NOTE TO REVIEWERS

This technical report is an extended version of our submission to the CPP 2012 conference. It contains more detailed explanations and more examples, as well as an explicit proof of theorem 4 in the appendix.

The authors

DKAL*: Constructing Executable Specifications of Authorization Protocols

Jean-Baptiste Jeannin[1] Guido de Caso[2] Juan Chen Yuri Gurevich Prasad Naldurg Nikhil Swamy
Cornell University[1] Universidad de Buenos Aires[2] Microsoft Research

***Abstract*—Many prior trust management frameworks provide authorization logics for specifying policies based on distributed trust. However, to implement a security protocol using these frameworks, one usually resorts to a general-purpose programming language. When reasoning about the security of the entire system, one must study not only policies in the authorization logic but also hard-to-analyze implementation code.**

This paper proposes DKAL*, a language for constructing executable specifications of authorization protocols. Protocol and policy designers can use DKAL*'s authorization logic for expressing distributed trust relationships, and its small rule-based programming language to describe the message sequence of a protocol. Importantly, many low-level details of the protocol (e.g., marshaling formats or management of state consistency) are left abstract in DKAL*, but sufficient details must be provided in order for the protocol to be executable.

We formalize the semantics of DKAL*, giving it both an operational semantics and a type system. We prove various properties of DKAL*, including type soundness and a decidability property for its underlying logic. We also present an interpreter for DKAL*, mechanically verified for correctness and security. We evaluate our work experimentally on several examples.

Using our semantics, DKAL* programs can be analyzed for various protocol-specific properties of interest. Using our interpreter, programmers obtain an executable version of their protocol which can readily be tested and then deployed.

I. Introduction

Despite many years of successful research in protocol design, federated cloud services continue to be plagued by flaws in the design and implementation of critical authorization protocols. For example, recent work by Wang et al. (2011) reveals authorization errors in a variety of federated online payment services. Among other reasons, Wang et al. argue that the ad hoc implementation of such services obscures the delicate protocols on which they are based, making the design and implementation of these protocols difficult to analyze for vulnerabilities. We propose to address such difficulties by providing a domain-specific language to concisely specify authorization protocols so that the protocol design is *evident* (and suitable for security analysis, both formal and informal), and *executable*.

Our work derives from the long line of work on trust management starting from the ABLP calculus (Abadi et al. 1991) and ranging to more recent efforts like DKAL (Gurevich and Neeman 2008). From these, we borrow the insight that an appropriate authorization logic is needed for specifying complex, dynamic trust relationships that arise in typical authorization scenarios. However, we contend that a practical language for distributed authorization must go beyond logic-related aspects of trust and provide a way to describe the concrete message flows of a protocol.

To illustrate, consider the following simple scenario. An online retailer W wishes to use a third-party payment provider P (e.g., PayPal) to process payments on her website. Many tools exist to help the retailer build her website to process such payments. However, as Wang et al. report, these tools are often buggy, with no clear specification of the protocol they implement.

Informally, we would like to start by specifying that the retailer W trusts P to process payments. Prior authorization logics allow such trust relationships to be expressed concisely; e.g., in infon logic (Gurevich and Neeman 2008), one might write a policy for W of the form below:

$\forall$c, oid, n. P said Paid(c, oid, n) $\Longrightarrow$Paid(c, oid, n)

expressing that W is willing to conclude that a principal c paid n for order oid, if P said so.

However, the means to arrive at a specific authorization protocol based on this trust relationship alone is unclear. Even a simple protocol involves several rounds of communication between a customer C, the website W, and the payment provider P. For example, the protocol illustrated in Figure 1 involves five steps: (1) a customer C requests to purchase some item i for a price n; (2) the retailer W requests C to provide a certificate from PayPal (P) authorizing C's payment; (3) C forwards the payment request to P; (4) P authorizes the payment from C to W and issues a certificate confirming the payment; (5) W, relying on a trust relationship with P, concludes that the payment has indeed been processed and ends the protocol by returning a confirmation to C.

Prior work on trust management ignores the specifics of such protocols. Typically, one is expected to implement a protocol of one's choice in a general-purpose programming language, where one can make queries to a trust management engine to determine if access to a protected resource is to be permitted or not; e.g., this is the methodology of SecPAL (Becker et al. 2010) and many other tools. While this approach provides great flexibility, it also leaves the design of the authorization protocol unclear, and opens the door to vulnerabilities due to improper protocol design or other, more mundane programming errors.

A. Specifying authorization protocols in DKAL*

To address these problems, we propose DKAL*, a domain-specific language for executable specifications of authorization

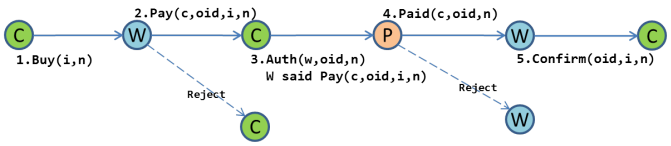

Fig. 1: A simple protocol for processing online payments

protocols. We formalize the semantics of DKAL* and implement a verified interpreter for it using F* (Swamy et al. 2011), a verification-oriented dialect of ML. DKAL* programs include three conceptual components. First, we have the quantified primal infon logic (QPIL) for expressing distributed trust relationships. Next, we have a small rule-based programming language to describe the message flow of a protocol. Finally, DKAL* programs may embed expressions from F*, the host language of our interpreter—one can use this facility to interface with the external computational fabric, e.g., to evaluate arithmetic expressions, connect to databases, etc. Thus, having designed a protocol in DKAL*, one may readily obtain a full-fledged secure implementation.

Figure 2 shows an example of DKAL* code, a policy specified by each of the three principals in our online retail scenario. This will be our running example throughout the paper. DKAL* programs are a collection of rules, each of which can be thought of as handlers that cause specific *actions* to occur in response to events that meet certain *conditions*. Actions include sending messages (send), forwarding messages (fwd), a logging facility (log), generating fresh identifiers (with fresh), and introducing new information (learn) to the principal's QPIL knowledge base, permitting new facts to be derived. Conditions have two forms: when e is satisfied if the principal has received a message that matches the pattern constructed by the term e; the condition if e is satisfied if the proposition constructed by the term e is derivable in QPIL. Terms include the form **eval**(e), where e is an F* expression evaluated by the interpreter; variables (lower-case identifiers, e.g., i, n); constants (upper-case letters, e.g., W, P); and constructed terms (Buy(i, n), etc.). Variables are (implicitly) universally quantified at the beginning of rules unless they have explicit bindings.

Rules C_1, W_2, C_3, P_4, W_5 correspond to the steps (1)-(5) in Figure 1: (1) Rule C_1 initiates the protocol in response to an external event Click(i,n) issued by the customer C, by sending a message C said Buy(i,n) to the website W. C also logs a message Init(W,i,n) to indicate that she has initiated the transaction. (2) After receiving the message C said Buy(i,n), W applies rule W_2 to request a payment certificate. W checks the price of the item by calling an F* function (checkPrice), and sends a message W said Pay(c,oid,i,n) to C requesting payment for a new order oid containing i. W also logs a message to keep track of the transaction currently underway. (3) Once C gets such a message from W and checks her log for the message Init(W,i,n), she applies rule C_3 to forward a payment request to P. (4) Rule P_4 is P's policy that authorizes the payment by sending a message P said Paid(c,oid,n) to the website W (after checking and updating C's balance, using F*). (5) If

```
(* CUSTOMER's (C) policy *)
C1:
when C said Click(i, n) then
send W (C said Buy(i, n))
log C said Init(W, i, n)

C3:
when C said Init(w, i, n)
when w said Pay(C, oid, i, n) as m1
then send P (C said Auth(w, oid, n))
       fwd P m1

(* PAYPAL's (P) policy *)
P4 :
when c said Auth(w, oid, n)
when w said Pay(c, oid, i, n)
if eval(checkBalance "c" "n") then
send w (P said Paid(c, oid, n))
where checkBalance = (* F* code *)
```

```
(* Website's (W) policy *)
W2:
when c said Buy(i, n)
if eval(checkPrice("i","n")) then
with fresh oid
send c (W said Pay(c, oid, i, n))
log W said Pay(c, oid, i, n)
where checkPrice = (* F* code *)

W5:
when W said Pay(c,oid,i,n)
if Paid(c, oid, n) then
send c (W said Confirm(oid,i,n))

W6:
when P said x as i then
learn i

W7:
learn ∀p,oid,n.
       P said Paid(p, oid, n)
  ⟹Paid(p, oid, n)
```

Fig. 2: A DKAL* policy implementing the online retail protocol

W receives a message P said Paid(c,oid,n), she uses her trust assumption in P (expressed in W_6 and W_7), and a decision procedure for QPIL to conclude that Paid(c,oid,n) is derivable, and sends a confirmation message to C (expressed in W_5).

B. Contributions

This paper makes several technical contributions.

***Design of* DKAL*.** We formalize the design of DKAL*. We analyze the central entailment relation of QPIL, and prove Theorem 3. We give an operational semantics and a type system for DKAL* and prove Theorem 4, and Theorem 5, which ensures that the execution of well-typed programs is insensitive to the order of rule evaluation. Our semantics provides the formal basis on which to analyze DKAL* policies
***A verified implementation in* F*.** We provide an interpreter for DKAL* in F*. We prove (mechanically checked by F*) that our interpreter soundly implements the formal semantics of DKAL*, including a verified implementation of a decision procedure for QPIL. Additionally, our interpreter includes verified implementations of a simple protocol based on public-key cryptography for establishing message authenticity. Thus, we obtain a mechanically checked security theorem, a correspondence property on traces that ensures that a message recipient only accepts authentic messages. Using refinement type checking, we show how to securely embed and evaluate F* terms within DKAL*, allowing a DKAL* protocol to easily and safely interface with its environment.
Experimental evaluation. We report on a preliminary experimental evaluation of DKAL* by using it to develop a suite of 7 examples. While we expect to conduct further experiments in the future, our experience indicates that DKAL* specifications can be terse, conveying the important high-level aspects of a distributed security protocol, while leaving many of the low-level details necessary to produce an executable implementation to our verified interpreter.

Limitations. We note a few limitations of our current work,

which we aim to alleviate in the future. We do not address secrecy of communications. Our authenticity property applies to single communications only, rather than a session-like global property (Bhargavan et al. 2009). We prove that there exists a complete decision procedure for QPIL entailment, but we have not proven the completeness of the decision procedure in our implementation yet (we prove only its soundness). Finally, we take DKAL* programs as specifications. As such, it is certainly possible to write entirely insecure protocols in DKAL*. We leave the analysis of DKAL* for protocol-specific security properties to future work. However, by making the protocol design evident, and by employing F* as our implementation vehicle, we are optimistic that we can carry out property checking using F*'s refinement type system.

Organization of the paper. Our presentation begins by formalizing DKAL* independently of its implementation language F*. Section II defines QPIL, the authorization logic that underlies DKAL*. Section III presents the design of the rule-based programming language in DKAL*, including its operational semantics and type system, and develops the metatheory of DKAL*. We discuss the online retailer example through the course of this section to illustrate various aspects of the design. Section IV reviews F* and shows how we use it to model the semantics of DKAL*. We discuss our implementation strategy, including our approach for embedding F* terms in DKAL* using **eval**. We present our verification results, including the use of verified cryptography to establish the authenticity of communications, and a verified implementation of a QPIL decision procedure. We also discuss our experimental evaluation. Section V discusses related work, and Section VI concludes.

II. QPIL: Quantified Primal Infon Logic

We start by reviewing QPIL, Gurevich and Neeman's primal infon logic with quantifiers. Our formulation of QPIL differs from prior works in that we pay particularly close attention to binders, α-equivalence, and the uniqueness of names—all to facilitate a mechanically verified implementation of decision procedures for the logic. We use QPIL as the authorization logic underlying DKAL*, although DKAL* is designed to be parametric in the choice of the logic—other logics could just as well be used with DKAL*, so long as they satisfy certain weak admissibility constraints. We view QPIL as perhaps the simplest modal logic that could be useful for common authorization protocols.

A. Syntax

QPIL is designed around two basic concepts. The first is *infon*, a formula which represents a unit of information (which may be learned, communicated, etc.). The second is *evidence*—an infon i may be accompanied by a term t which serves as evidence for the validity of i. The form of evidence used in QPIL is left abstract; e.g., an infon i may be accompanied by a digital signature to serve as evidence that it was communicated by a principal p; or, it may represent a proof tree recording a derivation of i from some set of hypotheses according to the inference rules of the logic.

The display below shows the syntax of QPIL. Predicates Q and constants c are subscripted with their types, although we elide the subscripts when the types are unimportant. Types include booleans and integers (and other common types in the implementation), principals, as well as a distinguished type for evidence terms, ev. The terms in the logic include variables x, y, z and constants c (tagged) with their types. Later (Section IV) we add embedded F* terms to the term language.

Syntax of QPIL

Meta-variables: x, y, z variables; $Q_{\bar\tau}$ predicates; c_τ constants

type	τ	::=	$\mathsf{bool} \mid \mathsf{int} \mid \mathsf{prin} \mid \mathsf{ev}$
term	p, t	::=	$x \mid c_\tau$
infon	i, j	::=	$\top \mid Q_{\bar\tau}\ \bar t \mid i \wedge j \mid i \Rightarrow j \mid p\ \mathsf{said}\ i \mid \mathsf{Ev}\ t\ i$
q. infon	ι	::=	$i \mid \mathsf{Ev}\ t\ \iota \mid \forall \overline{x{:}\tau}.i$
type ctxt.	Γ	::=	$\cdot \mid x{:}\tau \mid \Gamma, \Gamma$
infon set	M, K	::=	$\bar\iota$

Infons i include the true infon $\top$; the application of a predicate symbol Q to a sequence of terms t; a conjunction form $i \wedge j$; an implication form $i \Rightarrow j$; the form p said i, which is the modal operator of speech applied to an infon; and finally *justified infons*, Ev t i, which associates an evidence term t with an infon i; note that when a principal sends Ev t i, he is merely asserting that t is evidence for i, and the receiver of the message, if he desires so, can check t. An example of an authorization is the infon: Bob said CanRead(Alice,"file.txt"). QPIL includes quantified infons ι, where an infon i may be preceded by a sequence of binders for universally quantified variables $\overline{x{:}\tau}$. The use of quantifiers allows for more general and flexible policies such as: ∀(x:prin). Bob said Trusted(x) ⟹ CanRead(x, "file.txt"). Quantified infons may also be justified by associating them with evidence using Ev t ι. Unless explicitly mentioned, we blur the distinction between quantified infons and infons.

B. Typing

QPIL has three typing judgments (shown below): $\Gamma \vdash \iota$ for quantified infons; $\Gamma \vdash i$ for infons; and $\Gamma \vdash t : \tau$ for terms, where the typing context Γ maps variables to their types. Intuitively, $\Gamma \vdash \iota$ ensures that the variables of ι appear in Γ at suitable types. The typing judgments also rely on a well-formedness judgment for the context: we write Γ ok for an environment where no variable appears twice, and $\Gamma(\tau)$ for the type τ such that Γ contains $x : \tau$.

Typing terms and infons

$$\frac{\Gamma\ \mathsf{ok}}{\Gamma \vdash c_\tau : \tau} \qquad \frac{\Gamma\ \mathsf{ok}}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma\ \mathsf{ok}}{\Gamma \vdash \top} \qquad \frac{\Gamma, \overline{x{:}\tau} \vdash i}{\Gamma \vdash \forall \overline{x{:}\tau}.i}$$

$$\frac{\forall i.\Gamma \vdash t_i : \tau_i}{\Gamma \vdash Q_{\bar\tau}\ \bar t} \qquad \frac{\Gamma \vdash i \quad \Gamma \vdash j}{\Gamma \vdash i \wedge j} \qquad \frac{\Gamma \vdash i \quad \Gamma \vdash j}{\Gamma \vdash i \Rightarrow j}$$

$$\frac{\Gamma \vdash p : \mathsf{prin} \quad \Gamma \vdash i}{\Gamma \vdash p\ \mathsf{said}\ i} \qquad \frac{\Gamma \vdash t : \mathsf{ev} \quad \Gamma \vdash \iota}{\Gamma \vdash \mathsf{Ev}\ t\ \iota}$$

The typing rules for terms are straightforward—constants are typed using their subscripts, and variables by the typing context. The rules for infons are straightforward, with only one subtle point to mention. The last rule is overloaded to apply to both justified infons and justified quantified infons.

C. Entailment

We define an entailment relation for QPIL, a Hilbert-style calculus defining the inference rules of the logic. Our formulation relies on the notion of a prefix π, a possibly empty sequence of terms $\bar{t}$ of type prin. We write $\pi\ i$ to mean i when π is empty, or t said $(\pi'\ i)$ when $\pi = t, \pi'$. The calculus includes two relations, $K;\Gamma \vDash \iota$ for quantified infons and $K;\Gamma \vDash i$ for infons. The context in each of these relations includes an *infostrate*, K, a set of infons, representing a principal's knowledge, and a typing context Γ. We write K ok for an infostrate where for each $\iota \in K$ we have $\cdot \vdash \iota$, i.e., K is a set of well-typed closed infons. We write $K;\Gamma$ ok for (K ok and Γ ok).

Entailment relations: $K;\Gamma \vDash \iota$ **and** $K;\Gamma \vDash i$

$$\frac{K;\Gamma\ \mathsf{ok} \quad \Gamma \vdash \pi\ \top}{K;\Gamma \vDash \pi\ \top}\ \text{T} \qquad \frac{K;\Gamma\ \mathsf{ok} \quad \iota \in K \quad \iota \equiv_\alpha \iota' \quad \Gamma \vdash \iota'}{K;\Gamma \vDash \iota'}\ \text{Hyp-K}$$

$$\frac{K;\Gamma \vDash \pi\ i \quad K;\Gamma \vDash \pi\ j}{K;\Gamma \vDash \pi(i \wedge j)}\ \wedge\text{-I} \qquad \frac{K;\Gamma \vDash \pi(i \wedge j)}{K;\Gamma \vDash \pi\ i}\ \wedge\text{-E1}$$

$$\frac{K;\Gamma \vDash \pi(i \wedge j)}{K;\Gamma \vDash \pi\ j}\ \wedge\text{-E2} \qquad \frac{\Gamma \vdash \pi\ i \quad K;\Gamma \vDash \pi\ j}{K;\Gamma \vDash \pi(i \Rightarrow j)}\ \Rightarrow\text{-WI}$$

$$\frac{K;\Gamma \vDash \pi(i \Rightarrow j) \quad K;\Gamma \vDash \pi\ i}{K;\Gamma \vDash \pi\ j}\ \Rightarrow\text{-E} \qquad \frac{K;\Gamma \vDash \pi\ (\text{Ev}\ t\ \iota)}{K;\Gamma \vDash \pi\ \iota}\ \text{Ev-E}$$

$$\frac{K;\Gamma, \overline{x{:}\tau} \vDash i}{K;\Gamma \vDash \forall \overline{x{:}\tau}.i}\ \text{Q-I} \qquad \frac{K;\Gamma \vDash \forall \overline{x{:}\tau}.j \quad \forall i.\Gamma \vdash t_i : \tau_i}{K;\Gamma \vDash j[\overline{t/x}]}\ \text{Q-E}$$

The inference rule (T) allows well-typed infon $\pi\top$ to be derived from any well-formed context. The rule (Hyp-K) allows using infostrate hypotheses $\iota \in K$, but only after they have been suitably α-converted to ι', so as to avoid the bound names of ι' clashing with the names in the context. The premise $\Gamma \vdash \iota'$ guarantees no name clashing. The definition of alpha equivalence, $\iota \equiv_\alpha \iota'$ is standard—we elide it due to space constraints.

The rule ($\wedge$-I) is an introduction rule for conjunctions, with ($\wedge$-E1) and ($\wedge$-E2) the corresponding elimination rules, showing the modality π distributing over the conjuncts.

The rule ($\Rightarrow$-WI) is the weak introduction rule for implications, and the rule ($\Rightarrow$-E) is the usual elimination form. The weak form of implication is characteristic of primal infon logic—it allows deriving $\pi(i \Rightarrow j)$ only if $\pi\ j$ can already be derived. This may seem pointless, except for two reasons: (1) this weak form of implication lends itself to an efficient linear-time decision procedure, at least for the propositional primal infon logic; and (2) in the case of authorization, a principal may know the conclusion $\pi\ j$, but may be willing to share only a weaker part $\pi\ (i \Rightarrow j)$ with another principal. As such, we conjecture that the weak implication form makes primal infon logic possibly the simplest logic that one might use for distributed authorization.

The rule (Ev-E) is the elimination form for evidence—note that the only way of introducing justified infons is by hypothesis or by elimination of other infon forms. Finally, we have the rules (Q-I) and (Q-E) for introducing and eliminating quantifiers.

With these definitions, we can state and prove our first lemma, namely that entailment derives only well-typed infons.

Lemma 1 (*Entailment is well-typed):* For all K, Γ, ι, if $K;\Gamma \vDash \iota$ then $\Gamma \vdash \iota$.

Proof: By induction on the structure of entailment. ■

D. Decidability of QPIL

Finally, we show that there exists a complete decision procedure for QPIL entailment. Due to space constraints, we are unable to provide details—a companion technical report contains the full development (Jeannin et al. 2012). We give a flavor of the main result here.

Gurevich and Neeman (2008) present a linear-time algorithm for the multiple derivability problem for propositional primal infon logic (PIL). Given hypotheses K and queries i, the algorithm finds which queries are derivable from K, i.e., it computes the entailment $K;\cdot \vDash i$. (Note, in the propositional case, the variable context Γ is always empty.) This algorithm is linear in the size of the input sequence K, i. It relies on a *sub-formula* property of PIL entailment, namely that the derivation $K;\cdot \vDash i$ only uses the sub-formulas of K, i. The algorithm begins by computing the set of sub-formulas of K, i. Each subformula also indexes other subformulas that may be involved in the application of any one-step derivation rules. These derivations are iteratively computed until a fixed-point is reached.

The completeness of query derivation (multiple or otherwise) for QPIL has not been studied before. We show an analog of the sub-formula property for QPIL, namely that every QPIL derivation respects *local substitutions*. If a substitution rule is applied in any derivation, it is only applied to variables occurring in formulas from what we call the extended locality (XL) set of K and query ι. We define a judgment $K;\Gamma \vDash^{XL} \iota$, representing a derivation that uses only variables in the XL set. The following lemma establishes that for any QPIL derivation, there exists a derivation using only variables in the XL set, and vice versa.

Lemma 2 (*Extended locality):* Let K be a set of QPIL infons, Γ a variable context, and ι be a QPIL query. Then $K;\Gamma \vDash \iota \Leftrightarrow K;\Gamma \vDash^{XL} \iota$.

Using this lemma, we can further show that QPIL has a complete decision procedure, using an argument similar to the proof of the subformula property for PIL.

Theorem 3 (*A complete decision procedure for* QPIL*):* Let K be a set of QPIL infons, Γ a variable context, and ι be a QPIL query. Then, there exists an algorithm to decide whether $K;\Gamma \vDash \iota$ is derivable.

Proof: (Sketch) From the extended locality lemma, we only need to look at theory variables that occur in K and ι. We can extend the algorithm for multiple derivability in propositional PIL as follows: For finite domains, the original algorithm can be used as a black-box by instantiating all variables that occur to all theory constants in formulas from the XL set. The proof of completeness for QPIL can therefore rely on the algorithm for propositional PIL. ■

Note, while the quantifier instantiation strategy in our proof may cause an exponential blowup in the size of the input, in practice, this blowup can be controlled by relevant policy scoping and intelligent search strategies. A goal-directed search can be implemented starting with constants in the query ι. Further, if ι contains only constants (which is usually the case when it represents a concrete access control request), we observe that the substitution rule is only applied to these constants.

Finally, we remark that while the existence of a complete decision procedure for QPIL is a useful property, the rest of DKAL* is designed so that it may also be used with other, more powerful authorization logics, e.g., the full infon logic with a more standard form of implication introduction.

III. The Design and Semantics of DKAL*

We now define DKAL*, a rule-based language for specifying the communication patterns in an authorization protocol. DKAL* artifacts are, simultaneously, *programs*, *policies* and *specifications*—we use the terms interchangeably, unless explicitly noted otherwise. This section introduces DKAL*'s syntax and semantics, relying on our online retail scenario for illustrative examples.

A. Syntax of DKAL*

The display below shows the syntax of DKAL*. A program R is a finite set of rules, each of the form $(C \text{ then } A)$. The semantics of DKAL* executes a program by evaluating the guards C of each rule against a principal's local configuration, and applying the actions A of only those rules whose guards are satisfied. The local configuration P of a principal p is a triple (K, M, R). It includes (1) an *infostrate*, K, which is a monotonically increasing set of infons, representing p's knowledge; (2) a *message store*, M (also a set of infons), which p may use to retain messages that it receives; and, (3) the *program* R itself. The global configuration G is the parallel composition of configurations (p, P), one for each principal p. We give a message-passsing semantics for DKAL* in which the reduction of a local configuration P causes infons to be sent to other principals.

Syntax of DKAL*

program	R	::=	C then $A \mid R\ R \mid \cdot$
local configuration	P	::=	(K, M, R)
global configuration	G	::=	$(p, P) \mid G \parallel G$
guards	C	::=	upon ι as $x \mid$ if $\iota \mid C\ C \mid \cdot$
actions	A	::=	send $p\ \iota \mid$ fwd $p\ \iota \mid$ drop ι
		$\mid$	learn $\iota \mid$ with fresh $x\ A \mid A\ A$
infon	i	::=	$\ldots \mid x$
typing context	Γ	::=	$\ldots \mid x$:infon $\mid x$:qinfon

Guards come in two flavors. The guard (upon ι as x) is a pattern which checks whether a message matching ι is present in the principal's message store M and binds the message to x if matched. We extend the syntax of infons i so that they may contain pattern variables x. Evaluating an upon condition requires computing a substitution σ for the pattern variables such that $\sigma\ \iota$ is in the message store M. In order to ensure that pattern variables are properly used, we extend our syntax of typing environments Γ to include bindings for variables typed as infons and quantified infons (qinfon).

Guards also include boolean conditions of the form (if ι). Evaluating this guard involves a call to a decision procedure of QPIL to check that the infon ι is derivable from the principal's knowledge K. If derivable, the actions of the rule are applied; otherwise the rule is inactive. This kind of guard does not bind pattern variables.

Actions include (send $p\ \iota$), which sends ι to p authenticated by the sender; (fwd $p\ \iota$), which forwards a previously received message to p; (drop ι), which deletes a message from M; (learn ι), which adds an infon to the knowledge K; and, finally, a construct (with fresh $x\ A$) to generate fresh identifiers.

We view the language formalized here as the core of DKAL*. When writing examples we use some syntactic sugar, which can easily be macro-expanded into the core. In particular, the when and log constructs (used in Figure 2) are desugared as shown below, where Self is a principal constant for the local principal. Note, we do not formalize the use of **eval** until Section IV.

$$\begin{aligned} \text{when } \iota \text{ then } A &= \text{upon Ev } x\ \iota \text{ as } m \text{ then } (A, \text{drop } m) \\ &\quad \text{for fresh } x \text{ and } m \\ \text{log } \iota &= \text{send Self } \iota \end{aligned}$$

B. Operational semantics of DKAL*

The operational semantics of DKAL*, deriving from semantics of ASMs, are carefully set up to ensure a few properties. We discuss these properties informally here, motivating various elements of the design—we formalize these properties in the metatheory study of Section III-D.

State consistency. We desire a semantics with a consistent notion of state updates. To achieve this, we have a message passing semantics for global configurations. But, the reduction of each principal's configuration P is given using a big-step reduction in which all applicable actions from the rules in P are computed atomically, with respect to an unchanging local state. Big steps of local evaluation are interleaved with messages being exchanged among the principals, modifying their local states.

Determinism. We aim to ensure that the semantics of a program is independent of the order of execution of the rules in a program R. We achieve this by evaluating the set of actions computed from a set of rules in a canonical order.

Authenticity. DKAL* semantics takes into consideration the possible presence of network adversaries, and imperfect message delivery—messages may not reach their intended recipients. The reduction of global configurations models this by allowing any principal to receive any message, even if

the message is meant for someone else. Our implementation guarantees message authenticity, as developed in Section IV-D.

We begin by presenting the big-step evaluation of local configurations, $P \Downarrow_p A$, where a local configuration P for a principal p evaluates to a set of actions A. The rule (Ev) picks a rule C then A from the rule set and evaluates its guard C. Guard evaluation produces a set of substitutions $\bar{\sigma} = \{\sigma_1, \ldots, \sigma_n\}$ of the free variables in C such that the conditions $\sigma_i C$ are satisfied. The actions $[\![\sigma_i\ A]\!]$ are added to the actions computed from the evaluation of the other rules in the program. Here, the function $[\![A]\!]$ interprets a set of actions A by introducing fresh integer constants in the actions A, as required by the (with fresh x A) construct.

Local rule evaluation: $P \Downarrow_p A$

$$\text{Ev}\ \frac{(K, M, (R_1, R_2)) \Downarrow_p A' \quad \mathsf{holds}_p\ K\ M\ C = \overline{\sigma}}{(K, M, (R_1, (C\ \mathsf{then}\ A), R_2)) \Downarrow_p A' \cup_i [\![\sigma_i A]\!]}$$

$$\text{EvEmp}\ \frac{}{(K, M, \cdot) \Downarrow_p \{\}}$$

$$\begin{array}{l}
[\![\cdot]\!] : A \to A \\
[\![A]\!] = A \ \text{ when (with fresh } x\ A') \notin A \\
[\![A, \text{with fresh } x\ A']\!] = [\![A]\!], [\![A'[c_{\mathsf{int}}/x]]\!] \ \text{ for } c \text{ fresh} \\
\mathsf{holds} : p \times K \times M \times C \to 2^{\sigma} \\
\mathsf{holds}_p\ K\ M\ (\text{upon } \iota \text{ as } x) = \{(\sigma, x \mapsto \sigma\iota) \mid \\
\qquad \sigma\iota \in M \wedge \vdash \sigma\iota \wedge \mathrm{dom}\,\sigma = \mathrm{FV}(\iota)\} \\
\mathsf{holds}_p\ K\ M\ (\text{if } \iota) = \{id \mid K; \cdot \vDash \iota\} \\
\mathsf{holds}_p\ K\ M\ \cdot = \{id\} \\
\mathsf{holds}_p\ K\ M\ (C_1, C_2) = \{(\sigma_2 \circ \sigma_1) \mid \\
\quad \sigma_1 \in \mathsf{holds}_p\ K\ M\ C_1 \ \wedge\ \sigma_2 \in \mathsf{holds}_p\ K\ M\ (\sigma_1\ C_2)\}
\end{array}$$

The evaluation of guards is given by the function $\mathsf{holds}_p\ K\ M\ C$, which computes a set of substitutions. Evaluation of multiple guards involves composing the substitutions returned by the evaluation of each guard.

Evaluation of an (upon ι as x) guard returns every substitution σ such that a well-typed message $\sigma\iota$ can be found in the store M. Our verified implementation ensures that messages that match patterns are always properly justified, should they contain any evidence.

For (if ι), we require that the infon ι be derivable from the hypotheses in the infostrate K. Note that, unlike for the evaluation of (upon ι as x), the semantics requires the infon ι to be a closed term for rule evaluation to succeed. The reason is that we aim to enforce a deterministic semantics. An if-guard with a free variable may be handled by a decision procedure that picks an arbitrary witness for the variable, but that leads to non-determinism. Alternatively, one might give a semantics in which the decision procedure is required to return *all* possible satisfying assignments of the free variable. However, that is problematic, both because the number of assignments may be very large, and because this may require the use of a complete decision procedure for the logic—for a logic more expressive than QPIL, such a procedure may not exist.

We now define $G \longrightarrow G'$, a small-step reduction relation for global configurations. The single rule in the semantics (GoP) picks a principal p and evaluates the rules of p to obtain a set of actions A, and then applies these actions atomically to the configuration G. In order to ensure that the effect of applying the actions is independent of the order of evaluation of the rules, we require that all the (drop i) actions in A precede all the other actions. The ordering among multiple drop actions, and the ordering among all other actions is immaterial. For this, we define a unary operator on actions, $\mathsf{order}(A)$, and use it to reorder a set of actions A according to a partial order in which all the (drop ι) actions come first.

The definition of $\mathsf{app}(G, p, A)$ applies a set of actions A according to this partial order. We use $\mathsf{app1}(G, p, A)$ in the base cases to apply a single action. As explained before, drop ι and learn ι only affect the local principal p's state, removing a message from M, and adding an infon to K, respectively. The action fwd q ι adds a message to a principal's message store. Finally, the action send q ι adds a justified infon Ev t ι to a principal's message store, i.e., the sent message is in a form that can be authenticated by q. Note that, when p sends or forwards a message and to model the network imperfectness, the actual recipient q' may not be the intended principal q.

Reduction semantics of global configurations: $G \longrightarrow G'$

$$\text{GoP}\ \frac{P \Downarrow_p A}{G_1 \parallel (p, P) \parallel G_2 \ \longrightarrow\ \mathsf{app}\ (G_1 \parallel (p, P) \parallel G_2)\ p\ (\mathsf{order}(A))}$$

$$\begin{array}{l}
\mathsf{order} : A \to A \\
\mathsf{order}(A) = A_1, A_2 \text{ where } A_1 = \{\text{drop } \iota | \text{drop } \iota \in A\} \text{ and } A_2 = A \setminus A_1 \\
\mathsf{app} : G \to p \to A \to G \\
\mathsf{app}\ G\ p\ \cdot = G \\
\mathsf{app}\ (G_1 \parallel G \parallel G_2)\ p\ A = G_1 \parallel (\mathsf{app1}\ G\ p\ A) \parallel G_2 \\
\mathsf{app}\ G\ p\ (A, A') = \text{let } G' = \mathsf{app}\ G\ p\ A \text{ in} \\
\qquad \text{let } G'' = \mathsf{app}\ G'\ p\ A' \text{ in} \\
\qquad G'' \\
\mathsf{app1} : (p \times P) \to p \to A \to (p \times P) \\
\mathsf{app1}\ (p, (K, M, R))\ p\ (\text{drop } \iota) = (p, (K, (M \setminus \{\iota\}), R)) \\
\mathsf{app1}\ (p, (K, M, R))\ p\ (\text{learn } \iota) = (p, ((K, \iota), M, R)) \\
\mathsf{app1}\ (q', (K, M, R))\ p\ (\text{fwd } q\ \iota) = (q', (K, (M, \iota), R)) \\
\mathsf{app1}\ (q', (K, M, R))\ p\ (\text{send } q\ \iota) = (q', (K, (M, \text{Ev } t\ \iota), R))
\end{array}$$

C. Illustrating the semantics

We illustrate the operational semantics by considering the evaluation of some steps of the protocol specified in Figure 2.

Initialization. Consider an initial configuration in which the local message store M and infostrate K of C, W and P are all initially empty. In this configuration, the only rule which has a satisfiable guard is W_7 which adds the infon ∀p,oid,n. P said Paid(p, oid, n) ⟹Paid(p,oid,n) to W's infostrate. This rule is free to fire repeatedly, but it has no further action on W's infostrate.

Customer starts the protocol. Next, suppose due to some external event, e.g., the user clicking on a button, a message Click(ITEM17,10) is added to C's message store. This message allows the rule C_1 to fire, and in a single big step, the reduction of C's program produces the actions $A = \{$send W (C said Buy(ITEM17, 10)), send C (C said Init(W, ITEM17, 10)), drop Click(ITEM17, 10)$\}$. In applying A to the global configuration, we apply the drop action first, removing the Click message from C's store—since messages in the store serve as triggers for the rules, removing the message ensures that this rule fires only once in response to a user click. Next, by evaluating the first send action, we

send an infon Ev t (C said Buy(ITEM17, 10)) to W. At this level of our formalization, the semantics permits any t:ev to be communicated as evidence for the infon. As such, the semantics as presented here is clearly insecure—a principal p can freely forge an infon, e.g., q said IsAwesome(p), and communicate this to other principals, who have no means to tell this apart from authentic statements made by q. In Section IV-D, we remedy this problem by showing how to instantiate evidence using digital signatures and recovering message authenticity.

Finally, we evaluate the second send action, adding the Init message to C's own store, indicating that the protocol, from C's perspective, is in the initialization state.

Payment certificate. After P authorizes the payment and sends W a certificate P said Paid(C,OID,10) (suppose OID is a fresh constant generated for this transaction), rule W_6 is fired and P said Paid(C,OID,10) is put into W's infostrate. Now W_5 fires because a message W said Pay(C, OID, ITEM17, 10) is in W's message store, and because Paid(C, OID, 10) can be derived from the two infons ∀p,oid,n. P said Paid(p, oid, n) ⟹Paid(p,oid,n) and P said Paid(C,OID,10) in the infostrate.

D. A type system for DKAL⋆

We provide a type system to ensure that the reduction of DKAL⋆ programs is well-behaved, i.e., that configurations remain well-typed as reduction proceeds, and that that rule evaluation is deterministic.

Arbitrary DKAL⋆ programs may execute in undesirable ways. For example, an ill-scoped program may inject ill-typed infons into the infostrate, potentially allowing nonsensical terms to become derivable. Consider the example program upon (∀(p:principal). ALICE said x) as m then learn x. When evaluating this program against a message store M that contains the infon ∀(p:principal). ALICE said Good(p), the upon condition is satisfiable, with $\sigma = [\mathrm{x} \mapsto \mathrm{Good(p)}]$. However, applying the action σ(learn x) results in adding the term Good(p) to the infostrate, which is clearly ill-formed—the variable p has escaped its scope.

Our type system is designed to rule out this and other undesirable behaviors. Informally, our system ensures that pattern variables whose first binding occurence appears under a quantifier may not appear in a context outside the quantifier. The type system contains three main judgments, one each for programs ($\vdash R$), actions ($\Gamma \vdash A$) and guards ($\Gamma \vdash C : \Gamma'$). It also relies on an auxiliary judgment $\Gamma \vdash \iota \rightsquigarrow \Gamma'$, which infers the set of pattern variables in scope from a set of guards.

The next display starts by showing the judgment $\vdash R$, the typing rule for programs. Typing multiple rules involves typing each rule independently. In the base case, typing a rule C then A involves typing the guard C to produce a set of binders Γ for the pattern variables that appear in C, and using this context to type the actions A. Typing actions is given by the judgment $\Gamma \vdash A$ and is entirely straightforward.

The subtle elements of typing come into play when typing the guards. The judgment $\Gamma \vdash C : \Gamma'; \Gamma''$ informally states that in a context Γ the guard C binds the pattern variables Γ' which may be used in the remainder of a rule. Variables appearing in Γ'' appear in C under a ∀ in an upon statement, and as such cannot appear free later in the rule. This restriction is made to keep these variables from escaping their scope. By construction, Γ, Γ' and Γ'' have pairwise disjoint domains. The first two rules show that typing a sequence of guards involves typing each element and threading the set of pattern-bound variables through. Side conditions ensure that variables appearing under a ∀ statement do not appear later in the condition. When typing an upon-guard, we type the pattern ι, extracting a set of pattern variables Γ' from it using the relation $\Gamma \vdash \iota \rightsquigarrow \Gamma'; \Gamma''$, and add the variable x to the output set of pattern-bound variables. In contrast, when typing an if-guard, we require the infon ι to be well-typed, and do not allow it to bind any further pattern variables. Since C may bind pattern variables for (quantified) infons, we extend the judgment $\Gamma \vdash \iota$ with one more rule to allow typing such infon pattern variables.

Typing DKAL⋆ programs, actions, and guards

$$\boxed{\vdash R} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash R_1 \quad \vdash R_2}{\vdash R_1\ R_2} \qquad \frac{\vdash C : \Gamma; \Gamma' \quad \Gamma \vdash A}{\vdash C \text{ then } A}$$

$$\boxed{\Gamma \vdash A} \qquad \frac{\Gamma \vdash \iota}{\Gamma \vdash \text{drop } \iota} \qquad \frac{\Gamma \vdash \iota}{\Gamma \vdash \text{learn } \iota} \qquad \frac{\Gamma \vdash p : \mathsf{prin} \quad \Gamma \vdash \iota}{\Gamma \vdash \text{send } p\ \iota}$$

$$\frac{\Gamma \vdash p : \mathsf{prin} \quad \Gamma \vdash \iota}{\Gamma \vdash \text{fwd } p\ \iota} \qquad \frac{\Gamma, x{:}\mathsf{int} \vdash A}{\Gamma \vdash \text{with fresh } x\ A}$$

$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash A_1\ A_2}$$

$$\boxed{\Gamma \vdash C : \Gamma'; \Gamma''} \qquad \frac{}{\Gamma \vdash \cdot : \cdot; \cdot}$$

$$\frac{\Gamma \vdash C_1 : \Gamma_1; \Gamma_1' \qquad \Gamma, \Gamma_1 \vdash C_2 : \Gamma_2; \Gamma_2' \quad \mathrm{dom}\,\Gamma_1' \cap \mathrm{dom}\,\Gamma_2 = \emptyset \quad \mathrm{dom}\,\Gamma_1' \cap \mathrm{dom}\,\Gamma_2' = \emptyset}{\Gamma \vdash C_1\ C_2 : \Gamma_1, \Gamma_2; \Gamma_1', \Gamma_2'}$$

$$\frac{\Gamma \vdash \iota \rightsquigarrow \Gamma'; \Gamma'' \qquad x \notin \mathrm{dom}(\Gamma, \Gamma', \Gamma'')}{\Gamma \vdash \text{upon } \iota \text{ as } x : \Gamma', x{:}\mathsf{qinfon}; \Gamma''} \qquad \frac{\Gamma \vdash \iota}{\Gamma \vdash \text{if } \iota : \cdot}$$

$$\boxed{\Gamma \vdash \iota} \qquad \frac{\Gamma \text{ ok} \qquad \Gamma(x) = \mathsf{infon} \text{ or } \mathsf{qinfon}}{\Gamma \vdash x}$$

What remains now is the typing of patterns in upon-conditions, shown in the display below. In general, the judgments mirror the structure on infon typing, except these rules infer the types of the variables in a term based on their context, and impose additional constraints to ensure that pattern variables do not escape their scope. We start with the typing of ι-patterns, the judgment $\Gamma \vdash \iota \rightsquigarrow \Gamma'; \Gamma''$, indicating that the pattern ι binds the pattern variables Γ', and disallows later use of variables appearing in Γ''. By construction, Γ, Γ' and Γ'' have pairwise disjoint domains. The second rule shows that pattern variables that are bound under a quantifier may not be used in the remainder of a rule, since this (as our example illustrates) may cause variables to escape their scope. When typing a justified infon term appearing in a pattern, we require

as usual that the evidence t be typed as ev.

Typing patterns: $\Gamma \vdash \iota \rightsquigarrow \Gamma'; \Gamma''$, $\Gamma \vdash i \rightsquigarrow \Gamma'$ **and** $\Gamma \vdash t : \tau \rightsquigarrow \Gamma'$

$$\frac{\Gamma \vdash i \rightsquigarrow \Gamma'}{\Gamma \vdash i \rightsquigarrow \Gamma'; \cdot} \qquad \frac{\Gamma, \overline{x{:}\tau} \vdash i \rightsquigarrow \Gamma'}{\Gamma \vdash \forall \overline{x{:}\tau}.i \rightsquigarrow \cdot; \Gamma'}$$

$$\frac{\Gamma \vdash t : \mathsf{ev} \rightsquigarrow \Gamma' \quad \Gamma, \Gamma' \vdash \iota \rightsquigarrow \Gamma''; \Gamma'''}{\Gamma \vdash \mathsf{Ev}\ t\ \iota \rightsquigarrow \Gamma', \Gamma''; \Gamma'''}$$

$$\frac{\Gamma\ \mathsf{ok}}{\Gamma \vdash \top \rightsquigarrow \cdot} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma(x) = \mathsf{infon}}{\Gamma \vdash x \rightsquigarrow \cdot} \qquad \frac{\Gamma\ \mathsf{ok} \quad x \notin \mathrm{dom}\,\Gamma}{\Gamma \vdash x \rightsquigarrow x : \mathsf{infon}}$$

$$\frac{\Gamma \vdash p : \mathsf{prin} \rightsquigarrow \Gamma' \quad \Gamma, \Gamma' \vdash i \rightsquigarrow \Gamma''}{\Gamma \vdash p\ \mathsf{said}\ i \rightsquigarrow \Gamma', \Gamma''}$$

$$\frac{\Gamma\ \mathsf{ok} \quad \Gamma_0 = \Gamma \quad \forall i \in \{1 \ldots n\}.\Gamma_0 \ldots \Gamma_{i-1} \vdash t_i : \tau_i \rightsquigarrow \Gamma_i}{\Gamma \vdash Q_{\overline{\tau}} \overline{t} \rightsquigarrow \Gamma_1 \ldots \Gamma_n}$$

$$\frac{op \in \{\wedge, \Rightarrow\} \quad \Gamma \vdash i \rightsquigarrow \Gamma' \quad \Gamma, \Gamma' \vdash j \rightsquigarrow \Gamma''}{\Gamma \vdash i\ op\ j \rightsquigarrow \Gamma', \Gamma''}$$

$$\frac{\Gamma\ \mathsf{ok}}{\Gamma \vdash c_\tau : \tau \rightsquigarrow \cdot} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow \cdot} \qquad \frac{\Gamma\ \mathsf{ok} \quad x \notin \mathrm{dom}\,\Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x : \tau}$$

The typing of infon patterns $\Gamma \vdash i \rightsquigarrow \Gamma'$ is shown next. This is straightforward except for the variable case. Here, a variable x appearing where an infon is expected is typed as an infon, and we propagate a binder for x:infon in the output, if it does not already appear in the context.

Finally, we have the typing of terms in patterns, $\Gamma \vdash t : \tau \rightsquigarrow \Gamma'$, which is also easy, following the same behavior for variables as in the typing of infon variables. Note, in this judgment, the type τ is provided as input from the context.

In order to state and prove our main type-soundness theorem, we define (below) a notion of well-formed configurations. We write $G\ \mathsf{ok}_{\{p_1,\ldots,p_n\}}$ for a well-formed global configuration involving the principals $p_1 \ldots p_n$. This is a structural property on global configurations, requiring at most one local configuration per principal p_i. When the names of principals is immaterial we simply write $G\ \mathsf{ok}$

Well-formed configurations

$$\frac{K\ \mathsf{ok} \quad \vdash R}{(K, M, R)\ \mathsf{ok}} \qquad \frac{\cdot \vdash p : \mathsf{prin} \quad P\ \mathsf{ok}}{(p, P)\ \mathsf{ok}_{\{p\}}}$$

$$\frac{}{\cdot\ \mathsf{ok}_\emptyset} \qquad \frac{G_1\ \mathsf{ok}_{\overline{p_1}} \quad G_2\ \mathsf{ok}_{\overline{p_2}} \quad \overline{p_1} \cap \overline{p_2} = \emptyset}{G_1 \parallel G_2\ \mathsf{ok}_{\overline{p_1} \cup \overline{p_2}}}$$

The well-formedness of a local configuration relies on the well-typedness of the infostrate K and rules R. Note that a message store M is free to contain ill-typed terms—this captures the notion that a message store contains messages from all principals, including the adversary, who is free to send arbitrary messages, not just well-typed ones.

The following theorem ensures that well-formedness of a configuration is preserved under reduction. Note, the corresponding progress property (that a well-formed configuration can always make a step) is trivial, since identity steps ($G \longrightarrow G$) are always possible.

Theorem 4 (*Type soundness):* Given a configuration G such that G ok, if $G \longrightarrow G'$ then G' ok.

Proof: By induction over the structure of the of the local configuration that is evaluated. ■

The next theorem ensures that the order of evaluation of rules in a local configuration does not matter.

Theorem 5 (*Determinism of local rule evaluation):* Given a configuration G, a local configuration (p, P) such that $G \parallel (p, P)$ ok and A_1, A_2 such that $P \Downarrow_p A_1$ and $P \Downarrow_p A_2$; then $\mathsf{app}((G \parallel (p, P)), p, A_1) = \mathsf{app}((G \parallel (p, P)), p, A_2)$.

IV. A VERIFIED INTERPRETER FOR DKAL*

In this section, we describe our verified interpreter for DKAL*, implemented in F*, a variant of ML with a similar syntax and dynamic semantics but with a more expressive type system. Its type system allows the programmer to write down precise specifications using dependent types where types depend on values. F*'s type checker makes use of an SMT solver in an attempt to automatically discharge proofs of these specifications. F* enables general-purpose programming, with recursion and effects; it has libraries for concurrency, networking, cryptography, and interoperability with other .NET languages. After typechecking, F* is compiled to .NET bytecode, with runtime support for proof-carrying code.

Space constraints preclude a thorough review of F*—we refer the reader to Swamy et al. (2011) for details. Instead, we present selected elements of the mostly ML-like code of our interpreter (slightly simplified for the paper), discussing F*-specific constructs as they arise. The full code of our verified interpreter is available from `http://dkal.codeplex.com`.

We highlight three key elements of our interpreter:

A verified decision procedure for **QPIL.** We formalize the QPIL entailment relation using a collection of inductive types in F*. We then implement a unification-based, backwards chaining decision procedure for QPIL and prove it sound, i.e., that it only constructs valid entailments.

Authenticity of infons. Whereas the previous sections left the evidence terms associated with an infon abstract, in our interpreter we concretize evidence terms by representing them using digital signatures. By relying on previously developed verified libraries for cryptography, we prove a correspondence property on execution traces of DKAL* configurations.

Secure embedding of **F*** ***in*** **DKAL*.** We show how to securely implement the (**eval** e) construct, where the term e is an F* expression embedded within DKAL*. By relying on the type checker of F*, we show that embedded terms can safely be executed without breaking the invariants of the rest of the interpreter. This mechanism signficantly broadens the scope of DKAL*, allowing programmers to drop into a powerful general-purpose programming language when needed, and allowing a DKAL* protocol to be seamlessly integrated within the context of a larger secure system.

A. Formal syntax and typing of QPIL

We begin by discussing the abstract syntax and typing judgments used throughout our interpreter. Our interpreter for DKAL* is based on a prior, ad hoc implementation of DKAL in

F#—as such, our verified interpreter inherits some idiosyncracies from the DKAL system, so as to facilitate interoperability. The most significant of these inherited features is the definition of the abstract syntax of DKAL*, shown in the listing below. As is usual in ML, we define the syntax using a collection of algebraic types. We separate the syntax of quantified infons (polyterm) from infons, but, unlike in Section II, we use a single type term to represent both terms t and infons i.

```
type term =
  | Const : constant → term
  | Var : var → term
  | App : constructor → list term → term
and constant =
  | Bytes : bytes → constant
  | Prin : principal → constant
  | Int : int → constant
  | ...
and var = {name:string; typ:tau}
and tau = PrinT | BytesT | IntT | EvT | InfonT | QInfonT | ...
and constructor = Evidence | SaidInfon | AndInfon | ...
  | Rel : string → list tau → constructor
type polyterm =
  | Term : term → polyterm
  | Ev : term → polyterm → polyterm
  | Forall : vars → term → polyterm
  | PolyVar : var → polyterm
```

The type term includes constants, variables, and constructed terms. Constants include basic constructs like byte arrays, principal constants, etc. Variables var are pairs of a variable name (represented as a string) and its type typ, representing the types τ of Section II. Constructed terms are built using the App constructor. We illustrate using a few examples. We represent the DKAL* infon P said i as the term (App SaidInfon [Const (Prin "P"); I]), where I is the representation of i. As another example, we represent a tuple of integers, of any size, as the term (App Tuple [Const (Int 1); Const (Int 2)]. Finally, a relation in DKAL* Buy(1, 2) is represented as the term (App (Rel "Buy"[IntT;IntT]) [Const (Int 1); Const (Int 2)]).

Quantified infons are represented by the type polyterm, which includes term (corresponding to infons i), polyterm with evidence (corresponding to Ev t ι), prenex-quantified infon terms (Forall), and variables of type QInfonT.

The representation above is flexible in that it allows terms and infons to be represented by a single type term, but it allows malformed terms to be constructed, e.g., (App AndInfon [Const (Int 1)]). We recover well-formedness by expressing the typing judgment for QPIL using the inductive types shown (partially) below.

```
type constructor_typing :: constructor ⇒ list tau ⇒ tau ⇒ P =
  | CT_Said : constructor_typing SaidInfon [PrinT; InfonT] InfonT
  | CT_And : constructor_typing AndInfon [InfonT; InfonT] InfonT
  | ...
type typing :: vars ⇒ term ⇒ tau ⇒ P =
  | Ty_Int : G:vars → x:int → wfG G → typing G (Const (Int x)) IntT
  | Ty_Var : G:vars→ v:var→ Mem v G→ wfG G→ typing G (Var v) v.typ
  | Ty_App : ...
type polytyping :: vars ⇒ polyterm ⇒ P = ...
```

The inductive type constructor_typing is used to give types to values of type constructor. Its kind (constructor ⇒ list tau ⇒ tau ⇒ P) indicates that it constructs a proposition (of kind P) from three term arguments of type constructor, list tau and tau respectively. Values of type (constructor_typing c tl t) are witness that a constructor c can be applied to terms of type tl, to construct a term of type t. For example, using the constructor CT_Said, we assert that the SaidInfon constructor can be applied to a principal and an infon to build an infon. The definition of CT_Said shows the use of F*'s dependent function types. These have the form $x{:}t \to t'$, where x names the formal parameter of type t and is in scope in the return type t'.

We also show the signatures of typing, which corresponds to the judgments $\Gamma \vdash t : \tau$ and $\Gamma \vdash i$ of Section II. We show the two simple rules for illustration—the third rule, for typing App terms is longer, but essentially straightforward, through the use of the constructor_typing type. The last inductive type polytyping corresponds to the judgment $\Gamma \vdash \iota$.

We also define functions to decide typability of terms and polyterms. We show the signatures of these below, dependent functions in F* that given a context g:vars and a t:term or p:polyterm, returns a suitable typing derivation. In both cases, these functions are partial (e.g., they may raise exceptions if the term is untypeable). The F* type system requires partial functions that construct proof terms to be explicitly annotated as being partial—hence the Partial tag on the return type.

```
val doTyping: g:vars → t:term → Partial (ty:tau * typing g t ty)
val doPolyTyping: g:vars → p:polyterm → Partial (polytyping g p)
```

B. *Verifying a decision procedure for* QPIL *entailment*

Next, we turn to our mechanical formalization of QPIL entailment and the implementation of its decision procedure. We begin by showing the definitions of two mutually recursive inductive types, entails and polyentails. The type entails K G i corresponds to the judgment $K; \Gamma \models i$ and the type polyentails K G i corresponds to the judgment $K; \Gamma \models \iota$ (from Section II).

```
type prefix = list term
logic function Prefix : prefix → term → term
assume ∀i. (Prefix [] i) = i
assume ∀p pi i. (Prefix (p::pi) i) = (Prefix pi (App SaidInfon [p; i]))

type entails :: infostrate ⇒ vars ⇒ term ⇒ P =
  | Entails_And_Elim1: K:infostrate → G:vars
    → i:term → j:term → pi:prefix
    → entails K G (Prefix pi (App AndInfon [i; j]))
    → entails K G (Prefix pi j)
  | ...
and polyentails :: infostrate ⇒ vars ⇒ polyterm ⇒ P =
  | ...
  | Entails_Hyp_Knowledge :
      K:infostrate → G:vars → okCtx K G
    → i:polyterm{In i K} → i':polyterm
    → alphaEquiv i i' → polytyping G i'
    → polyentails K G i'
```

The code above illustrates two features of F*. First, we define the notion of an infon i with a quotation prefix π (written $\pi\ i$ in Section II). A quotation prefix is simply a list of terms and we define a function symbol Prefix to attach a prefix to term. This function is axiomatized by the **assume** equations, allowing the SMT solver underlying F*'s typechecker to reason about applications of the Prefix function symbol. Using this

construct, we can define the constructor Entails_And_Elim1, which corresponds to the rule ($\wedge$-E1).

Next, we show the definition of a constructor Entails_Hyp_Knowledge. Again, the correspondence with the rule (Hyp-K) is mostly straightforward, with the relation okCtx representing the well-formedness of the context and alphaEquiv corresponding to the relation $\equiv_\alpha$. The premise $\iota \in K$ from (Hyp-K) is represented by the *ghost refinement* type i:polyterm{In i K}, another feature of F*. This is the type of a polyterm i for which the property In i K is derivable by the SMT solver, without requiring the programmer to supply a (lengthy) constructive proof.

With the above types as our specification, we implement and prove sound a unification-based, goal-directed proof search procedure to (partially) decide QPIL entailment. Our algorithm is implemented by the function derivePoly, whose signature is shown below. The type says that in an infostrate K, given a quantified infon goal with free variables included in the set U, if successful in proving the goal, the function returns a substitution s whose domain includes the variables in U such that the subsitution s applied to the goal is derivable from K.

```
val derivePoly: K:infostrate → U:vars → goal:polyterm
                → option (s:substitution{Includes U (Domain s)} *
                          polyentails K [] (PolySubst goal s))
```

A limitation of our current work is that we do not prove our decision procedure complete. However, as discussed in Section II-D, we aim to use the insights behind Theorem 3 to extend our implementation to include a complete algorithm.

C. Main interpreter loop

The top-level of our interpreter is the infinite loop shown in the code below. At a high level, given a program represented by a list of rules rs, the interpreter computes and applies all enabled actions, and then, unless the actions cause a change to the local state, blocks waiting for new messages before looping.

```
let rec run (rs:list rule) =
  let actions = allEnabledActions rs in
  let stateChanged = applyAllActions actions in
  if stateChanged then run rs
  else (block_until_messages_received(); run rs)
```

Conceptually, the function allEnabledActions implements the local rule evaluation judgments $P \Downarrow_p A$, while applyAllActions implements message dispatch over the network, corresponding to the global transition step in the semantics of Section III. Recall that in our semantics the local configuration of a process, in addition to the rule set, involves two components: the infostrate K and the message store M. We represent each of these using mutable state and globally-scoped references. Each intepreter also has a single global constant, me:principal, the name of principal on whose behalf the interpreter runs.

The rules themselves are represented using the abstract syntax shown below.

```
type condition =
  | If : polyterm → condition
  | Upon : polyterm → var → condition
type action = | ... |
  | Send : term → polyterm → action
type rule = Rule : list condition → list action → rule
```

We also axiomatize rules corresponding to the holds function of Section III, and prove that the interpreter can apply only actions that have satisfiable guard conditions. As such, we prove a soundness property for our interpreter—the set of actions executed by the interpreter is a subset of the actions that may be executed in the operational semantics of Section III. A limitation, as in the case of the decision procedure, is that we do not prove completeness of our interpreter, i.e., we do not prove that *all* enabled actions are indeed computed and applied.

D. Authenticity of communications

As discussed earlier, the semantics of DKAL* presented in Section III is clearly insecure—a principal p can freely forge an infon. However, our setup hints at a solution: justified infons, terms of the form Ev t i carry evidence terms t that can be used to convince a recipient of the authenticity of the infon. In this section, we instantiate t using digital signatures.

Our goal is to prove an authenticity property by analyzing execution traces of a DKAL* protocol running in the presence of a Dolev-Yao network adversary. Informally, we relate an event recording the receipt of a message Ev t (q said ι) by an honest participant p at step k in an execution trace, to a corresponding event at step $k' < k$ recording the sending of the message Ev t (q said ι) by q, unless the signing key of q has been compromised, i.e., a standard correspondence property on traces (Woo and Lam 1993) to establish the authenticity of communications.

We set up the verification of this property following a methodology due to Gordon and Jeffrey (2003), and later in RCF (Bengtson et al. 2008) and F*. The basic idea is to augment the dynamic semantics of the programming language with a facility to accumulate protocol events in an abstract log, and to prove trace properties by analyzing the abstract log. The semantics of F* is equipped with just this facility: the reduction semantics of an F* program takes the form of a small-step relation $(A, e) \rightarrow (A', e')$, where an expression e reduces to e', while accumulating a set of events in an abstract log, which transitions from A to A'. Among the expression forms of F* is a construct **assume** ϕ, which adds the formula ϕ to the log, thus recording ϕ as a protocol event.

Broadly, our verification strategy is to record the sending of messages by adding an event (Sent p i) to the log when p sends a message i, and when receiving a message, through the use of a verified library of cryptographic primitives, we attempt to prove that the corresponding Sent event is in the log, unless the key of p has been leaked to the attacker. The listing below shows the declarations of predicate symbols Sent and Leaked—their kinds shows the use of F*'s E-kind, indicating that these predicates will be used purely to mark a protocol events, rather than as propositions with constructive proofs.

```
type Sent :: principal ⇒ polyterm ⇒ E
type Leaked :: principal ⇒ E
```

We give a flavor of the main elements in our proof below—the constructions are essentially standard; the reader may consult Swamy et al. (2011) for more details about our cryptographic libraries.

We start with the type of evidence used in our infon terms, a pair of a principal name and a byte string represent a digital signature of the infon it witnesses. The definition of CT_Ev below shows that we use the Evidence constructor to build a term of type EvT from a principal and a byte string.

```
type constructor_typing :: constructor ⇒ list tau ⇒ tau ⇒ P = ...
  | CT_Ev : constructor_typing Evidence [PrinT; BytesT] EvT
```

The code below shows a fragment of the API exposed by F*'s crypto library. It begins by defining an abstract predicate Serialized t x b to relate values of type x:t to their serialized representation as a byte string b. It then defines the type of digital signatures dsig, and the type of private and public keys. The type of a key carries three indexes, e.g., privkey t Q p, where t is a type, Q is a predicate on t-kinded values, and p is the name of the principal owning the key. Using the rsa_sign function and its private key, a principal p can sign a byte-string representation of a value of type x:t{Q x}, obtaining a signature dsig for it. Conversely, using rsa_verify with a public key, a principal can check if a claimed signature for some data x:t is authentic, obtaining the property Q x, unless the signing key of p has been compromised. The final funtion in the API below allows the private key of an principal p to be leaked to an adversary q—however, the type of leak ensures that each key compromise is logged using the Leaked p event.

```
type Serialized :: α::* ⇒ α ⇒ bytes ⇒ E
type dsig = bytes
type privkey :: α::* ⇒ (α ⇒ E) ⇒ principal ⇒ ⋆
type pubkey :: α::* ⇒ (α ⇒ E) ⇒ principal ⇒ ⋆
val rsa_sign: p:principal → privkey α 'P p → x:α
            → b:bytes{Serialized α x b && 'P x} → dsig
val rsa_verify: p:principal → pubkey α 'P p → x:α
              → b:bytes{Serialized α x b} → s:dsig
              → r:bool{r=true ⇒ ('P x || Leaked p)}
val leak: p:principal → q:principal → privkey α 'P p
        → _:privkey α (fun _ ⇒ True) q{Leaked p}
```

For use in our particular scenario, we provide each principal with keys to sign their own infons, and to authenticate the infons sent by others—these keys can be retrieved using the functions lookup_privkey and lookup_pubkey shown below. The predicate CanSign p used to index the keys of a principal p is defined so that p can only sign infons of the form p said i.

```
type CanSign (p:principal) (x:polyterm) =
  (∃ (i:term). x=(Infon (App SaidInfon [p;i]))) && Sent p x
val lookup_privkey : p:prin → option (privkey polyterm (CanSign p) p)
val lookup_pubkey : p:prin → pubkey polyterm (CanSign p) p
```

We also implement a marshaling layer for infons, and verify this implementation by relating these functions to a function Repr which axiomatizes the representation of a term as a byte string. We use this function to give an interpretation to the Serialized predicate.

```
logic function Repr : polyterm → bytes
assume ∀(p:polyterm). (Crypto.Serialized p (Repr p))
val polyterm2bytes : x:polyterm → b:bytes{(Repr x)=b}
val bytes2polyterm : b:bytes → option (x:polyterm{(Repr x)=b})
```

Finally, using all these elements, we can show a (simplified) fragment of applyOneAction, a function called from the main loop of the interpreter. We focus on the case where a message is signed and sent. We check that the message to be signed of the right shape; lookup the signing key; mark the protocol event; marshall the message to be signed; call rsa_sign (requiring the typechecker to prove the pre-condition CanSign p x); and, finally, package the message together with its evidence and send it to q.

```
let applyOneAction = function | ...
 | Send q ((App SaidInfon [p;i]) as x) where p=me →
    let Some sk = lookup_privkey me in
    assume(Sent p i); (* mark protocol start event *)
    let b = polyterm2bytes x in
    let dsig = rsa_sign me sk x b in
    Net.send q (Ev (mkEvidence p dsig) x)

let receiveMesage () = match bytes2polyterm (Net.receive ()) with
 | Some (Ev (App Evidence [Prin p; Bytes dsig]) x) →
    let pk = lookup_pubkey p in
    if rsa_verify p pk x b dsig
    then assert ((Sent p x) || (Leaked p)); (* protocol end *) else ...
```

On the receiver's side, when receiving a message with an evidence term, we check the evidence by calling rsa_verify to verify the signature and asserting that either Sent p i is in the log or Leaked p. From the type safety of F* (which ensures that the assertion never fails at runtime), we obtain authenticity by typing and our desired correspondence property.

E. Embedding F* in DKAL*

Our interpreter provides a simple and elegant solution to extend DKAL* with more general-purpose programming facilities. The example in Section I embeds an F* expression checkBalance "c""n" within a DKAL* protocol using the **eval** construct. When evaluating the if-condition, the interpreter executes the **eval**'d term by calling the F* function checkBalance defined along with the policy. Once in F*, we have the power of a full-fledged programming language at our disposal—we query a database to check if the customer has sufficient funds, update the database, and return the result (an infon) to the **eval** context.

```
let checkBalance c n (env:env) : t:term{typing [] t Infon} =
  match env[c], env[n] with
   | Const (Prin p), Const(Int n) →
     DB.transaction(fun () →
      let b = lookupBalance p in
      if b >= n then (updateBalance p (b − n); App TrueInfon [])
      else (App FalseInfon []))
```

Implementing **eval** is relatively straightforward. We parse the concrete syntax of DKAL* into the abstract syntax of our interpreter, leaving the **eval**'d terms as F* concrete-syntax expressions. We extend the type of DKAL* terms with one additional constructor. The Eval constructor takes two arguments: a type t:tau indicating the type that the **eval**'d term should reduce to, and a function from variable environments to terms. The refinement type given to the function ensures that the term it returns has the expected type.

```
type term = ...
  | Eval : t:tau → (env → tm:term{typing [] tm t}) → term
and env = map string term
```

Of course, one may be concerned that **eval**'ing an arbitrary F* term may be dangerous, e.g., it may inappropriately access internal data structures of the interpreter, or it could accept improperly signed messages, etc. However, because the **eval**'d term is statically typed by F*, we ensure that it never breaks any such critical invariants.

When evaluating the F* function, the interpreter passes in a variable environment as an argument, which contains bindings for each of the pattern variables in scope at the point where the **eval**'d term is defined. Aside from the mild inconvenience of passing parameters to F* in this dictionary style, we find the programming pattern quite natural. In the future, we plan to exploit this idiom at a larger scale, aiming to build and deploy full-fledged cloud services using this DKAL*/F* hybrid language.

F. Experimental evaluation

The table below shows 7 examples we developed using DKAL*. Each example involves one or more principals. Configuration files contain cryptography keys and communication ports for principals. Each principal stores her policies in a DKAL* file. The DKAL* file is compiled to F* for the interpreter to evaluate the rules. We measure the sizes of configuration files (column **Cfg**), the DKAL* files (column **DK.**), and the resulting F* files (column F*). All numbers are line counts of files.

Name	Descript.	Cfg	DK.	F*
Hello world	Two parties exchange hello messages.	13	14	45
Ping-Pong	Two parties bounce messages.	13	10	54
File system	A system restricting file accesses.	15	18	89
Calculator	Integer alrithmetic.	27	27	115
Turing Machine	A simulator of Turing machines.	22	40	121
Rumors	Four principals spread messages.	32	44	144
Retail	Our online retail example in the Intro.	25	59	195

These examples cover very diverse scenarios, ranging from simple message exchanges, to authorization, arithmetic, simulating turing machines, and online retailing. The diversity demonstrates DKAL*'s practicality. "Hello world" is the simplest, with two parties exchanging messages once. "Ping-pong" has more rounds of communication, with two parties bouncing messages back and forth. "File system" checks file access permission. A user U sends a justified message U said Ask(''f.txt'', U, ''read'') to the file system to request file access. The file system checks if the access is permitted according to the file system's knowledge. "Calculator" implements integer arithmetic, demonstrating the eval construct in DKAL*. A user sends an integer expression to the calculator. The calculator has embedded F* functions to compute the results and sends the results back. "Tur. Mach." simulates turing machines. It uses DKAL* policies to control state transitions. "Rumors" involves trust management among four parties. "Retail" is our example in Section I.

V. Related work

The design of DKAL* is informed by a long line of work on abstract state machines (ASMs), also called evolving algebras or dynamic structures (Gurevich 1995), and especially by the work on applications of the specification language AsmL (Gurevich et al. 2005) and the ASM-based Spec Explorer tool (CACM Staff 2011). ASMs have been successfully used for executable specifications of software and model-based testing. We intend to benefit not only from the ASM methodology but also from ASM tools, especially Spec Explorer, to design, analyze and deploy secure distributed authorization protocols.

More directly, DKAL* derives from its predecessor Evidential DKAL (Blass et al. 2011). Evidential DKAL extends the authorization logic DKAL (Gurevich and Neeman 2008) with a construct similar to our Ev t ι. The authors of Evidential DKAL also suggest incorporating the authorization logic within an ASM-based language to specify message flows. Our work improves on Evidential DKAL in a number of ways. First, we formulate QPIL in a manner suitable for mechanical verification—the prior formulation is informal in its treatment of quantifiers and variables. Next, although Evidential DKAL suggests incorporating an ASM-based language, it does not formalize this language at all—our semantics is novel. Of course, our verified implementation and embedding of F* in DKAL* is new. Indeed, in the process of our verification work, we encountered and fixed several bugs in the prior formulation, including one serious bug related to ill-scoped variables.

Our authorization logic QPIL is related to many prior logics used in a variety of trust management systems. These are too numerous to discuss exhaustively here—Chapin et al. (2008) provide a useful survey. However, one trust management system does bear mentioning. SD3 (Jim 2001) focus on the problem of deciding authorization by means of solving a query on a distributed database. A salient feature of SD3 is its certified evaluator, which is related to our verified decision procedure for QPIL. Both systems not only decide the validity of a query, but also construct a proof witness. Because it is implemented in an ad hoc manner, SD3 includes a proof-checking pass to ensure that the constructed witness is indeed a well-formed proof. In contrast, our implementation, by virtue of F*'s strong type system, statically guarantees that the constructed witness is always a valid proof without need for a separate proof-checking pass.

Another line of related work includes programming languages that are combined with authorization logics. For example, Aura (Jia et al. 2008) is a dependently typed functional programming language whose type system embeds the authorization logic DCC (Abadi 2006). Aura programmers are meant to build constructive proofs of authorization properties before performing security-sensitive operations. Our approach is perhaps the opposite. We provide a decision procedure for an authorization logic within the runtime of a high-level protocol-specification language, but allow terms from a dependently typed functional programming language to be embedded within the specification.

Finally, our approach to embedding F* terms inside DKAL* and then compiling the result to F* for interpretation is a weak form of meta-programming. In a sense, our approach is related to template Haskell (Sheard and Jones 2002) in that after code generation, we typecheck the resulting program as a normal

F* program before interpretation. However, unlike template Haskell, we do not support execution of embedded F* code when generating F* from DKAL*. As such, our approach also resembles the facility provided by many C compilers which allow inlining assembly instruction in source programs, which are carried through verbatim by the compiler, although, of course, in DKAL*, the inlined F* expressions are type checked before execution.

VI. Conclusions

We have shown DKAL*, a language that allows to both specify and execute distributed authorization protocols. A DKAL* program is a set of rules, each rule specifying some actions to take when receiving some messages matching a pattern. We have formalized DKAL*, giving it an operational semantics and a type system. We have also built a DKAL* interpreter, mechanically verified for correctness and security. Protocol designers can use our formalization to analyze their authorization policies, while programmers can use our verified interpreter to deploy them.

References

M. Abadi. Access control in a core calculus of dependency. *SIGPLAN Not.*, 41(9):263–273, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1160074.1159839.

M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. In *CRYPTO*, pages 1–23, 1991.

M. Becker, C. Fournet, and A. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, 2009.

A. Blass, Y. Gurevich, M. Moskal, and I. Neeman. Evidential authorization. In S. Nanz, editor, *The Future of Software Engineering*, pages 73–99. Springer, 2011.

CACM Staff. Microsoft's protocol documentation program: interoperability testing at scale. *Commun. ACM*, 54(7): 51–57, July 2011. ISSN 0001-0782. doi: 10.1145/1965724.1965741. URL `http://doi.acm.org/10.1145/1965724.1965741`.

P. Chapin, C. Skalka, and X. Wang. Authorization in trust management: Features and foundations. *ACM Computing Surveys (CSUR)*, 40(3):9, 2008.

A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 300(1-3):379–409, 2003.

Y. Gurevich. Evolving algebra 1993: Lipari guide. *Specification and Validation Methods*, 1995.

Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *21st IEEE Computer Security Foundations Symposium*, pages 149–162. IEEE, 2008.

Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.

J.-B. Jeannin, G. de Caso, J. Chen, Y. Gurevich, P. Naldurg, and N. Swamy. DKAL*: Constructing executable specifications of authorization protocols (extended version). Technical report, Microsoft Research, 2012. Available from `http://research.microsoft.com/fstar`.

L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *ICFP*, 2008.

T. Jim. SD3: A trust management system with certified evaluation. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 106–115. IEEE, 2001.

T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02. ACM, 2002.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.

R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *IEEE Symposium on Security and Privacy*, pages 465–480, 2011.

T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, SP '93, pages 178–, Washington, DC, USA, 1993. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=882489.884188`.

Appendix

A. Soundness of the type system

In this appendix we prove theorem 4. Let us first introduce a notion of well-typedness for substitutions: given a substitution σ and type environment Γ, we write $\vdash \sigma : \Gamma$ whenever $\mathrm{dom}\,\sigma = \mathrm{dom}\,\Gamma$, and for all $x \in \mathrm{dom}\,\sigma$, $\vdash \sigma(x)$ if $\Gamma(x) = \mathsf{infon}$ and $\vdash \sigma(x) : \Gamma(x)$ otherwise.

Lemma 6 (*Substitution lemma for terms):* If $\vdash \sigma : \Gamma$ and $\Gamma, \Gamma' \vdash t : \tau$ then $\Gamma' \vdash \sigma\ t : \tau$.

Proof: If $t = c_\tau$, $\sigma\ t = c_\tau$ and $\Gamma' \vdash c_\tau : \tau$.

If $t = x$, $\sigma\ t = \sigma(x)$ and $\tau = \Gamma(x)$, and since $\vdash \sigma : \Gamma$, $\Gamma' \vdash \sigma\ t : \tau$. ∎

Lemma 7 (*Substitution lemma for infons):* If $\vdash \sigma : \Gamma$ and $\Gamma, \Gamma' \vdash i$ then $\Gamma' \vdash \sigma\ i$.

Proof: By structural induction on the derivation of $\Gamma \vdash i$.

If $\iota = \top$ the result is trivial.

If $\iota = i \wedge j$, then $\Gamma, \Gamma' \vdash i$ and $\Gamma, \Gamma' \vdash j$, therefore by induction hypothesis $\Gamma' \vdash \sigma(i)$ and $\Gamma' \vdash \sigma(j)$, leading to $\Gamma' \vdash \sigma(i \wedge j)$. The remaining cases are similar, some needing to use lemma 6. ∎

Lemma 8 (*Substition lemma for quantified infons):* If $\vdash \sigma : \Gamma$ and $\Gamma \vdash \iota$ then $\vdash \sigma\ \iota$.

Proof: By structural induction on the derivation of $\Gamma \vdash \iota$.

If $\iota = \forall \overline{x : \tau}.i$, then $\Gamma, \overline{x : \tau} \vdash i$, which ensures that none of the variables in $\mathrm{dom}\,\overline{x : \tau}$ appears in Γ. Therefore $\sigma\ \iota =$

$\forall \overline{x:\tau}.\sigma(i)$, and using lemma 7, $\overline{x:\tau} \vdash \sigma(i)$, therefore $\vdash \sigma\ \iota$. ■

Lemma 9 (*Substitution lemma for actions):* If $\vdash\ \sigma\ :\ \Gamma, \Gamma'$ and $\Gamma \vdash A$ then $\vdash \sigma\ A$.

Proof: By structural induction on A. All cases are similar to the $i \wedge j$ case of the proof of lemma 7, and some may refer to lemmas 6 and 7. The variables appearing in Γ' are irrelevant since they do not appear in A, and thus are never replaced. ■

Lemma 10 (*Substitution lemma for conditions):* If $\vdash\ \sigma\ :\ \Gamma$ and $\Gamma \vdash C : \Gamma'; \Gamma''$ then $\vdash \sigma\ C : \Gamma'; \Gamma''$.

Proof: By structural induction on C. All cases are similar to $i \wedge j$ case of the proof of lemma 7, and some may refer to lemmas 6 and 7. ■

Lemma 11 (*Free variables):*
If $\Gamma \vdash t : \tau \rightsquigarrow \Gamma'$ then $\text{FV}(t) \subseteq \text{dom}\,\Gamma \cup \text{dom}\,\Gamma'$;
if $\Gamma \vdash i \rightsquigarrow \Gamma'$ then $\text{FV}(i) \subseteq \text{dom}\,\Gamma \cup \text{dom}\,\Gamma'$;
if $\Gamma \vdash \iota : \Gamma'; \Gamma''$ then $\text{FV}(\iota) \subseteq \text{dom}\,\Gamma \cup \text{dom}\,\Gamma' \cup \text{dom}\,\Gamma''$;
if $\Gamma \vdash C : \Gamma'; \Gamma''$ then $\text{FV}(C) \subseteq \text{dom}\,\Gamma \cup \text{dom}\,\Gamma' \cup \text{dom}\,\Gamma''$;
if $\Gamma \vdash A$ then $\text{FV}(A) \subseteq \text{dom}\,\Gamma$;
and in all cases, Γ, Γ' and Γ'' have pairwise disjoint domains.

Proof: Easy induction on the structure of each premise. ■

Lemma 12 (*Soundness of $\rightsquigarrow$):* If $\Gamma \vdash \sigma\ i$ and $\Gamma \vdash i \rightsquigarrow \Gamma'$ then $\forall x \in \text{dom}\,\Gamma'$, $x \in \text{dom}\,\sigma$ and $\Gamma \vdash \sigma(x)$ if $\Gamma'(x) = \mathsf{infon}$, or $\Gamma \vdash \sigma(x) : \Gamma'(x)$ otherwise.

If $\Gamma \vdash \sigma\ \iota$ and $\Gamma \vdash \iota \rightsquigarrow \Gamma'; \Gamma''$ then $\forall x \in \text{dom}\,\Gamma', \Gamma''$, $x \in \text{dom}\,\sigma$ and $\Gamma \vdash \sigma(x)$ if $(\Gamma', \Gamma'')(x) = \mathsf{infon}$, or $\Gamma \vdash \sigma(x) : (\Gamma', \Gamma'')(x)$ otherwise.

Proof: The first result is by structural induction on the derivation of $\Gamma \vdash i \rightsquigarrow \Gamma'$.

In the cases where $\Gamma' = \cdot$ and $\Gamma'' = \cdot$, the result is trivial.

If $i = x$, $\Gamma' = x : \mathsf{infon}$ with $x \notin \text{dom}\,\Gamma$, and $\Gamma'' = \cdot$, then $\text{dom}\,\Gamma' = \{x\}$ and we already know $\Gamma \vdash \sigma\ x$. The case where $\iota = x$ and $\Gamma' = x : \tau$ with $x \notin \text{dom}\,\Gamma$ is similar.

If $i = j_1\ op\ j_2$, there exists Γ_1, Γ_2 such that $\Gamma' = \Gamma_1, \Gamma_2$, $\Gamma \vdash j_1 \rightsquigarrow \Gamma_1$ and $\Gamma, \Gamma_1 \vdash j_2 \rightsquigarrow \Gamma_2$. Since $\Gamma \vdash \sigma(j_1\ op\ j_2)$, $\Gamma \vdash \sigma(j_1)$ and $\Gamma \vdash \sigma(j_2)$. By induction hypothesis on j_1, the conclusion is true for all $x \in \text{dom}\,\Gamma_1$, and by induction hypothesis on j_2, it is also true for all $x \in \text{dom}\,\Gamma_2$. Since $\Gamma' = \Gamma_1, \Gamma_2$, this concludes this case. The other remaining cases are similar.

The second result is by structural induction on the derivation of $\Gamma \vdash \iota \rightsquigarrow \Gamma'; \Gamma''$.

If $\iota = \forall \overline{x:\tau}.i$, then $\Gamma' = \cdot$. Invoking the first result on i allows to conclude.

If $\iota = \mathsf{Ev}\ t\ \iota$, the case is similar to $i = j_1\ op\ j_2$ in the previous theorem. ■

Lemma 13 (*Substitutions generated by hold are well-typed):* If S ok, K ok, $\vdash C : \Gamma; \Gamma'$ and $\sigma, \epsilon \in \mathsf{holds}_p\, K\, M\, C$, then $\vdash \sigma : \Gamma, \Gamma'$.

Proof: By structural induction on C.

If $C = \cdot$ or $C = \textbf{if}\ \iota$, then $\sigma = id$, i.e., $\vdash \sigma : \cdot$. Because the typing on guards if purely syntactic, it is necessary that $\Gamma = \cdot$ and $\Gamma' = \cdot$, therefore $\vdash \sigma : \Gamma, \Gamma'$.

If $C = \mathsf{upon}\ \iota\ \textbf{as}\ x$ then there exists Γ'' such that $\Gamma = \Gamma'', x : \mathsf{qinfon}$, and σ' such that $\sigma = \sigma', x \mapsto \sigma'\iota$, with $\vdash \sigma'\iota$, $\vdash \iota \rightsquigarrow \Gamma''; \Gamma'$, $x \notin \text{dom}(\Gamma'', \Gamma')$ and $\text{dom}\,\sigma' = \text{FV}(\iota)$. We now need to prove $\vdash\ \sigma'\ :\ \Gamma'', \Gamma'$. From lemma 11 we know that $\text{FV}(\iota) \subseteq \text{dom}(\Gamma'', \Gamma')$, and from lemma 12, we know that $\text{dom}(\Gamma'', \Gamma') \subseteq \text{dom}\,\sigma'$, therefore $\text{dom}\,\sigma' = \text{FV}(\iota) = \text{dom}(\Gamma'', \Gamma')$. Again using lemma 12, we know that $\forall x \in \text{dom}\,\sigma'$, $\vdash \sigma'(x)$ if $(\Gamma'', \Gamma')(x) = \mathsf{infon}$, or $\vdash \sigma(x) : (\Gamma'', \Gamma')(x)$ otherwise. This means that $\vdash \sigma' : \Gamma'', \Gamma'$, which concludes this case.

If $C = C_1\ C_2$, then there is Γ_1, Γ_1', Γ_2 and Γ_2' such that $\Gamma = \Gamma_1, \Gamma_2$, $\Gamma' = \Gamma_1', \Gamma_1'$, $\vdash C_1 : \Gamma_1; \Gamma_1'$, $\Gamma_1 \vdash C_2 : \Gamma_2; \Gamma_2'$ and $\text{dom}\,\Gamma_1' \cap \text{dom}\,\Gamma_2' = \emptyset$, and σ_1, σ_2 such that $\sigma = \sigma_2 \circ \sigma_1$, $\sigma_1 \in \mathsf{holds}_p\, S\, K\, M\, C_1$ and $\sigma_2 \in \mathsf{holds}_p\, S\, K\, M(\sigma_1\ C_2)$. By induction hypothesis on σ_1, $\vdash \sigma_1 : \Gamma_1, \Gamma_1'$. Now using lemma 11, $\text{FV}(C_2) \subseteq \text{dom}(\Gamma_1, \Gamma_2, \Gamma_2')$, and the domains of $\Gamma_1, \Gamma_2, \Gamma_2'$ on one side, and Γ_1' on the other side, are disjoint. Therefore, if σ_1' is the substitution σ_1 restricted to $\text{dom}\,\Gamma_1$, then $\sigma_1\ C_2 = \sigma_1'\ C_2$ and $\vdash \sigma_1' : \Gamma_1$. Using lemma 10, leads to $\vdash\ \sigma_1'\ C_2\ :\ \Gamma_2; \Gamma_2'$, i.e., $\vdash\ \sigma_1\ C_2\ :\ \Gamma_2; \Gamma_2'$. By induction hypothesis on σ_2, $\vdash \sigma_2 : \Gamma_2, \Gamma_2'$. Now Γ_1, Γ_2, Γ_1' and Γ_2' have disjoint domains, therefore σ_1 and σ_2 have disjoint domains too and $\vdash \sigma_2 \circ \sigma_1 : \Gamma, \Gamma'$. ■

Lemma 14 (*Soundness of* app*):* If G ok and $\vdash\ A$ then $\mathsf{app}(G, p, A)$ ok.

Proof: app G p A consists in applying all the individual actions one by one, starting with all the drop actions. If we can prove the lemma for individual actions, a simple recurrence on the sequence of actions will prove the lemma for all actions.

Let us now prove it for individual actions: we prove that if $(p, (K, M, R))\ \mathsf{ok}_{\{p\}}$ and $\vdash A$, and if app1 $(p, (K, M, R))\ q\ A$ exists, then app1 $(p, (K, M, R))\ q\ A\ \mathsf{ok}_{\{p\}}$. If A is a drop ι the result is trivial. If it is a learn ι, from $\vdash A$ we know that $\vdash \iota$, therefore (K, ι) ok. If it is a fwd $q\ \iota$ or a send $q\ \iota$, it is trivial too. ■

Proof of the Soundness Theorem: $G \longrightarrow G'$ therefore there exists G_1, G_2, p, P and A such that $P \Downarrow_p A$, $G = G_1 \parallel (p, P) \parallel G_2$, and $G' = \mathsf{app}(G, p, \mathsf{order}(A))$. G ok, Therefore there exists $\overline{p_1}$ and $\overline{p_2}$ disjoint from each other and from p such that $G_1\ \mathsf{ok}_{p_1}$, $G_2\ \mathsf{ok}_{p_2}$ and $(p, P)\ \mathsf{ok}_{\{p\}}$. Let S, K, M, R such that $P = (S, K, M, R)$, and let us prove that $\vdash A_0$ for all $A_0 \in A$, by induction on the structure of R such that $\vdash R$.

If $R = \cdot$ then $A = \{\}$, therefore the result is vacuously true.

If $R = R_1, (C\ \textbf{then}\ A_1), R_2$ then $A = A' \cup_i [\![\sigma_i A_1]\!]$ where by induction hypothesis all the elements of A' already verify the desired property, and the $\sigma_i \in \mathsf{holds}_p\, S\, K\, M\, C$. Because $\vdash C\ \textbf{then}\ A_1$, there exists Γ such that $\vdash C : \Gamma; \Gamma'$ and $\Gamma \vdash A_1$. Since S ok and K ok, for any of the σ_i we can apply lemma 13 which leads to $\vdash \sigma_i : \Gamma, \Gamma'$. Applying lemma 9 leads to $\vdash \sigma_i\ A_1$. Applying lemma 9 again for actions with a fresh x leads to $\vdash [\![\sigma_i\ A_1]\!]$.

Therefore $\vdash A_0$ for all $A_0 \in A$. Since the set of actions in A and $\mathsf{order}(A)$ are the same, $\vdash A_0$ for all $A_0 \in \mathsf{order}(A)$. We conclude by applying lemma 14. ■