sim-App}$$

$$\frac{a \sqsubseteq W \qquad W \vdash e_1 \simeq e_2}{W \vdash \mathcal{E}(a, e_1, n_1) \simeq \mathcal{E}(a, e_2, n_2)}\ \textsc{sim-Decrypt}$$

$$\frac{a \not\sqsubseteq W \qquad b \not\sqsubseteq W}{W \vdash \mathcal{E}(a, e_1, n_1) \simeq \mathcal{E}(b, e_2, n_2)}\ \textsc{sim-Opaque}$$

Figure 8: Selected rules from the definition of similar terms.

Lemmas 1, 2, and 4 together imply that AuraConf is sound.

Noninterference Noninterference properties, which state that a program's secret inputs do not influence its public outputs, are a common way of defining security for programming languages [4, 39, 40]. Such properties are formalized by saying programs which differ only in their secret components are *similar* and showing that similar terms reduce to similar values. The following develops a noninterference property for AuraConf.

AuraConf similarity is defined relative to particular set of keys used to analyze ciphertexts. Figure 8 gives the key rules from the definition of similarity. Most often, two terms are related when they are identical, as in SIM-VAR, or share a top-level constructor with similar subterms, as in SIM-APP. The figure elides a tedious quantity of rules implementing this scheme. Similarity is more interesting for ciphertexts. Rule SIM-DECRYPT finds two ciphertexts similar when they are encrypted with the same key, can be decrypted by W (captured by premise $a \sqsubseteq W$), and have similar payloads. This formalizes the idea that encrypting similar terms should yield similar results. Finally, SIM-OPAQUE states two ciphertexts are similar when neither can be decrypted. This captures the intuition that ciphertexts are black boxes, immune to analysis without a key. We implicitly assume ciphertexts are (randomly) padded such that ciphertext length cannot be used at a side channel. A faithful implementation will require care to properly handle AuraConf's rich data structures.

The following lemma gives AuraConf's noninterference property. It considers running two terms, e_1 and e_2, that step without error under authority W. If the terms are similar at W (or any higher world W_0), the resulting terms, e_1' and e_2', are similar as well. That is, running a program twice with two different confidential inputs yields results that cannot be distinguished without a sufficiently privileged private key.

THEOREM 1 (Noninterference). *Assume that both* $\Sigma \vdash \diamond$ *and* $\Sigma; \mathcal{F}_1; W|\cdot; V; U \vdash e_1 : k_1$. *Pick* W_0 *and* e_2 *where* $W_0 \vdash e_1 \simeq e_2$ *and* $W \sqsubseteq W_0$. *If*

- $\Sigma; W; \mathcal{F}_1 \vdash \{\!|e_1, n_1|\!\} \mapsto \{\!|e_1', n_1'|\!\}$ *learning* $\mathcal{F}_1'$,
- $\Sigma; W; \mathcal{F}_2 \vdash \{\!|e_2, n_2|\!\} \mapsto \{\!|e_2', n_2'|\!\}$ *learning* $\mathcal{F}_2'$,
- *there is no* p *such that* e_1' blames p, *and*
- *there is no* p *such that* e_2' blames p,

then $W_0 \vdash e_1' \simeq e_2'$.

5. Discussion

Information-flow and Aura Information-flow analyses [32] inspired this paper's goal of augmenting Aura to handle confidential data. However, while these techniques influenced and informed the design of AuraConf, they cannot be directly applied.

In standard information-flow systems, programmers use *labels* to express confidentiality and integrity constraints on data, and the language's typing judgment is specialized to deal with these labels [40]. Well-typed terms are correct by construction; they satisfy noninterference. (However, increasingly expressive information-flow languages often satisfy variously weakened versions of the property.) Most conventional information-flow languages are limited by a focus on closed systems: the programmer must, for example, manually encrypt confidential data leaving the program with an unsafe *declassification* operator.

Aura can encode this style of of information flow analysis [22]. However, this encoding makes also makes "closed world" assumptions—attackers are assumed to respect Aura's runtime invariants and confidential data is not protected by cryptography. This work provides a means for programmers to rule out implementation errors on a single host, but, unlike AuraConf, does not directly address distributed aspects of confidentiality. Similar information flow encodings are possible in Fine [35], and Haskell [24].

In previous work with Zdancewic [39], I described an information-flow language, SImp, suitable for programming in open systems. SImp resolves the mismatch between policy specification and enforcement by connecting information flow labels directly with public key cryptography. Policies and data may be combined into *packages* that use digital signatures and encryption to ensure only principals with appropriate keys may access data.

SImp policies are specified by annotating data values and heap locations with semantically rich *labels*. Labels are lists of security sublabels with owner, confidentiality, and integrity components. Sublabel $o : \overline{r} ! \overline{w}$ means owner o certifies that any principal in set $\overline{r}$ may read from the associated location, and any principal in set $\overline{w}$ may write. Full labels allow (groups of) principals to read or write when each sublabel is satisfied. This is a variant of Myers and Liskov's [29] decentralized label model (DLM).

Although SImp's design influenced AuraConf, its technical mechanisms could not be adopted wholesale. Information-flow analysis with DLM labels, the basis for SImp, provides a very different model of declarative information security than Aura. In particular Aura's **says** monad decorates propositions to express *endorsement*, while SImp's integrity sublabels described tainted data. While intuitively related, these concepts demand different treatments. It is unclear how to understand DLM owners in Aura. Additionally, interpreting the semantics of a DLM label—that is, calculating its effective reader and writer sets—requires knowledge of the global delegation relation, or "acts-for hierarchy," information that cannot be reliably obtained in Aura's distributed setting. SImp's design did provide direct inspiration for several aspects of AuraConf, including declarative policy specification, a key-based notion of identity, and automatic encryption.

On Noninterference AuraConf's noninterference property (Lemma 1) is weak in the following sense. It discusses what happens when a pair of terms with different secrets successfully take a step, but does not deal with the situation in which one steps successfully and the other fails. The reason is subtle. Consider the following definitions and terms:

```
data Singleton : bits → Type
{ | inject : (t: bits) → Singleton t }

ok: Singleton E(a, "hi", 1) → Prop
e1 ≡ ok( inject (E(a, "hi", 1)) )
e2 ≡ ok( inject (E(a, "hi", 2)) )
```

Terms e_1 and e_2 represent differently randomized encryptions of the same string. It's intuitively appealing that these are similar for purposes of noninterference, and indeed $\mathsf{a} \vdash e_1 \simeq e_2$. However term e_1 is well-typed, but e_2 is not. Terms like these can cause $\mathbf{run_f}$ to show different failure behavior when applied to similar terms. Consequently, Lemma 1's definition of noninterference is an example of termination-insensitive noninterference [4].

Termination insensitivity is required because AuraConf and its metatheory have the following three properties. First, the language can express singleton types on ciphertexts—useful in general and necessary for **isa** propositions. Second, it features a type-safe decryption operator that works at arbitrary types—a design goal. Third, the similarity relation is aligned with standard Dolev-Yao [1983] cryptanalysis. While it's possible to alter one of these properties to induce a stronger form of noninterference, such a change appears counterproductive.

6. Related Work

Modal logics provide a framework to describe the way in which a proposition holds. Common modalities can specify that a sentence is necessarily vs. possibly true or that a condition will be met eventually vs. from-now-on. In the vernacular of Kripke structures this is a technique for reasoning about different worlds, a terminology that AuraConf borrows [17]. Pfenning and Davies [30] introduced a constructive, type-theoretic treatment of modal logic. Their account focuses on the logical foundations of the system. Jia and Walker [21] studied a similar theory from a distributed-programming per-

spective, interpreting modal operators as specifying the locations at which code may run. While Pfenning discusses three judgments: truth, validity and possibility, Jia presents an indexed judgment form that can describe a large quantity of locations. Murphy's [27] dissertation describes a full-scale programming language based on these ideas.

The systems above have an absolute static semantics. That is, although executing code may depend on location or resource availability, checking that a type (or proposition) is well-formed can happen anywhere. AuraConf's ability to make typing more precise using statically available keys appears novel.

One intended semantics for AuraConf implements objects of form $\mathsf{sign}(\mathtt{a}, \mathtt{P})$ as digital signatures and objects like $\mathcal{E}(\mathtt{a}, \mathtt{e}, \mathtt{n})$ as ciphertext. All cryptography occurs at a lower level of abstraction than the language definition. This approach has previously been used to implement or model declarative information flow policies [25, 39]. An alternative approach is to treat keys as types or first class objects and to provide encryption or signing primitives in the language [3, 15, 23, 33]. Such approaches typically provide the programmer with additional flexibility but complicate the programming model; in contrast, a goal of AuraConf is to free the programmer from such explicit key management.

Askarov and Sabelfeld [5] describe both approaches to information flow and give a translation of a high-level language with a declassification operator into a low-level language that implements declassification via publication of specific cryptographic keys. Similarly to this paper, Askarov and Sabelfeld's work uses an algebraic model of randomized cryptography to demonstrate that noninterference is preserved when secrets are encrypted. Their model can describe declassification directly, something not done here, and it would be interesting to extend the analysis of AuraConf to account for intentional declassification.

Sumii and Pierce [34] studied λ_{seal}, an extension to lambda calculus with terms of form $\{e\}_{e'}$, meaning e sealed-by e', and a corresponding elimination form. Unlike AuraConf, λ_{seal} makes seal (i.e. key) generation explicit in program text. Additionally, λ_{seal} includes black-box functions that analyze sealed values, but cannot be disassembled to reveal the seal (key). It is unclear how to implement these functions using cryptography.

Heintze and Riecke's [18] SLam calculus is an information flow lambda calculus in which the right to read a closure corresponds to the right to apply it. This sidesteps the black-box function issue from λ_{seal}. In SLam, some expressions are marked with the function writer's authority. This differs from AuraConf's notion of dynamic authority which describes the program's available keys.

We use the algebraic Dolev-Yao model to study the connection between information flow and cryptography. Laud and Vene [23] examined this problem using a computational model of encryption. More recently, Smith and Alpízar [33] extended this work to include a model of decryption. They prove noninterference for a simple language without declassification (or packing) and a two-point security lattice.

Abadi and Rogaway [2] proved that Dolev-Yao analysis is sound with respect to computational cryptographic analysis in a setting similar to Vaughan and Zdancewic's [39]. However, there are several significant differences between these approaches. In particular, Abadi and Rogaway do not discuss public key cryptography, which we use extensively. Backes and Pfitzmann [6] with Waidner [7] have also investigated the connection between symbolic and computational models of encryption. They define a Dolev-Yao style library and show that protocols proved secure with respect to library semantics are also secure with respect to computational cryptographic analysis. Likewise Barthe et al. [8] have published a Coq formalization of several cryptographic algorithms, including ElGamal digital signatures. These techniques and artifacts might provide an excellent foundation for further rigorous analysis of AuraConf.

7. Conclusions and Future Work

AuraConf's treatment of cryptography includes several novel elements. Because AuraConf uses statically available keys and fact contexts to augment compile-time typechecking, it places unusual demands on its type system. In particular, the very notion of well-typedness is dependent on which keys are available statically, and it is challenging to predict where a term can typecheck. Additionally, evaluation can also use private keys, and programs will get stuck if run in the wrong context. AuraConf answers "where can a term be typechecked?" and "where can it be run?" by combining, in a new way, ideas from modal type theory and type-and-effect analysis.

AuraConf trades increased complexity for increased ability to detect and debug security problems both statically via type checking and dynamically via audit. It appears possible to reduce complexity by giving up certain features, such as higher-order confidential data or static detection of some key-management bugs. It would be interesting to explore and evaluate these and other points in the design space.

Currently AuraConf provides a fail-stop semantics: decryption failures lead to an uncatchable, fatal exception. It would be better to handle these errors programmatically, and a variety of techniques could be brought to bear. Most promising, AuraConf could be extended with a general purpose exception mechanism like ML's or Java's. Doing this properly requires merging exceptions with effects analysis with higher-order types, a topic of active research [10, 11]. A more readily implementable scheme might make run_f return a discriminated union.

AuraConf's worlds provide a simple model of key management. Programs may be run and typechecked using zero or one statically available keys. The typing judgment additionally allows $\top$ to represent "all keys" in effect labels and soft decryption limits. However worlds are treated primarily as lattice elements and it appears interesting to define worlds using a richer structure, such as sets of keys or DLM-inspired labels. This generalization would require making (hopefully) straightforward modifications to the language metatheory and, more interestingly, carefully designing an expressive and practical security lattice.

Acknowledgments

This work was initiated as part of my dissertation at the University of Pennsylvania. Thank you to my Ph.D. advisor, Steve Zdancewic, and to my thesis committee, Benjamin C. Pierce, Frank Pfenning, Andre Scedrov, and Stephanie Weirich. Their feedback greatly strengthened the ideas in this paper.

References

[1] M. Abadi. Access control in a core calculus of dependency. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pp. 263–273. ACM, 2006.

[2] M. Abadi, P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *Journal of Cryptology*, 15(2):103–127, 2002.

[3] A. Askarov, D. Hedin, A. Sabelfeld. Cryptographically-masked flows. In *SAS '06*, LNCS. Seoul, Korea, 2006.

[4] A. Askarov, S. Hunt, A. Sabelfeld, D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08*, pp. 333–348. 2008.

[5] A. Askarov, A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *SP '07: Proceedings of the*

2007 IEEE Symposium on Security and Privacy, pp. 207–221. IEEE Computer Society, Washington, DC, USA, 2007.

[6] M. Backes, B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *IEEE Trans. Dependable Secur. Comput.*, 2(2):109–123, 2005.

[7] M. Backes, B. Pfitzmann, M. Waidner. A composable cryptographic library with nested operations. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications security*, pp. 220–230. ACM Press, Washington D.C., USA, 2003.

[8] G. Barthe, B. Grégoire, S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 90–101. ACM, New York, NY, USA, 2009.

[9] L. Bauer, S. Garriss, M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security & Privacy*, pp. 81–95. 2005.

[10] N. Benton, P. Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '07)*. ACM, Nice, France, 2007.

[11] M. Blume, U. A. Acar, W. Chae. Exception handlers as extensible cases. In *Proceedings of the Sixth ASIAN Symposium on Programming Languages and Systems (APLAS 2008)*. Bangalore, India, 2008. To appear.

[12] H. B. Curry, R. Feys, W. Craig. *Combinatory Logic*, vol. 1. North-Holland, Amsterdam, 1958.

[13] D. Dolev, A. Yao. On the security of public key protocols. In *IEEE Transactions on Information Theory*, 2(29), 1983.

[14] C. Fournet, A. D. Gordon, S. Maffeis. A type discipline for authorization in distributed systems. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*. 2007.

[15] C. Fournet, T. Rezk. Cryptographically sound implementations for typed information-flow security. In *POPL '08*, pp. 323–335. 2008.

[16] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09*. 2009.

[17] R. Goldblatt. Mathematical modal logic: a view of its evolution. In *J. of Applied Logic*, 1(5-6):309–392, 2003.

[18] N. Heintze, J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL '98*, pp. 365–377. ACM Press, New York, NY, 1998.

[19] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin, J. R. Hindly, eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pp. 479–490. Academic Press, New York, 1980.

[20] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, S. Zdancewic. Aura: A programming language for authorization and audit. In *ICFP '08*, pp. 27–38. 2008. Extended version U. Penn. Tech. Rep. MS-CIS-08-10.

[21] L. Jia, D. Walker. Modal proofs as distributed programs (extended abstract). In *ESOP '04*. 2004. Full version Princeton Tech. Rep. TR-671-03.

[22] L. Jia, S. Zdancewic. Encoding information flow in aura. In *PLAS '09*. 2009.

[23] P. Laud, V. Vene. A type system for computationally secure information flow. In *FCT '05*, pp. 365–377. Lübeck, Germany, 2005.

[24] P. Li, S. Zdancewic. Encoding information flow in haskell. In *CSFW '06*. 2006.

[25] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 321–334. ACM, New York, NY, USA, 2009.

[26] J. M. Lucassen, D. K. Gifford. Polymorphic effect systems. In *POPL '88*. 1988.

[27] T. Murphy, VII. *Modal Types for Mobile Code*. Ph.D. thesis, CMU, 2008.

[28] T. Murphy, VII, K. Crary, R. Harper, F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS '04*. IEEE Press, 2004.

[29] A. Myers, B. Liskov. Protecting privacy using the decentralized label model. In *ACM Trans. on Software Engineering and Methodology*, 9(4):410–442, 2000.

[30] F. Pfenning, R. Davies. A judgmental reconstruction of modal logic. In *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, 2001.

[31] D. K. Rappe. *Homomorphic Cryptosystems and Their Applications*. Ph.D. thesis, University of Dortmund, Germany, 2004.

[32] A. Sabelfeld, A. C. Myers. Language-based information-flow security. In *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[33] G. Smith, R. Alpízar. Secure information flow with random assignment and encryption. In *FSME'06*, pp. 33–43. Alexandria, Virgina, USA, 2006.

[34] E. Sumii, B. C. Pierce. A bisimulation for dynamic sealing. In *POPL '04*. 2004.

[35] N. Swamy, J. Chen, R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ESOP '10*. 2010.

[36] J.-P. Talpin, P. Jouvelot. The type and effect discipline. In *LICS '92*. 1992.

[37] J. A. Vaughan. *Aura: Programming with Authorization and Audit*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, USA, 2009.

[38] J. A. Vaughan, L. Jia, K. Mazurak, S. Zdancewic. Evidence-based audit. In *CSF '08*, pp. 177–191. 2008. Extended version U. Penn. Tech. Rep. MS-CIS-08-09.

[39] J. A. Vaughan, S. Zdancewic. A cryptographic decentralized label model. In *IEEE Security and Privacy*, pp. 192–206. Berkeley, California, 2007.

[40] D. Volpano, G. Smith, C. Irvine. A sound type system for secure flow analysis. In *Journal of Computer Security*, 4(3):167–187, 1996.