DKAL*: Constructing Executable Specifications of Authorization Protocols

Jean-Baptiste Jeannin[1], Guido de Caso[2], Juan Chen[3],
Yuri Gurevich[3], Prasad Naldurg[3], and Nikhil Swamy[3]

[1] Cornell University
[2] Universidad de Buenos Aires
[3] 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. To reason 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 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.

1 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. [24] 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) and *executable*.

To illustrate, consider the following scenario. An online retailer W wishes to use a third-party payment provider P (e.g., PayPal) to process payments. As Wang et al. report, many of the existing tools used to build such a website 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 [14], one might write a policy for W stating that it is safe to conclude that a principal c paid n for order oid, if P said so: $\forall$c, oid, n. P said Paid(c, oid, n) $\Longrightarrow$Paid(c, oid, n).

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 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.

Typically, one implements such protocols in a general-purpose programming language, where one makes queries to a trust management engine (e.g., SecPAL [4]) to determine if access to a protected resource is to be permitted. While this approach provides flexibility, it leaves the design of the authorization protocol unclear, and opens the door to vulnerabilities due to improper protocol design or other mundane programming errors. Of course, such errors can be detected by using semi-automated program verification tools, but this demands considerable expertise. Besides, even for experts, a methodology in which the protocol design is made evident by construction, facilitates simpler analysis.

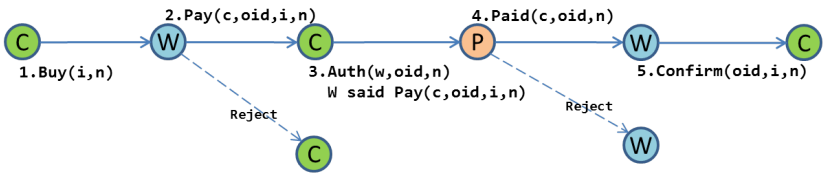

Fig. 1: A simple protocol for processing online payments

To address these problems, we propose DKAL*, a domain-specific language for executable specifications of authorization protocols. We formalize the semantics of DKAL* and implement a verified interpreter using F*, a verification-oriented dialect of ML. DKAL* programs include three conceptual components: the quantified primal infon logic (QPIL) for expressing distributed trust relationships; a small rule-based programming language for describing message flow of protocols; and finally, DKAL* programs may embed F* expressions, the host language of our interpreter—one can use this facility to evaluate arithmetic expressions, connect to databases, etc. Thus, having designed a protocol in DKAL*, one may readily obtain an executable implementation in F*. Once in F*, the source code can, in principle, be directly analyzed for high-level security properties using F*'s type system and related tools such as the Crypto Verification Toolkit [7]. However, in this work, we take DKAL* programs as specifications, and our interpreter is

proven to faithfully implement the specification, regardless of any end-to-end security objective that the DKAL* programmer may have had in mind.

Figure 2 shows an example of DKAL* code, a policy specified by each of the three principals in our online retail scenario. 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. 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 (e.g., i, n); constants (e.g., W, P); and constructed terms (Buy(i, n), etc.).

```
(* 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

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 a click issued by the customer. C sends a Buy message to the website W. C also logs an Init message to indicate that she has initiated the transaction. (2) After receiving the Buy message, W applies rule W_2 to request a payment certificate. W checks the price of the item (by calling an F* function), and sends a message Pay to C requesting payment. 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 Init message, 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 Paid message to the website W (after checking and updating C's balance, using F*). (5) If W receives a Paid message, she uses her trust assumption in P (W_6 and W_7), and a decision procedure for QPIL to conclude that the item is paid, and sends a confirmation message to C (W_5).

This paper makes several technical contributions.

(i) We formalize the design of DKAL* and analyze the central entailment relation of QPIL. We give an operational semantics and a type system for DKAL* and prove that execution is insensitive to the order of rule evaluation. Our semantics provides the formal basis on which to analyze DKAL* policies.

(ii) We provide an interpreter for DKAL* in F*. We mechanically check with F* that our interpreter soundly implements the formal semantics of DKAL*, including a verified implementation of a decision procedure for QPIL. Our interpreter includes a verified protocol based on public-key cryptography for establishing message authenticity, where we can mechanically check that recipients only accept 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.

(iii) We report on an experimental evaluation of DKAL* by developing a suite of 8 examples. 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.

2 QPIL: Quantified primal infon logic

We first review QPIL, Gurevich and Neeman's primal infon logic with quantifiers. Gurevich and Neeman introduced QPIL pragmatically, because of its combination of feasibility and expressivity. But QPIL is arguably one of two intrinsic logics of information (used by arbitrary principals for communication and reasoning) [5]. Our formulation differs from theirs in that we pay close attention to binders to facilitate a mechanically verified implementation of its decision procedure.

Syntax QPIL has 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 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 syntax of QPIL is shown below. 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), principals, and a distinguished type for evidence terms, ev. The terms include variables x, y, z and constants c (tagged) with their types. Later (Section 4) we add embedded F* terms to the term language.

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) $\Longrightarrow$ 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.

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	$::= \top \mid Q_{\bar\tau}\ \bar t \mid i \wedge j \mid i \Rightarrow j$	quantified infon	ι	$::= i \mid \text{Ev}\ t\ \iota \mid \forall\overline{x{:}\tau}.i$
		$\mid p$ said $i \mid \text{Ev}\ t\ i$	type context	Γ	$::= \cdot \mid x{:}\tau \mid \Gamma, \Gamma$
infon set	M, K	$::= \bar\iota$			

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 $\Gamma(x)$ for the type τ such that Γ contains $x : \tau$.

Typing terms and infons

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

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

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.

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** $K; \Gamma \vDash i$

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

$$\frac{K; \Gamma \vDash \pi\ i \qquad K; \Gamma \vDash \pi\ j}{K; \Gamma \vDash \pi(i \wedge j)}\ \wedge\text{-I} \qquad \frac{K; \Gamma \vDash \pi(i \wedge j)}{K; \Gamma \vDash \pi\ i}\ \wedge\text{-E1} \qquad \frac{K; \Gamma \vDash \pi(i \wedge j)}{K; \Gamma \vDash \pi\ j}\ \wedge\text{-E2}$$

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

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

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', 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 and elided due to space constraints.

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

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. Finally, we have (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 $\Gamma \vdash \iota$.*

Decidability of QPIL There exists a complete decision procedure for QPIL entailment. Gurevich and Neeman [14] present a linear-time algorithm for the multiple derivability problem for propositional primal infon logic (PIL). 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. QPIL exhibits an analogous property. We refer the reader to a companion technical report for the full development [17].

While the existence of a complete decision procedure for QPIL is useful, 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.

3 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.

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-passing semantics for DKAL* in which the reduction of a local configuration P causes infons to be sent to other principals.

Syntax of DKAL* (with syntactic sugar on the right)

program	$R ::= C \text{ then } A \mid R\ R \mid \cdot$	when ι then A	$=$ upon Ev $x\ \iota$ as m then $(A, \text{drop } m)$ for fresh x and m
local cfg.	$P ::= (K, M, R)$		
global cfg.	$G ::= (p, P) \mid G \parallel G$		
guards	$C ::= \text{upon } \iota \text{ as } x \mid \text{if } \iota \mid C\ C \mid \cdot$	log ι	$=$ send Self ι
actions	$A ::= \text{send } p\ \iota \mid \text{fwd } p\ \iota \mid \text{drop } \iota$ $\mid$ learn $\iota \mid$ with fresh $x\ A \mid A\ A$		
infon	$i ::= \ldots \mid x$		
typing ctxt.	$\Gamma ::= \ldots \mid x$:infon $\mid x$:qinfon		

Guards come in two flavors. The guard $(\text{upon } \iota \text{ as } x)$ 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 $(\text{if } \iota)$. 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 $(\text{send } p\ \iota)$, which sends ι to p authenticated by the sender; $(\text{fwd } p\ \iota)$, which forwards a previously received message to p; $(\text{drop } \iota)$, which deletes a message from M; $(\text{learn } \iota)$, which adds an infon to the knowledge K; and, finally, a construct $(\text{with fresh } x\ A)$ to generate fresh identifiers. In writing examples, we also use the syntactic sugar shown at the right of the display, where Self is a principal constant for the local principal.

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 3.

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.

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.

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.

Local rule evaluation: $P \Downarrow_p A$

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

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

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.

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. We do this through a unary operator on actions, $\mathsf{order}(A)$, that reorders a set of actions A according to a partial order in which all the (drop ι) actions come first.

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

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

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

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, following the syntax given in section 3. 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.

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 σ = [x ↦ 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. After defining several judgments on rules, actions and guards, it defines a judgment G ok for well-formedness of a global configuration G. Space constraints prevent us from presenting the full details of the type system here—the companion technical report contains the full development [17].

Theorem 1 ensures that well-formedness of a configuration is preserved under reduction. 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 2 ensures that the order of evaluation of rules in a local configuration does not matter.

Theorem 1 (Type soundness). *Given a configuration G such that G* ok*, if $G \longrightarrow G'$ then G'* ok.

Theorem 2 (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* $\mathsf{app}((G \parallel (p, P)), p, A_1) = \mathsf{app}((G \parallel (p, P)), p, A_2)$.

4 A verified interpreter for DKAL*

This section describes our verified interpreter for DKAL*, implemented in F*, a variant of ML with a more expressive type system. F* allows programmers to write down precise specifications using dependent types where types depend on values. F*'s type checker makes use of an SMT solver 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.

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. We refer the reader to Swamy et al. [22] for full definition of F*. 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 evidence terms are represented as 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 significantly broadens the scope of DKAL*, empowering programmers with a powerful general-purpose programming language when needed, and allowing a DKAL* protocol to seamlessly integrate within the context of a larger secure system.

As is usual in ML, our interpreter defines DKAL* syntax using a collection of algebraic types. We separate the syntax of quantified infons (polyterm) from infons, but, unlike in Section 2, we use a single type term to represent both terms t and infons i. This representation 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. We recover well-formedness by expressing the typing judgment for QPIL using inductive types (see the companion technical report [17]).

Verifying a decision procedure for QPIL *entailment* We show below our mechanical formalization of QPIL entailment and the implementation of its decision procedure. We define two mutually recursive inductive types, entails and polyentails. The type entails K G i corresponds to the judgment $K; \Gamma \models i$, and polyentails K G i corresponds to the judgment $K; \Gamma \models \iota$ (from Section 2).

```
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 2). 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).

The constructor Entails_Hyp_Knowledge corresponds to the rule (Hyp-K), 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 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 substitution 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))
```

The completeness of QPIL comes from [8]. We aim to extend our implementation to include a complete algorithm.

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 3. 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 interpreter also has a single global constant, me:principal, the name of principal on whose behalf the interpreter runs.

We also axiomatize rules corresponding to the holds function of Section 3, 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 3. 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.

Authenticity of communications As discussed earlier, the semantics of DKAL* presented in Section 3 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 [25] to establish the authenticity of communications.

We set up the verification of this property following a methodology due to Gordon and Jeffrey [13], and later in RCF [6] 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.

Broadly, we 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.

We give a flavor of the main elements in our proof in the companion technical report [17]—the constructions are essentially standard; the reader may consult Swamy et al. [22] for more details about our cryptographic libraries.

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 1 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.

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. 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.

Experimental evaluation The table below shows 8 examples we developed using DKAL*. 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 **Config**), the DKAL* files (column **DKAL**), and the resulting F* files (column F*). All numbers are line counts of files.

Name	**Description**	**Config**	**DKAL**	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 arithmetic.	27	27	115
Turing Machine	A simulator of Turing machines.	22	40	121
Rumors	Four principals spread messages.	32	22	144
Retail	Our online retail example in the Intro.	25	59	195
Clinical Trials	Checking that a physician can conduct a trial.	57	86	296

These examples cover diverse scenarios, ranging from simple message exchanges, to authorization, arithmetic, simulating turing machines, and online retailing. "Hello world" and "Pingpong" are simple message exchanges. "File system" has user U send a justified message U said Ask(''f.txt'', U, ''read'') to the file system to request file access and responds if authorized. "Calculator" implements integer arithmetic, demonstrating the eval construct. "Turing Machine" simulates turing machines. It uses DKAL* policies to control state transitions. "Rumors" involves trust management among four parties. "Retail" is our example in Section 1.

Our most complex example is "Clinical Trials", which simulates a pharmaceutical company hiring an independent research organization to conduct a clinical trial before releasing a new drug. This scenario was originally discussed by Blass et al.[10]. Briefly, the research organization hires sites such as hospitals or labs to execute the

trial. Each site finds appropriate patients and assigns physicians to work with them. We use DKAL* to specify a protocol to enforce patient privacy: patient records are guarded by a key manager, which gives only authorized physicians the keys to access patient records. The protocol involves four message exchanges among the principals and reasoning about integer arithmetic and authorization delegation. We tie the abstract DKAL* specification to a concrete implementation of messages and wire formats through the use of standard cryptographic protocols for authentication, but with implementations that are verified (in a symbolic model) against the abstract DKAL* specifications. The arithmetic reasoning is performed by embedded F* expressions.

5 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 [15], and especially by the work on applications of the specification language AsmL [16] and the ASM-based Spec Explorer tool [11]. More directly, DKAL* derives from its predecessor Evidential DKAL [10]. Evidential DKAL extends the authorization logic DKAL [14] with a construct similar to our Ev t ι. The evidential nature of DKAL is related to Necula's proof-carrying code [20] that was followed by proof-carrying authentication [3] and more recently by evidence-based audit [23] and code-carrying authorization [1].

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—our semantics is novel. Our verified implementation and embedding of F* in DKAL* is new. In the process of our verification, we found 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. [12] provide a useful survey. One representative however is SD3 [19], where the problem of deciding authorization by means of solving a query on a distributed database is studied. SD3 has a 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. SD3 requires an additional proof checking step, whereas our system statically guarantees that we construct only valid proofs.

Another line of related work includes programming languages that are combined with authorization logics. For example, Aura [18] is a dependently typed functional programming language whose type system embeds the authorization logic DCC [2]. Aura programmers build constructive proofs of authorization before performing security sensitive operations, whereas we provide a decision procedure within the runtime, and allow the embedding of F* terms in the specification.

Compiling DKAL* to F* allows the possibility of using F*'s verification-oriented type system to prove various properties of the protocol implementation. Thus DKAL* stands to benefit both from the extensive study of properties of abstract protocol models and the automated verification of protocol implementations. This line of work, too

extensive to discuss in detail here, is covered thoroughly by a recent survey on protocol verification [9].

Our approach to embedding F* terms inside DKAL* compiling the result to F* for interpretation is a weak form of meta-programming. It is related to template Haskell [21] 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 is similar to inlining assembly instructions by many C compiler, with additional typechecking before execution.

Conclusions. DKAL* is a language that allows for the specification and execution of distributed authorization protocols.We have formalized DKAL*, giving it an operational semantics and a type system. We have also built a DKAL* interpreter, mechanically verified to soundly implement its semantics. Protocol designers can use our formalization to describe and analyze their authorization policies, while programmers can use our verified interpreter to deploy them.

References

[1] Abadi, M., Maffeis, S., Fournet, C., Gordon, A.: Code-carrying authorization. In: ESORICS 2008. pp. 563–579. Springer (2008)

[2] Abadi, M.: Access control in a core calculus of dependency. SIGPLAN Not. 41(9), 263–273 (2006)

[3] Appel, A., Felten, E.: Proof-carrying authentication. In: CCS'99. pp. 52–62. ACM (1999)

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

[5] Beklemishev, L., Blass, A., Gurevich, Y.: What is the logic of information? (2012), in Preparation

[6] Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: CSF (2008)

[7] Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for TLS. In: ACM Conference on Computer and Communications Security. pp. 459–468 (2008)

[8] Bjørner, N., de Caso, G., Gurevich, Y.: Correct reasoning. chap. From primal infon logic with individual variables to datalog, pp. 72–86. Springer-Verlag, Berlin, Heidelberg (2012)

[9] Blanchet, B.: Security protocol verification: Symbolic and computational models. In: Degano, P., Guttman, J. (eds.) First Conference on Principles of Security and Trust (POST'12). Lecture Notes on Computer Science, vol. 7215, pp. 3–29. Springer Verlag, Tallinn, Estonia (Mar 2012)

[10] Blass, A., Gurevich, Y., Moskal, M., Neeman, I.: Evidential authorization. In: Nanz, S. (ed.) The Future of Software Engineering. pp. 73–99. Springer (2011)

[11] CACM Staff: Microsoft's protocol documentation program: interoperability testing at scale. Commun. ACM 54(7), 51–57 (Jul 2011), `http://doi.acm.org/10.1145/1965724.1965741`

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

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

[14] Gurevich, Y., Neeman, I.: DKAL: Distributed-knowledge authorization language. In: 21st IEEE Computer Security Foundations Symposium. pp. 149–162. IEEE (2008)
[15] Gurevich, Y.: Evolving algebra 1993: Lipari guide. Specification and Validation Methods (1995)
[16] Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL. Theor. Comput. Sci. 343(3), 370–412 (2005)
[17] Jeannin, J.B., de Caso, G., Chen, J., Gurevich, Y., Naldurg, P., Swamy, N.: DKAL*: Constructing executable specifications of authorization protocols (extended version). Tech. rep., Microsoft Research (2012), available from `http://research.microsoft.com/fstar`
[18] Jia, L., Vaughan, J., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.: Aura: A programming language for authorization and audit. In: ICFP (2008)
[19] Jim, T.: SD3: A trust management system with certified evaluation. In: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on. pp. 106–115. IEEE (2001)
[20] Necula, G.C.: Proof-carrying code. In: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 106–119. ACM Press, New York, NY, USA (1997)
[21] Sheard, T., Jones, S.P.: Template meta-programming for haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. Haskell '02, ACM (2002)
[22] Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP. pp. 266–278 (2011)
[23] Vaughan, J., Jia, L., Mazurak, K., Zdancewic, S.: Evidence-based audit. In: CSF'08. pp. 163–176. IEEE (2008)
[24] Wang, R., Chen, S., Wang, X., Qadeer, S.: How to shop for free online - security analysis of cashier-as-a-service based web stores. In: IEEE Symposium on Security and Privacy. pp. 465–480 (2011)
[25] Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols. In: Proceedings of the 1993 IEEE Symposium on Security and Privacy. pp. 178–. SP '93, IEEE Computer Society, Washington, DC, USA (1993), `http://dl.acm.org/citation.cfm?id=882489.884188`