

A Confidentiality Extension to the Aura Programming Language

Jeffrey A. Vaughan

Harvard University*

Abstract. The core Aura language [7, 21] supports a security-focused programming model that mixes access control, audit, and ML-like computations. This paper presents a language extension, AuraConf, which adds support for the secure handling of confidential data. AuraConf allows a confidentiality policy 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.

1 Introduction

This paper introduces $\text{Aura}_{\text{conf}}$, a confidentiality extension to the Aura programming language [7, 21].

Aura was developed as a platform for programming with access control and audit. Programs construct proofs of their access-control rights at runtime. Such proofs are consumed and logged by procedures that performs secure operations. Dependent types provide a precise way to connect access-control predicates with specific data values. Mutually distrusting principals may use digitally signed propositions to introduce new access-control policies.

Access-control and audit alone are insufficient for programs that deal with confidential data. For instance, applications in medical, financial, and legal arenas need to be able process and, as appropriate, share secrets.

This paper introduces $\text{Aura}_{\text{conf}}$, an extension to Aura that provides programmers with a principled set of tools for handling secret information. The goals of this design are as follows.

- To establish a natural connection between confidential language expressions and cryptography.
- To leverage Aura’s expressive core features to provide a cohesive and useful design.
- To provide technical mechanisms for associating decryption failures with proof objects that can be logged for later analysis.

* This work was completed while the author was at the University of Pennsylvania.

Mixing an informative type system with encryption 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 $\text{Aura}_{\text{conf}}$ 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 [8, 12]. Ensuring that needed keys are available dynamically requires type-and-effect analysis [11, 20]. (In principle other techniques could be used, but the concepts involved are essential.)

The $\text{Aura}_{\text{conf}}$ language resolves the tensions indicated above and helps programmers to safely handle confidential data.

Meeting these goals requires two substantial technical contributions:

- The design of $\text{Aura}_{\text{conf}}$ including the confidentiality type constructor **for** and a sophisticated type system that enforces both key-management and code-mobility constraints.
- A mechanized proof that $\text{Aura}_{\text{conf}}$ is syntactically sound and satisfies a non-interference property.

2 Confidential Computations and the For-Monad

This section provides an informal introduction to $\text{Aura}_{\text{conf}}$ ’s new features.

In $\text{Aura}_{\text{conf}}$ 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 private key, yielding a blob of ciphertext, written $\mathcal{E}(\text{Alice}, 42, 0x2b63)$, 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**, $\text{Aura}_{\text{conf}}$ 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)
  ( $\lambda\{\text{Alice}\} x: \text{int} . \text{return Alice (x * 2)$ )
: 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 $\{\text{Alice}\}$ component of the λ ; this is an effect annotation and will be defined and discussed later.

As illustrated above, the for-monad treats its arguments lazily. Imagine for the moment that we could use homomorphic encryption [4, 16] 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 termination 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 $\mathcal{E}(\text{Alice}, 42, 0x2b63)$ with type **int for Alice**. The entire point of encryption is to ensure that users other than Alice cannot meaningfully inspect the ciphertext, and Bob has no way to decompose and examine the newly created object.

$\text{Aura}_{\text{conf}}$ resolves this tension as follows. Ciphertexts may be annotated with one of two forms of typing metadata. First, the term

```
cast  $\mathcal{E}(\text{Alice}, 42, 0x2b63)$  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 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,

$p : (\text{Bob says } (c \text{ isa } (\text{int for Alice})))$.

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

In combination, true and justified casts allows 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.

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($\mathcal{E}(\text{Alice}, 42, 0x2b63)$) to (int for Alice)) : bits

Type **bits** classifies naked ciphertexts, and the above term reduces to

$\mathcal{E}(\text{Alice}, 42, 0x2b63) : \text{bits}$.

3 An Example

This section shows a sample $\text{Aura}_{\text{conf}}$ program. For illustration, we use modules—a feature that is not part of the formal language definition.

Figure 1 defines a simple networking interface. The functions **send** and **recv** are intended to send and receive data values. The 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 the 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

```

(* This interface provides a basic API for a network library *)
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

```

Fig. 1: A simple communications library.

a proof object permitting arbitrary network writes. See Vaughan et al. [21] for an overview of access-control in Aura.

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 (int for Bob) 312 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 **casts**, **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, soft decryption limit, and effect label all equal to singleton world Alice. Furthermore, the code must be run with authority Alice.

4 Language Definition

This section describes the definition of $\text{Aura}_{\text{conf}}$ and its metatheory.

In type-safe languages such as $\text{Aura}_{\text{conf}}$, 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. $\text{Aura}_{\text{conf}}$ ’s type system, like Aura’s or ML’s, rules out these particular errors. However, an $\text{Aura}_{\text{conf}}$ 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.

We address the first challenge by constraining the canonical forms of **for** types. Enforcing these constraints requires types and terms have (loosely) consistent meanings to typecheckers with different capabilities, i.e. different access to private keys. $\text{Aura}_{\text{conf}}$ ’s type system accomplishes this using ideas based on modal type systems for distributed computing [8, 12, 13].

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 [11, 20].

Syntax $\text{Aura}_{\text{conf}}$ adds to and modifies Aura’s syntax. These changes are summarized in Figure 2, and include the new operators introduced in Section 2. Not shown are standard functional-programming constructs like pattern matching and access-control features like digital signatures. To enable the type-and-effect analysis described above, abstractions and arrows are labeled with *worlds* that summarize latent uses of private keys.

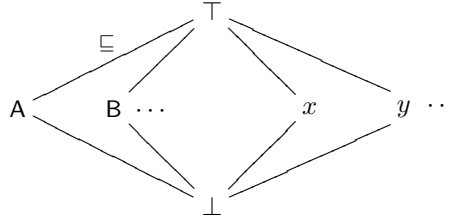
Syntactically, the set of worlds is the set of terms augmented with distinguished top and bottom elements. $\text{Aura}_{\text{conf}}$ ’s static semantics identify only some worlds as well formed: namely principal constants, variables of type **prin**, \top , and \perp . We define a partial order on worlds,

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

Worlds	
$W, V, U ::= \perp$	Bottom world (no keys)
t	Singleton worlds
\top	Top world (all keys)
Terms	
$t ::= \dots$	(Standard Aura syntax)
$(x:t) \rightarrow_{\{W\}} t$	Implication, quantification, arrow
$\lambda_{\{W\}} x:t. t$	Abstraction
$a \text{ for } P$	Type of encrypted data
$\text{return}_f e \text{ as } t$	Private data for a
$\text{bind}_f x = e_1 \text{ in } e_2 \text{ as } t$	Private computation
$\text{run}_f e$	Extract private data
$\text{cast } e \text{ to } t \text{ blaming } p$	Cast using type-evidence
$\text{cast } e \text{ to } t$	Emperical cast
$\text{say } a P$	Saying
$\mathcal{E}(a, e, n)$	Ciphertext
$\text{fail } p$	Decryption-failed exception

Fig. 2: Aura_{conf} Syntax

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.

Unlike in Aura, the **say** operator is now annotated with a principal, so that

say Alice P : Alice **says** P

This allows removal of the **self** constant and makes for a consistent treatment of principals throughout Aura_{conf}.

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 Aura_{conf}'s static semantics is based on Aura's, but with several substantial changes. Aura_{conf} typechecks programs using the following

judgments.

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

The typing relation (well-typed term) is huge. 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

$$\begin{aligned} &\text{Fact Contexts} \\ \mathcal{F} ::= &\cdot \mid \mathcal{F}, \mathcal{E}(a, e, n) : t. \end{aligned}$$

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.

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 on which decryption keys or facts should be used when typechecking a particular term. Intuitively the keys used to check a term are $W \sqcup V$. 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 $V = \text{Bob}$ 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 may be found in the appendix and the author's thesis, to be published in advance of ESOP.) WF-TM-FORRET has form

$$\frac{_ ; \mathcal{F}; W \mid _ ; a; _ \vdash e : t \quad a \sqsubseteq V \quad \dots}{_ ; \mathcal{F}; W \mid _ ; V; _ \vdash \text{return}_f e \text{ as } (t \text{ for } a) : t \text{ for } a .}$$

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 left of the vertical bar differed between WF-TM-FORRET's premise and conclusion, but symbols on the bar's right 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.

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

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

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

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 .

Aura_{conf} 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, Aura_{conf} environments are defined by the following grammar.

Environments

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

Equalities in the environment enable type refinement as in core Aura [7].

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 $U = \perp$ indicates that an expression is *pure*—that it can execute with no private keys. For instance, **say** signs a proposition. Its typed as follows:

$$\frac{\Sigma; \mathcal{F}; W | E; \perp; \perp \vdash P : \mathbf{Prop} \quad \Sigma; \mathcal{F}; W | E; \perp; \perp \vdash a : \mathbf{prin} \quad a \sqsubseteq U \quad \dots}{\Sigma; \mathcal{F}; W | E; V; U \vdash \mathbf{say} a P : \mathbf{pf} a \mathbf{says} P}$$

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

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 $\lambda x.e$. 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 \perp 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 [8] **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.

Auxiliary judgments 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 $W = V = \perp$ and $\mathcal{F} = \cdot$.

Definitions of environment, world, and worlds well-formedness are more novel and are detailed in by Figure 3.

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 \perp . The well-formed environment relation also ensures that the statically available key is a *simple world*—either a principal constant or \perp .

The well-formed world relation always accepts \top and \perp . 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.

New and modified language constructs Moving from Aura to $\text{Aura}_{\text{conf}}$ 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 $\text{Aura}_{\text{conf}}$. This is necessary to work with soft decryption limits and effect labels.

The variable rule is

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

From an $\text{Aura}_{\text{conf}}$ 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 .

$\boxed{\text{simple } W}$

$$\frac{}{\text{simple } \perp} \qquad \frac{a \in \{A, B, C \dots\}}{\text{simple } a}$$

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

$$\frac{\text{simple } W}{\Sigma; \mathcal{F}; W \mid \cdot} \qquad \frac{\Sigma; \mathcal{F}; W \mid \vdash E \quad \Sigma; \mathcal{F}; W \mid E; V \vdash \diamond \quad \Sigma; \mathcal{F}; W \mid E; \perp; \perp \vdash t : k}{x \text{ fresh} \quad (k \in \{\mathbf{Type}, \mathbf{Prop}\})(t \in \{\mathbf{Type}, \mathbf{Prop}\} / V = \perp)}{\Sigma; \mathcal{F}; W \mid \vdash E, x : t \text{ at } V}$$

$$\frac{\Sigma; \mathcal{F}; W \mid \vdash E \quad \Sigma; \mathcal{F}; W \mid E; \perp; U \vdash e_1 : t \quad \Sigma; \mathcal{F}; W \mid E; \perp; U \vdash e_2 : t \quad \text{atomic } \Sigma t \quad x \text{ fresh}}{\text{value } e_1 \quad \text{value } e_2 \quad \Sigma; \mathcal{F}; W \mid E; \perp; \perp \vdash t : \mathbf{Type} \quad \Sigma; \mathcal{F}; W \mid E; V \vdash \diamond}{\Sigma; \mathcal{F}; W \mid \vdash E, x \sim (e_1 = e_2) : t \text{ at } V}$$

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

$$\frac{\Sigma; \mathcal{F}; W \mid \vdash E}{\Sigma; \mathcal{F}; W \mid E; \perp \vdash \diamond} \qquad \frac{\Sigma; \mathcal{F}; W \mid E; \perp; \perp \vdash a : \mathbf{prin} \quad \text{value } a}{\Sigma; \mathcal{F}; W \mid E; a \vdash \diamond} \qquad \frac{\Sigma; \mathcal{F}; W \mid \vdash E}{\Sigma; \mathcal{F}; W \mid E; \top \vdash \diamond}$$

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

$$\frac{\Sigma; \mathcal{F}; W \mid E; V \vdash \diamond \quad \Sigma; \mathcal{F}; W \mid E; U \vdash \diamond}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond}$$

Fig. 3: Major auxiliary judgments for $\text{Aura}_{\text{conf}}$'s static semantics.

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

$$\frac{\Sigma; \mathcal{F}; W \mid E; V; U_0 \vdash \diamond \quad \Sigma; \mathcal{F}; W \mid E, x : u_1 \text{ at } \perp; V; U_0 \vdash f : u_2 \quad \dots}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \lambda_{\{U_0\}} x : u_1. f : (x : u_1) \rightarrow_{\{U_0\}} u_2}.$$

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.

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

$$\frac{\Sigma; \mathcal{F}; W \mid E; V; U \vdash e_1 : (x : t_2) \rightarrow_{\{U_0\}} u \quad \Sigma; \mathcal{F}; W \mid E; \perp; U_2 \vdash e_2 : t_2 \quad (\text{value } e_2 \wedge U_2 = \perp) \vee (x \notin \text{fv}(u) \wedge U_2 = U) \quad U_0 \sqsubseteq U \quad \dots}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash e_1 e_2 : \{x/e_2\}u}$$

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 an effect label up to U .

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

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

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 **at** V_1 . The restriction $V_1 \sqsubseteq V$ is necessary to prevent **let at**s from raising the soft decryption limit and allowing the unsafe use of facts or statically available keys. While **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 $\text{Aura}_{\text{conf}}$ type system always interprets unannotated ciphertexts as unintelligible blobs.

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

As discussed above, more precise typings maybe 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.

$$\frac{\Sigma; \mathcal{F}; W|E; a; a \vdash e : t \quad a \sqsubseteq V \quad \dots}{\Sigma; \mathcal{F}; W|E; V; U \vdash \text{return}_f e \text{ as } (t \text{ for } a) : t \text{ for } a} .$$

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 **run_f**. Typing for **bind_f** works analogously; see rule WF-TM-FORBIND.

The **run_f** operator decrypts and evaluates annotated ciphertexts. It's typed by:

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash e : t \text{ for } a \quad a \sqsubseteq V \quad a \sqsubseteq U}{\Sigma; \mathcal{F}; W|E; V; U \vdash \text{run}_f e : t}$$

The premise $a \sqsubseteq U$ forces effect label U to record that the **run_f** uses a 's private key. Premise $a \sqsubseteq V$ prevents problems with nested occurrence of **run_f**.

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

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

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 $t \text{ for } a$. The approximation, formalized in Figure 4, 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.

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

$$\frac{(x : t \text{ at } V) \in E}{E \sim x : t} \text{ GE-TM-VAR} \qquad \frac{}{E \sim \mathcal{E}(a, e, n) : \text{bits}} \text{ GE-TM-ENC}$$

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

Fig. 4: Approximate typing judgment used by WF-TM-ASBITS

Dynamic semantics The dynamic semantics for $\text{Aura}_{\text{conf}}$ 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' . Finally \mathcal{F} is a fact context, with zero or one elements, containing facts about freshly created ciphertexts.

In general $\text{Aura}_{\text{conf}}$'s evaluation relation subsumes Aura 's. For intuition, when $e \mapsto e'$ in Aura ,

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

holds in $\text{Aura}_{\text{conf}}$. Figure 5 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 , $\text{STEP-FORRUN-ILLTYPED}$, and STEP-FORRUN-JUNK attempt to decrypt and typecheck an annotated ciphertext, signaling an error as needed.

Figure 5 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 $\text{Aura}_{\text{conf}}$ satisfies two important properties: syntactic soundness and noninterference. Syntactic soundness guarantees that all well-typed programs have a well-defined evaluation semantics. Noninterference [2, 23], states that a program's outputs are not affected (up to a natural equivalence induced by cryptography) by inputs intended to be secret. $\text{Aura}_{\text{conf}}$'s type system has several non-standard aspects; consequently, the technical statements and proofs of these properties are novel.

Except as noted, all properties of $\text{Aura}_{\text{conf}}$ are formalized as constructive proofs in the Coq proof assistant. Such formalization is particularly important for large languages and security-focused languages; $\text{Aura}_{\text{conf}}$ 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 $(\mathbf{fail} \ p)$ is a subterm of e , not located under a \mathbf{return}_f or a \mathbf{bind}_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 $\Sigma \vdash \diamond$ and $\Sigma; \mathcal{F}_0; W|E; V; U \vdash e : t$. Then $\Sigma; \mathcal{F}_0; W \vdash \{e, n\} \mapsto \{e', n'\} \text{ learning } \mathcal{F}$ implies either $\Sigma; \mathcal{F}_0 \vdash \mathcal{F}; W|E; V; U \vdash e' : t$ or there exists p such that e' blames p .*

The above lemma misses an important aspect of evaluation. Running an $\text{Aura}_{\text{conf}}$ 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.

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

$$\frac{\text{value } v}{\Sigma; F_0; W \vdash \{\mathbf{let } x \text{ at } V = v \text{ in } e, n\} \mapsto \{\{v/x\}e, n\} \text{ learning } \cdot} \text{STEP-LETAT}$$

$$\frac{}{\Sigma; F_0; W \vdash \langle \mathbf{return}_f e \text{ as } (t \text{ for } a), n \rangle \mapsto \langle \mathbf{cast } \mathcal{E}(a, e, n) \text{ to } (t \text{ for } a), n + 1 \rangle \text{ learning } \mathcal{E}(a, e, n) : t \text{ for } a} \text{STEP-FORRET}$$

$$\frac{\text{value } v}{\Sigma; F_0; W \vdash \langle \mathbf{bind}_f x = v \text{ in } e \text{ as } t \text{ for } a, n \rangle \mapsto \langle \mathbf{cast } \mathcal{E}(a, \mathbf{let } x \text{ at } a = (\mathbf{run}_f v) \text{ in } (\mathbf{run}_f e), n) \text{ to } (t \text{ for } a), n + 1 \rangle \text{ learning } \mathcal{E}(a, \mathbf{let } x \text{ at } a = (\mathbf{run}_f v) \text{ in } (\mathbf{run}_f e), n) : t \text{ for } a} \text{STEP-FORBIND}$$

$$\frac{\text{value}(\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } t [\mathbf{blaming } p])}{\Sigma; F_0; W \vdash \{\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } t [\mathbf{blaming } p], n\} \mapsto \{\mathcal{E}(a, e, m), n\} \text{ learning } \cdot} \text{STEP-ASBITS}$$

$$\frac{\text{value}(\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } t [\mathbf{blaming } p]) \quad \Sigma; F_0; W \vdash a; a \vdash e : t \quad a \sqsubseteq W}{\Sigma; F_0; W \vdash \{\mathbf{run}_f (\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } t [\mathbf{blaming } p]), n\} \mapsto \{e, n\} \text{ learning } \cdot} \text{STEP-FORRUN-OK}$$

$$\frac{\text{value}(\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } t \mathbf{blaming } p) \quad \Sigma; F_0; W \vdash a; a \not\vdash e : t \quad a \sqsubseteq W}{\Sigma; F_0; W \vdash \{\mathbf{run}_f (\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } t \mathbf{blaming } p), n\} \mapsto \{\mathbf{fail } p, n\} \text{ learning } \cdot} \text{STEP-FORRUN-ILLTYPED}$$

$$\frac{\text{value}(\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } (t \text{ for } b) \mathbf{blaming } p) \quad b \sqsubseteq W \quad a \neq b}{\Sigma; F_0; W \vdash \{\mathbf{run}_f (\mathbf{cast } \mathcal{E}(a, e, m) \text{ to } (t \text{ for } b) \mathbf{blaming } p), n\} \mapsto \{\mathbf{fail } p, n\} \text{ learning } \cdot} \text{STEP-FORRUN-JUNK}$$

$$\frac{\Sigma; F_0; W \vdash \{e_1, n\} \mapsto \{e'_1, n'\} \text{ learning } F}{\Sigma; F_0; W \vdash \{e_1 e_2, n\} \mapsto \{e'_1 e_2, n'\} \text{ learning } F} \text{STEP-APP-CONGL}$$

Fig. 5: Selected $\text{Aura}_{\text{conf}}$ evaluation rules.

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 $\mathcal{E}(a, e, n) : t \text{ for } b$ in \mathcal{F} it is the case that $a = b$ and $\Sigma; \cdot; b \vdash b; b \vdash e : t$.*

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.

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 $\text{Aura}_{\text{conf}}$'s type system.

The following lemma shows facts generated during reduction are valid.

Lemma 2 (New Fact Validity). *Assume $\Sigma \vdash \diamond$ 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 $\text{valid}_\Sigma \mathcal{F}$.*

We assume that, as shown for Aura in Jia et al. [7] type checking $\text{Aura}_{\text{conf}}$ is decidable. This is a reasonable conjecture because $\text{Aura}_{\text{conf}}$ is syntax directed and designed with decidability in mind. Transliterating the earlier proof would be tedious but should yield no deep difficulties or insights. Decidability is of independent theoretic interest, but also matters because evaluating run_f dynamically invokes the type checker. Were typing undecidable, run_f could instead conservatively approximate; otherwise the progress lemma would not hold.

Conjecture 1 (Decidability). If $\Sigma \vdash \diamond$, it is decidable if $\Sigma; \mathcal{F}; W|E; V; U \vdash e : t$.

The $\text{Aura}_{\text{conf}}$ 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 3 (Progress). *Assume Conjecture 1 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} .*

Lemmas 1, 2, and 3 together imply that $\text{Aura}_{\text{conf}}$ 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 [2, 22, 23]. 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 $\text{Aura}_{\text{conf}}$.

$\text{Aura}_{\text{conf}}$ similarity is defined relative to particular set of keys used to analyze ciphertexts. Figure 6 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.

The following lemma gives $\text{Aura}_{\text{conf}}$ ’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

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

$$\frac{}{W \vdash x \simeq x} \text{SIM-VAR} \qquad \frac{W \vdash t_{12} \simeq t_{12} \quad W \vdash t_{21} \simeq t_{22}}{W \vdash (t_{11} \ t_{21}) \simeq (t_{12} \ t_{22})} \text{SIM-APP}$$

$$\frac{a \sqsubseteq W \quad W \vdash e_1 \simeq e_2}{W \vdash \mathcal{E}(a, e_1, n_1) \simeq \mathcal{E}(a, e_2, n_2)} \text{SIM-DECRYPT}$$

$$\frac{a \not\sqsubseteq W \quad b \not\sqsubseteq W}{W \vdash \mathcal{E}(a, e_1, n_1) \simeq \mathcal{E}(b, e_2, n_2)} \text{SIM-OPAQUE}$$

Fig. 6: Selected rules from the definition of similar terms.

and e'_2 , are similar as well. That is, running a program twice with two different confidential inputs yields similar outputs.

Lemma 4 (Noninterference). *Assume $\Sigma \vdash \diamond$ 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 [17] inspired this chapter’s goal of augmenting Aura to handle confidential data. However, while these techniques influenced and informed the design of $\text{Aura}_{\text{conf}}$, 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 [23]. 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 information flow analysis [9].

In previous work with Steve Zdancewic [2007], 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 $o : \bar{r}! \bar{w}$ means owner o certifies that any principal in set \bar{r} may read from the associated location, and any principal in set \bar{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 [14] decentralized label model (DLM).

Although SIMp’s design influenced $\text{Aura}_{\text{conf}}$, 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 $\text{Aura}_{\text{conf}}$, including declarative policy specification, a key-based notion of identity, and automatic encryption.

On Noninterference $\text{Aura}_{\text{conf}}$ ’s noninterference property (Lemma 4) 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 terms:

$$\begin{aligned} \text{ok} &: \mathcal{E}(\text{a}, \text{"hi"}, 1) \rightarrow \mathbf{Prop} \\ e_1 &\equiv \text{ok}(\mathcal{E}(\text{a}, \text{"hi"}, 1)) \\ e_2 &\equiv \text{ok}(\mathcal{E}(\text{a}, \text{"hi"}, 2)) \end{aligned}$$

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 $\text{a} \vdash e_1 \simeq e_2$. However term e_1 is well-typed, but e_2 is not. Terms like these can cause run_f to show different failure behavior when applied to similar terms. Consequently, Lemma 4’s definition of noninterference is an example of termination-insensitive noninterference [2].

Termination insensitivity is required because $\text{Aura}_{\text{conf}}$ 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 typesafe 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 $\text{Aura}_{\text{conf}}$ borrows [5]. Pfenning and Davies [15] introduced a constructive, type-theoretic treatment of modal logic. Their account focuses on the logical foundations of the system. Jia and Walker [8] studied a similar theory from a distributed-programming perspective, 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 [12] 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. $\text{Aura}_{\text{conf}}$ ’s ability to make typing more precise using statically available keys appears novel.

One intended semantics for $\text{Aura}_{\text{conf}}$ implements objects of form $\mathbf{sign}(a, P)$ as digital signatures and objects like $\mathcal{E}(a, e, n)$ as ciphertext. All cryptography occurs at a lower level of abstraction than the language definition. This approach has previously been used to implement declarative information flow policies [22]. An alternative approach is to treat keys as types or first class objects and to provide encryption or signing primitives in the language [1, 10, 10, 18].

Sumii and Pierce [19] 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 $\text{Aura}_{\text{conf}}$, 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 cryptographically interpret these functions.

Heintze and Riecke’s [6] 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 $\text{Aura}_{\text{conf}}$ ’s notion of dynamic authority which describes the program’s available keys.

Bibliography

- [1] A. Askarov, D. Hedin, A. Sabelfeld. Cryptographically masked information flows. In *ISAS '06*, LNCS. Seoul, Korea, 2006.
- [2] A. Askarov, S. Hunt, A. Sabelfeld, D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08*, pp. 333–348. 2008.
- [3] D. Dolev, A. Yao. On the security of public key protocols. In *IEEE Transactions on Information Theory*, 2(29), 1983.
- [4] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09*. 2009.

- [5] R. Goldblatt. Mathematical modal logic: a view of its evolution. In *J. of Applied Logic*, 1(5-6):309–392, 2003.
- [6] 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.
- [7] 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.
- [8] L. Jia, D. Walker. Modal proofs as distributed programs (extended abstract). In *ESOP '04*. 2004. Full version Princeton Tech. Rep. TR-671-03.
- [9] L. Jia, S. Zdancewic. Encoding information flow in aura. In *PLAS '09*. 2009.
- [10] P. Laud, V. Vene. A type system for computationally secure information flow. In *FCT '05*, pp. 365–377. Lübeck, Germany, 2005.
- [11] J. M. Lucassen, D. K. Gifford. Polymorphic effect systems. In *POPL '88*. 1988.
- [12] T. Murphy, VII. *Modal Types for Mobile Code*. Ph.D. thesis, CMU, 2008.
- [13] T. Murphy, VII, K. Cray, R. Harper, F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS '04*. IEEE Press, 2004.
- [14] A. Myers, B. Liskov. Protecting privacy using the decentralized label model. In *ACM Trans. on Software Engineering and Methodology*, 9(4):410–442, 2000.
- [15] F. Pfenning, R. Davies. A judgmental reconstruction of modal logic. In *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, 2001.
- [16] D. K. Rappe. *Homomorphic Cryptosystems and Their Applications*. Ph.D. thesis, University of Dortmund, Germany, 2004.
- [17] A. Sabelfeld, A. C. Myers. Language-based information-flow security. In *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [18] G. Smith, R. Alpízar. Secure information flow with random assignment and encryption. In *FSME'06*, pp. 33–43. Alexandria, Virginia, USA, 2006.
- [19] E. Sumii, B. C. Pierce. A bisimulation for dynamic sealing. In *POPL '04*. 2004.
- [20] J.-P. Talpin, P. Jouvelot. The type and effect discipline. In *LICS '92*. 1992.
- [21] 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.
- [22] J. A. Vaughan, S. Zdancewic. A cryptographic decentralized label model. In *IEEE Security and Privacy*, pp. 192–206. Berkeley, California, 2007.
- [23] D. Volpano, G. Smith, C. Irvine. A sound type system for secure flow analysis. In *Journal of Computer Security*, 4(3):167–187, 1996.

Appendix

The following figure lists full versions of the most interesting typing rules for $\text{Aura}_{\text{conf}}$.

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

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

$$\frac{\Sigma; \mathcal{F}; W|E; \perp; \perp \vdash u_1 : k_1 \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \Sigma; \mathcal{F}; W|E; V; U_0 \vdash \diamond \quad \Sigma; \mathcal{F}; W|E, x : u_1 \mathbf{at} \perp; \perp; \perp \vdash u_2 : k_2}{\Sigma; \mathcal{F}; W|E; V; U \vdash (x : u_1) \rightarrow_{\{U_0\}} u_2 : k_2} \text{WF-TM-ARR}$$

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

$$\frac{\Sigma; \mathcal{F}; W|E; \perp; \perp \vdash u_1 : k_1 \quad \Sigma; \mathcal{F}; W|E; V; U_0 \vdash \diamond \quad \Sigma; \mathcal{F}; W|E, x : u_1 \mathbf{at} \perp; V; U_0 \vdash f : u_2 \quad \Sigma; \mathcal{F}; W|E; V; U \vdash (x : u_1) \rightarrow_{\{U_0\}} u_2 : k}{k \in \{\mathbf{Type}, \mathbf{Prop}\} \quad k_1 \in \{\mathbf{Type}, \mathbf{Prop}\} \vee u_1 \in \{\mathbf{Type}, \mathbf{Prop}\}} \text{WF-TM-ABS}$$

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

$$\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash e_1 \quad e_2 : \{x/e_2\}u}{\Sigma; \mathcal{F}; W|E; V; U \vdash e_1 \quad e_2 : \{x/e_2\}u} \text{WF-TM-APP}$$

$$\frac{\Sigma; \mathcal{F}; W|E; V_1; U \vdash e_1 : t_1 \quad \Sigma; \mathcal{F}; W|E; V; V_1 \vdash \diamond \quad \Sigma; \mathcal{F}; W|E; \perp; \perp \vdash t_1 : k \quad \Sigma; \mathcal{F}; W|E, x : t_1 \mathbf{at} V_1; V; U \vdash e_2 : t \quad k \in \{\mathbf{Type}, \mathbf{Prop}\} \quad V_1 \sqsubseteq V}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{let } x \mathbf{ at } V_1 = e_1 \mathbf{ in } e_2 : t} \text{WF-TM-LETAT}$$

$$\frac{\Sigma; \mathcal{F}; W|E; b; U \vdash e_1 : t_1 \mathbf{for } b \quad \Sigma; \mathcal{F}; W|E; b \vdash \diamond \quad \Sigma; \mathcal{F}; W|E; \perp; \perp \vdash t_1 : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; \perp; \perp \vdash t \mathbf{for } b : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; V \vdash \diamond \quad \Sigma; \mathcal{F}; W|E, x : t_1 \mathbf{at } b; b; b \vdash e_2 : t \mathbf{for } b \quad b \sqsubseteq V}{\Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{bind}_f x = e_1 \mathbf{ in } e_2 \mathbf{ as } (t \mathbf{for } b) : t \mathbf{for } b} \text{WF-TM-FORBIND}$$

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

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

Fig. 7: Selected typing rules for $\text{Aura}_{\text{conf}}$

$$\begin{array}{c}
\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; \perp; \perp \vdash P : \mathbf{Prop} \quad \Sigma; \mathcal{F}; W|E; \perp; \perp \vdash a : \mathbf{prin} \quad a \sqsubseteq U \quad \text{value } a}{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \mathbf{say } a P : \mathbf{pf} (a \text{ says } P)} \text{WF-TM-SAY} \\
\\
\frac{\Sigma; \mathcal{F}; W|E; V; \perp \vdash t : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; V; \perp \vdash a : \mathbf{prin} \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \text{value } t \quad \text{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} \\
\\
\frac{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad E \vdash 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; \perp; \perp \vdash t \mathbf{for } a : \mathbf{Type} \quad a \sqsubseteq V}{\Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \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 \text{value } e}{\Sigma; \mathcal{F}; W|E; V; U \vdash e \mathbf{isat 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 \text{converts } E t_1 t_2 \text{ at } V}{\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; \perp; \perp \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{\Sigma; \mathcal{F}; W|; b; b \vdash e : t \quad \Sigma; \mathcal{F}; W|; \perp; \perp \vdash t \mathbf{for } b : \mathbf{Type} \quad \Sigma; \mathcal{F}; W|E; V; U \vdash \diamond \quad \Sigma; \mathcal{F}; W|; b \vdash \diamond \quad b \sqsubseteq V \quad b \sqsubseteq W}{\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{\Sigma; \mathcal{F}; W|E; V; U \vdash p : \mathbf{pf} (a \text{ says } (e \mathbf{isat for } b)) \quad \Sigma; \mathcal{F}; W|E; V; U \vdash e : \mathbf{bits} \quad \Sigma; \mathcal{F}; W|E; \perp; \perp \vdash t \mathbf{for } b : \mathbf{Type} \quad \text{value } e}{\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}
\end{array}$$

Fig. 7: Selected typing rules for $\text{Aura}_{\text{conf}}$ (cont.)