User-Programmable Access Control via Communicating Virtual Assistants

GIOVANNI CAMPAGNA, SILEI XU, RAKESH RAMESH, MICHAEL FISCHER, and MONICA S. LAM, Stanford University

This paper proposes the concept of communicating virtual assistants to enable users share their data in existing web services and IoT devices easily with an unprecedented level of access control.

To provide privacy, we propose a distributed architecture where every user has a virtual assistant that manages his data and credentials to different services, possibly on their own personal devices. Our solution is general, allowing users to share anything that can be performed by a virtual assistant with an extensible set of skills. It is powerful and user friendly, as users can specify with whom, what, when, where, and how the data are to be shared in natural language. It is secure because the controls are translated into a formal language called TACL, which are enforced statically or dynamically by the owner’s own virtual assistant using a provably correct algorithm based on SMT (Satisfiability Modulo Theories). The conforming requests are executed by the owner’s virtual assistant, which sends only the results to the requester.

This paper presents a fully working prototype, based on the open-source Almond virtual assistant whose skill repository contains 177 functions of popular services and devices. We show that users are more willing to share their data given the ability to impose TACL constraints, and that static and dynamic conformance of policies can be enforced efficiently.

1 INTRODUCTION

Today, individuals’ information and capabilities are siloed across many web accounts and IoT devices, from social media accounts, to media subscriptions, file and photo repositories, calendar, bank accounts, health data, personal fitness devices, home security cameras, etc. Unless the service providers support sharing within the app, users wishing to share information must resort to giving out their account credentials, and hence full access to their accounts.

1.1 Communicating Virtual Assistants

This paper proposes the concept of communicating virtual assistants to let users share their data securely at a much finer granularity. Today, a virtual assistant keeps track of all our credentials to our IoTs and web services and lets us access them using a uniform natural language interface. In our proposed model, users give access control policies to their virtual assistants regarding with whom and how their devices and data can be shared. A requester can issue commands to the owner’s assistant, via his own assistant. If the commands conform to the owner’s access control policies, the owner’s assistant will execute them and send back the results.

This design is powerful, as the requester can potentially access anything the owner can, subject to approval. The requester does not have to have an account with the service provider, nor is the sharing limited by what is supported in the service. In the meantime, only need-to-know information is disclosed: a requester receives precisely the approved shared results, and does not gain access to any credentials or additional information.

1.2 User-Programmable Access Control

Access control languages are typically domain-specific languages designed to let sophisticated users or system administrators enforce a set of well-crafted policies for a particular software system. They are important as they govern the security of the system. The requirements of an access control language for virtual assistant tasks are very different from previous languages.
Open-world commands and predicates. A virtual assistant is typically capable of a large number of skills stored in a repository [1, 2, 20]. Richer virtual assistants, such as Almond [14], let users specify, in natural language, compound tasks combining multiple functions together, along with predicates to filter the execution. The scope of the commands is captured by the formal language ThingTalk and the open crowdsourced skill repository called Thingpedia. There are 45 devices in Thingpedia including many popular services and devices such as Twitter, Gmail, Nest and Fitbit, and in total 177 functions. Our goal is to support the full generality of ThingTalk commands, and letting users specify who can execute what combination of functions and predicates, with what kind of input parameters, as well as external factors, such as the location or weather, that are computable from any API in Thingpedia.

Synthesizable from natural language. The users are non-programmers rather than sophisticated professionals. Thus, the access control language also has to be simple, understandable and synthesizable from natural language.

To satisfy the above requirements, we created a policy language called TACL (Thing Access Control Language). TACL is defined as a syntactic superset of ThingTalk to leverage its generality, open-world functionality, power, and synthesizability. TACL can support many everyday life scenarios:

- "Allow my daughter to watch Netflix only before 10pm."
- "Authors can only read those files with their names in the title."
- "Allow colleagues to add Github issues into my to-do list."
- "Whenever I am out of town, let my secretary read email messages whose subject is marked ‘urgent!’"

These examples show access control on a variety of web services, storage systems, IoT devices, according to people, time, content, information flow, or location, for many purposes, such as limiting access, parental control or privacy protection. They can all be enforced by the virtual assistant.

Our system also supports interactive refinement of policies. For example, an Airbnb Guest may ask, “Can I open the front door?” and the home owner can limit the access by telling their assistant, “Yes, but make sure it expires at noon tomorrow.”

1.3 Contributions

This paper proposes the use of communicating virtual assistants to help users share information stored in existing web services and devices easily, securely, and with unprecedented flexibility. The contributions of this paper include:

- The first prototype of communicating virtual assistants, called Comma (Communicating Assistant), which shows the feasibility of supporting sharing on users’ devices without losing ownership. Comma uses the Thingpedia repository [31] which contains 46 different devices and 177 functions between them. It has a federated architecture that lets users keep their data and credentials on their own devices.
- TACL (Thing Access Control Language): A flexible, synthesizable access control language for virtual assistants that supports an open world of constraints.
- An efficient and theoretically sound algorithm, based on Satisfiability Modulo Theories (SMT), to enforce TACL policies statically and dynamically.
- Secure remote execution by informing users precisely of what is executed, by translating requested ThingTalk commands into natural language.
- We evaluate if people are interested in TACL access control, with a survey involving 200 people across a spectrum of 20 different sharing use cases. We find that 27% of the people find sharing more comfortable, on average, if given the power of TACL access control.
- We evaluate the expressiveness of TACL, and find that 90% of the enforceable access control suggested by 60 workers are within the scope of TACL.
- Experimental evaluation of Comma shows that our SMT-based policy enforcement algorithm is efficient.
2 REQUESTS AND ACCESS CONTROL

Here we discuss the languages for describing remote virtual assistant commands and access control. Throughout this paper, we will use the running example in Fig. 1 to illustrate the concepts.

2.1 ThingTalk Overview

ThingTalk was introduced in the Almond virtual assistant [14] as a formal language that lets a user interface with multiple web services and IoTs. It is designed to be synthesizable from natural language. A ThingTalk program has the following syntax:

\[ \text{when}, p_{\text{WHEN}} \Rightarrow \text{get}, p_{\text{GET}} \Rightarrow \text{do} \]

The program connects up to 3 functions: a \text{when} function to say when the action should occur, a \text{get} function to retrieve data, and a \text{do} function to execute an action. For example, the following program continually monitors the user’s Instagram profile for new pictures that have hashtag #cat, and copies them on their Twitter account:

\[
\text{@instagram.new_picture()}, \text{contains}(\text{hashtags}, \#cat), v_0 := \text{url} \\
\Rightarrow \text{noop} \Rightarrow \text{@twitter.post_picture}(\text{url} = v_0, \text{caption} = "\text{cat}")
\]

Input parameters can be passed as keywords to each function, and output parameters can be bound to variables to be used in later functions. Both the \text{when} and the \text{get} function can be qualified with predicates, denoted by \( p_{\text{WHEN}} \) and \( p_{\text{GET}} \) respectively, which operate on the output parameters of the current and the previous function. Information flows from the \text{when} function to the \text{get} function and then finally, to the \text{do} function. The \text{do} function is executed only if \( p_{\text{WHEN}} \) and \( p_{\text{GET}} \) are satisfied. If unspecified, the \text{when} defaults to now, the \text{get} defaults to noop, and the \text{do} defaults to notify, meaning that the results will be returned to the user.

The set of functions that can be used in ThingTalk is defined in Thingpedia [31]. Each function entry includes the API, with its list of input and output parameters and type information, its implementation, and a canonical confirmation sentence. The latter is needed to ensure that the input command was parsed correctly. An example Thingpedia entry for a security camera is shown in Fig. 1(g). The represented function “@security_camera.new_event” has no input parameter and 5 output parameters.
2.2 Remote ThingTalk Commands

We extend ThingTalk to allow commands that can be executed remotely, on another user’s virtual assistant. A (remote) ThingTalk program specifies a source, $\sigma$, who issues the command, and an executor, $\epsilon$, whose assistant is to run the command:

$$\sigma, \epsilon : \text{when}, p_\text{when} \Rightarrow \text{get}, p_\text{get} \Rightarrow \text{do}$$

By default, both the executor and the source are $\text{self}$, i.e. the person defining the program is also the person running it. We add one special $\text{return}$ function, which returns the result to the source instead of the executor.

For example, suppose a dad wants to monitor motion on the security camera for his daughter Alice. He can say in natural language “Ask @alice to let me know when her security camera detects motion” (Fig. 1(a)). His virtual assistant translates it into ThingTalk:

$$\sigma = \text{self}, \epsilon = @\text{alice} : \text{@security\_camera.new\_event()}, \text{has\_motion} = \text{true} \Rightarrow \text{noop} \Rightarrow \text{return}$$

Programs are executed by the virtual assistant of the owner of the services or devices. As shown in Fig. 1(b), the program is sent to Alice’s virtual assistant, which will execute the program only if it conforms to Alice’s access control.

2.3 TACL: ThingTalk Access Control Language

TACL, as a control policy language for ThingTalk, must be at least as expressive as ThingTalk. In addition, we wish to let users impose arbitrary open-world constraints, possibly dynamically, on remote ThingTalk programs. For example, Alice does not want her dad to access the security camera while she is home; she instructs her virtual assistant: “My dad can monitor my security camera only when I am not home”. To enforce such a policy, the virtual assistant needs to check Alice’s GPS location dynamically.

To let users leverage any of the get functions in Thingpedia in their access control, we introduce a get predicate. Syntactically, a get predicate has the form

$$@dn.f\text{n}([pn = v]^\ast)p$$

which invokes function $\text{fn}$ from device $dn$ with parameter bindings $pn = v$, and evaluates a logical expression $p$ that uses any of the variables defined in the current or previous functions in the policy.

Alice’s instruction in natural language is represented formally below in TACL:

$$\sigma = @\text{dad}, \epsilon = \text{self} : \text{@security\_camera.new\_event, @phone.get\_gps(location \neq home)} \Rightarrow \text{noop} \Rightarrow \text{return}$$

Note that words like home and work are tokens that are translated into concrete values after semantic parsing before execution.

A TACL policy is a generalization of a ThingTalk statement:

$$\hat{p}_\sigma, \epsilon = \text{self} : \text{when}, \hat{p}_\text{when} \Rightarrow \text{get}, \hat{p}_\text{get} \Rightarrow \text{do}, \hat{p}_\text{do}$$

Each of when, get, do can be a single Thingpedia function, a wildcard over any function in Thingpedia, denoted _, or a wildcard over any function of a specific Thingpedia device, denoted $@dn._$. $\hat{p}_\sigma$ is a predicate on the source, and can use equality or group membership. In addition, the construct allows three sets of predicates: $\hat{p}_\text{when}$, $\hat{p}_\text{get}$, $\hat{p}_\text{do}$. Whereas ThingTalk filters only on the outputs of when and get functions, here each predicate can place constraints on the input and output parameters of the current function and previous functions. The predicate is a logical expression, whose clauses can be numeric comparisons, array containment, string, and any get predicate from Thingpedia. There is no limit on the number of get predicates in the construct. To keep the predicates deterministic, multiple get predicate invocations of the same function with the same arguments are evaluated only once with each trigger of the when function.

This syntax supports a wide range of access control constraints which can be categorized as: source constraints, function constraints, allowing only the functions specified in the policy, input and output constraints, external
constraints by way of get predicates, and information flow constraints, i.e. forcing the input of a function to be equal to the output of a previous function.

Here is an example of an information flow constraint: "allow @alice or @bob to play anything coming from Netflix on the TV, as long as it is 'G' rated", and this translates to TACL:

\( (\sigma = @alice \mid \sigma = @bob) : \_ \Rightarrow @\text{netflix}\text{.search}, \text{rating} = "G", \nu_0 := \text{url} \Rightarrow @\text{tv}\text{.play_url}, \text{url} = \nu_0 \)

We say that a TACL policy is concrete if it contains no wildcards and all the required input parameters have constant values. We refer to concrete TACL policies as TACL programs, as they have well-defined execution semantics, omitted due to space limitation.

We observe that ThingTalk programs are a subset of TACL programs, as they do not have get predicates. The property that "programs are just very specific policies" brings important benefits. First and foremost, it is easy for the user to understand. The users specify the policies and issue commands in the same way, using natural language, and a similar algorithm can parse programs and policies. Second, the policy language is no less expressive than the programming language itself. Third, an incoming request can be converted into a policy trivially, allowing subsequent similar requests to execute without repeated approvals.

2.4 Runtime Enforcement of Policies

Runtime constraints, such as the current location of the owner, need to be enforced dynamically. Going back to the running example, Alice’s Dad’s request can be made compliant by adding a runtime check that she is not home. The synthesis of Dad’s interest in motion and Alice’s constraint can be expressed as a TACL program.

\[
\begin{align*}
\sigma &= @\text{dad}, \epsilon = @\text{alice}, \\
@\text{security\_camera.new\_event()}, \text{has\_motion} &= \text{true} \land \&( \text{\text{phone}\text{.get\_gps}()}\{\text{location} \neq \text{home}\}) \\
\Rightarrow \text{noop} \Rightarrow \text{return}
\end{align*}
\]

The execution of a TACL program, similar to a ThingTalk program, is shown in Fig. 1(g, h, i); the implementation is retrieved from Thingpedia, the runtime detects the event and sends a message to the source assistant, which notifies the user.

2.5 On-Demand Approval and Access Control

Typically, access control in computing systems needs to be set up ahead of time. The problem is that consumers are unlikely to do so until they need it. Instead of rejecting requests not conforming to existing policies outright, the executor virtual assistant asks the user for approval. The virtual assistant translates the incoming ThingTalk program into its unambiguous natural language representation, letting the user know exactly who the request is from, what functions are being invoked and under which conditions the request can proceed. Since the translation is done by the owner’s assistant, it is guaranteed to match the code.

In our running example, if the virtual assistant finds no usable policies in the database (Fig. 1(c)), it translates Dad’s request as: "Dad wants to get notified when any event is detected on your security camera and has motion is equal to true" (Fig. 1(d)). This sentence is clunky, being deterministically machine generated, but it is understandable and guaranteed to correspond exactly to the program being executed. The user can approve or deny this request and the requester is informed of the same. We additionally let users ignore a request, which suppresses the decision response, and block the source if the user does not wish to be spammed.

The virtual assistant also seizes this opportunity to help users add access control so they need not approve requests one by one. The virtual assistant gives the user options to save the program as a policy that can be executed by just the requester, or by anyone; the user can also supply additional constraints. In the running example, Alice feels that her Dad’s request is too broad and restricts his access to her camera only when she is not home (Fig. 1(e)). This policy is saved in the database for future requests (Fig. 1(f)).
2.6 User Interface

Users are not expected to use ThingTalk or TACL directly; COMMA lets users have a choice of either using natural language or a graphical user interface (GUI) to create remote ThingTalk programs or specify access control policies. As in our running example, Dad can issue the command by natural language to request access to Alice’s security camera (Fig. 2a), whereas Alice can approve the request by clicking the prompt buttons and filling the blanks (Fig. 2b and 2c). Then Dad will get the notification from Alice once an event is triggered after approval (Fig. 2d).

We have developed a neural-network based semantic parser that translates from natural language into TACL, whereas the parser in Almond is based on Sempre [13]. Ours is a sequence-to-sequence neural network [30] parser, modified with an attention mechanism [8, 16, 22] and grammar constraints [34]. Like Almond, we use the paraphrasing technique [33] to generate training data. The results are similar to those obtained in Almond, in that the parser works reasonably well for primitive programs and simple policies, but is not accurate enough to be used for compound programs. Details of this parser are outside the scope of this paper.

3 ENFORCEMENT OF TACL POLICIES

We say that a program is consistent with a policy if run-time checks can be added to enforce the policy. Our enforcement algorithm takes one program and a set of policies, and does the following:

1. If the input program conforms to the disjunction of the policies, run the program unmodified.
2. If a program is consistent with the disjunction of the policies, synthesize an optimal TACL program from the input that enforces policies dynamically, and execute that.
3. Otherwise reject the input program.

3.1 Formal Foundation

Definition 3.1. A policy $\pi_1$ is at least as restrictive as policy $\pi_2$, written as $\pi_1 \preceq \pi_2$, if the source predicate, functions, and predicates of $\pi_1$ are logically entailed by those of $\pi_2$. A constant function is entailed by a wildcard function of the same device, which is entailed by an unrestricted wildcard function.
\[ L[\hat{p}_0 : w \Rightarrow d] \iff L[\hat{p}_0] \land L[w] \land L[g] \land L[d] \]

\[ L[\_] \iff \text{true} \]

\[ L[\forall \nu \equiv w(\nu), \hat{p}] \iff L[\forall \nu] = \overline{\forall w} \land L[\forall \nu] = \overline{\forall w} \land L[\hat{p}] \]

\[ L[\forall \nu = g(\nu), \hat{p}] \iff L[\forall \nu] = \overline{\forall \nu} \land L[\forall \nu] = \overline{\forall \nu} \land L[\hat{p}] \]

\[ L[d(\nu), \hat{p}] \iff L[\forall \nu] = \overline{\forall d} \land L[\hat{p}] \]

\[ L[\forall \nu = f(\nu)[\hat{p}] \iff L[\forall \nu] = \overline{\forall f} (L[\forall \nu]) \land L[\hat{p}] \]

\[ L[\hat{p}_1 \&\& \hat{p}_2] \iff L[\hat{p}_1] \land L[\hat{p}_2] \]

\[ L[\hat{p}_1 \| \hat{p}_2] \iff L[\hat{p}_1] \lor L[\hat{p}_2] \]

\[ L[\neg(\hat{p})] \iff \neg L[\hat{p}] \]

\[ L[\nu \text{ op } v] \iff L[\nu] \text{ op } L[v] \]

\[ L[\text{str\_tr}(\nu, v)] \iff \text{StrContains}(L[\nu], L[v]) \]

\[ L[\text{starts\_with}(\nu, v)] \iff \text{StrPrefixOf}(L[\nu], L[\nu]) \]

\[ L[\text{ends\_with}(\nu, v)] \iff \text{StrSuffixOf}(L[\nu], L[\nu]) \]

\[ L[\text{contains}(\nu, v)] \iff L[v] \in L[\nu] \]

\[ L[\text{in\_group}(\nu, v)] \iff L[v] \in L[\nu] \in \text{GetGroups}(L[\nu]) \]

\[ L[\nu] \iff \nu \]

Fig. 3. Definition of the \( L \) transformation, which maps TACL and ThingTalk syntax to logical formulas. StrContains, StrSuffixOf and StrPrefixOf are predicates in the theory of strings. We omit the rules for literals.

It can be proven that \( \preceq \) is a partial order. The minimum value of this partial order is the \textit{null} policy, whose predicates return a constant \textit{false}. That is, the \textit{null} policy does not allow the return of any value or creating any side effects.

\textbf{Definition 3.2.} A policy \( \pi \) is the \textit{meet} of \( \pi_1 \) and \( \pi_2 \), \( \pi = \pi_1 \land \pi_2 \), if it is the greatest lower bound of \( \pi_1 \) and \( \pi_2 \) under the partial order.

\textbf{Theorem 3.3.} A TACL program conforms to a policy if it is as least as restrictive as the policy.

Our proof, based on a formal denotational semantic of TACL, is omitted due to space.

\textbf{Theorem 3.4.} If a program \( \pi_1 \) is consistent with a policy \( \pi_2 \), then the meet \( \pi = \pi_1 \land \pi_2 \) is not equivalent to the \textit{null} policy. Furthermore, \( \pi \) is the optimal, or least restrictive, TACL program that satisfies the semantics of \( \pi_1 \) and conforms to policy \( \pi_2 \).

\textit{A program is compatible} with the policy, if the source and functions match those in the policy. Compatibility can be checked trivially. Clearly, incompatibility implies inconsistency. We show below how to compute conformance and the meet between a program and a set of compatible policies.

### 3.2 Algorithm

Checking for access control conformance can be reduced to the problem of Satisfiability Modulo Theories [10]. SMT is a generalization of Boolean Satisfiability (SAT) where formulas can include predicates over many domains, such as integers, strings and arrays. Informally, an SMT checker receives a logical formula as input, and returns whether there exists an assignment of the free variables in the formula that makes it true. If such an assignment exists, the formula is \textit{satisfiable}. 


We translate TACL predicates into SMT formulas, and leverage the previous work that has gone into making SMT solvers fast. However, this power comes as a cost: because it is a strict superset of SAT, which is NP-complete, SMT is NP-hard. Nevertheless, we will show in Section 5.5 that empirically it scales well for our problem.

Transforming TACL to SMT. To apply SMT, we define a transformation $L$ from the space of programs and policies to logical formulas. $L$ transforms each predicate in the code into a predicate in SMT, and maps each parameter in the program to a variable in the resulting formula. The precise definition of $L$ is shown in Fig. 3. For clarity, we use positional parameters in the figure, even though TACL uses keyword parameters like ThingTalk.

We unify the namespace of the function parameters in the program and the policy by introducing variables $X_w$ and $X_w$ to mean the input and output to the WHEN function, $X_c$, $Y_c$ for GET and $X_o$ for DO. Other variables are mapped to fresh SMT variables.

For predicates that have an exact correspondence in SMT, such as strings and numbers, $L$ uses the exact SMT equivalent. For group membership predicates, $L$ uses an uninterpreted function that maps each principal to its groups. For every principal that is mentioned in the program, $L$ adds a constraint with the list of groups, which is acquired from the system. A GET predicate that invokes function $f$ is converted to an uninterpreted function $Y_f(x)$, with the same signature as the Thingpedia function.

It holds that if $L\{p\}$ is unsatisfiable, the program will never have any visible side effect. This occurs if the predicates in $p$ are contradictory, and in that case we say $p$ is a null program. For reasons of space, we omit the proof of the correctness of $L$.

Checking conformance. A program $\pi$ conforms to a set of compatible policies $\Pi$ if

$$L\{\pi\} \land \neg \bigvee_{\pi_i \in \Pi} L\{\pi_i\}$$

is unsatisfiable. That is, $\pi$ is more restrictive than the set of policies; there does not exist an instance of $\pi$ that is not covered by the union of the policies. Our conformance algorithm constructs this formula and asks the SMT solver to determine its satisfiability.

For example, if we check the program:

$\sigma = @bob : now \Rightarrow @onedrive.create(name = "bob.txt", txt = "Hi!")$

against the policies:

$\sigma = @alice : _ \Rightarrow @onedrive.create, substr(name, "alice")$

$\sigma = @bob : _ \Rightarrow @onedrive.create, substr(name, "bob")$

the SMT produces the formula

$\sigma = @bob \land X_d.name = \"bob.txt\" \land X_d.txt = \"Hi\" \land$

$\neg ((\sigma = @alice \land StrContains(X_d.name, \"alice\")) \lor (\sigma = @bob \land StrContains(X_d.name, \"bob\")))$

This formula is unsatisfiable, hence the algorithm concludes that the program conforms to the policies.

The meet of a program and policies. Given a program $\pi$ with components when, get, do, $p_{when}$, $p_{get}$, and compatible policies $\pi_i \in \Pi$ with predicates $\hat{p}_{i,\text{WH}}$, $\hat{p}_{i,\text{GE}}$, $\hat{p}_{i,\text{DO}}$, if $\pi$ does not conform statically our algorithm synthesizes program a $\pi'$ as follows:

$\text{when} \ s p_{\text{WHEN}} \ \&\& \ (\hat{p}_{1,\text{WHEN}} \ | \ | \hat{p}_{2,\text{WHEN}} \ldots)$

$\Rightarrow \text{get} \ s p_{\text{GET}} \ \&\& \ ((\hat{p}_{1,\text{WHEN}} \ \&\& \ \hat{p}_{i,\text{GET}} \ \&\& \ \hat{p}_{i,\text{DO}}) \ | \ | (\hat{p}_{2,\text{WHEN}} \ \&\& \ \hat{p}_{2,\text{GET}} \ \&\& \ \hat{p}_{2,\text{DO}}) \ldots)$

$\Rightarrow \text{do}$

This program checks that at least one $\hat{p}_{i,\text{WHEN}}$ of a policy is satisfied, in addition to the $p_{\text{WHEN}}$ of the program. Then it checks that the $\hat{p}_{i,\text{GET}}$ and $\hat{p}_{i,\text{DO}}$ predicate of the same policy are satisfied. This ensures that, for each program...
We preserve privacy with a distributed architecture, where each user can run the assistant on a device of their choice.

**Algorithm 1:** Synthesize a program from an input program to conform to a policy set

**Data:** program $\pi': w, p_{\text{WHEN}} \Rightarrow g, p_{\text{GET}} \Rightarrow d$, a set of compatible policies $\Pi$

**Result:** synthesized program $\pi'$ such that $L\{\pi'\} \models L\{\pi\}$ and $L\{\pi'\} \models \bigvee_{\pi_i \in \Pi} L\{\pi_i\}$

if $\Pi = \emptyset$ then return null // $\pi$ has no compatible policies
if $\neg \text{sat}(L\{\pi\})$ then return null // $\pi$ is null
if $\neg \text{sat}(L\{\pi\} \wedge \neg \bigvee_{\pi_i \in \Pi} L\{\pi_i\})$ then return $\pi$ // $\pi$ is conforming

$p'_{\text{WHEN}} \leftarrow$ false
$p'_{\text{GET}} \leftarrow$ false

for policy $\pi_i \in \Pi$ of the form $w, p_{\text{WHEN}}, \pi_i \Rightarrow g$, $p_{\text{GET}}, \pi_i \Rightarrow d$, $p_{\text{DO}}, \pi_i$, do

// check whether $\pi_i$ is relevant
if $\text{sat}(L\{\pi\} \wedge L\{p_{\text{WHEN}}, \pi_i\} \wedge L\{p_{\text{GET}}, \pi_i\} \wedge L\{p_{\text{DO}}, \pi_i\})$ then

// check whether $p_{\text{WHEN}}, \pi_i$ is redundant
if $\neg \text{sat}(L\{\pi\} \wedge \neg L\{p_{\text{WHEN}}, \pi_i\})$ then

$p'_{\text{WHEN}} \leftarrow$ true
$p_{\text{WHEN}}, \pi_i \leftarrow$ true

end
else $p'_{\text{WHEN}} \leftarrow p_{\text{WHEN}} || p_{\text{WHEN}}, \pi_i$

// check whether $p_{\text{GET}}, \pi_i$ and $p_{\text{DO}}, \pi_i$ are redundant
if $\neg \text{sat}(L\{\pi\} \wedge L\{p_{\text{WHEN}}, \pi_i\} \wedge \neg (L\{p_{\text{GET}}, \pi_i\} \wedge L\{p_{\text{DO}}, \pi_i\}))$ then $p'_{\text{GET}} \leftarrow p_{\text{GET}} || p_{\text{WHEN}}, \pi_i$
else $p'_{\text{GET}} \leftarrow p_{\text{GET}} || (p_{\text{WHEN}}, \pi_i \& \& p_{\text{GET}}, \pi_i \& \& p_{\text{DO}}, \pi_i)$

end

$p'_{\text{WHEN}} \leftarrow \text{Simplify}(p'_{\text{WHEN}})$
$p'_{\text{GET}} \leftarrow \text{Simplify}(p'_{\text{GET}})$

$\pi' \leftarrow w, p_{\text{WHEN}} \& \& p'_{\text{WHEN}} \Rightarrow g, p_{\text{GET}} \& \& p'_{\text{GET}} \Rightarrow d$

if $p'_{\text{WHEN}} = \text{false} \lor p'_{\text{GET}} = \text{false}$ then

return null // $\pi$ is inconsistent
end

return $\pi'$ // $\pi$ is consistent

execution, all three predicates are satisfied at the same time for at least one of the policies. The policy satisfied may be different for each execution.

We can prove that $L\{\pi'\} \models L\{\pi\}$ and $L\{\pi'\} \models \bigvee_{\pi_i \in \Pi} L\{\pi_i\}$, and that $\pi' = \pi \land \bigvee_{\pi_i \in \Pi} \pi_i$, by extending the definition of restriction to cover a disjunction of policies in the obvious way. If $\pi'$ is not null, $\pi$ is consistent, and $\pi'$ is executed. Otherwise, $\pi$ is rejected.

Algorithm 1 shows the full algorithm. To reduce the number of predicates in the synthesized program, the algorithm constructs the program incrementally, adding one policy at a time. Policies that conflict with the predicates in the original program are not relevant and are omitted. Predicates that are entailed by the original program or by a previous predicate in the same policy are replaced with true. The algorithm calls `Simplify`, omitted due to space, to simplify logical expressions using standard techniques.

4 A DISTRIBUTED SYSTEM FOR TACL

We preserve privacy with a distributed architecture, where each user can run the assistant on a device of their choice.
4.1 Naming and Messaging

We let users refer to people they know using known identities, such as email addresses and phone numbers. To ensure security, our communication protocol is implemented on top of a generic messaging service, which is responsible for mapping real-life identities to the messaging accounts, and sending messages securely between the accounts. The messenger verifies all the real-life identities before they can be associated with the account, for example by sending a validation code via SMS to verify a phone number. We assume that messages to the same account are delivered in order.

4.2 Data Communication Between Assistants

A ThingTalk program uses the `return` function to indicate that the source should be notified of the results, rather than the executor. The source virtual assistant rewrites the program

\[
\sigma = u_1, \epsilon = u_2 : \text{when } \Rightarrow \text{get } \Rightarrow \text{return}
\]

as a pair of ThingTalk programs:

\[
\sigma = u_1, \epsilon = u_2 : \text{when } \Rightarrow \text{get } \Rightarrow \text{send}(\text{to} = u_1, \text{flow} = f) \Rightarrow \text{notify}
\]

\[
\sigma = u_1, \epsilon = u_1 : \text{receive}(\text{from} = u_2, \text{flow} = f) \Rightarrow \text{notify}
\]

where send and receive are communicating functions between two virtual assistants. The two functions are connected by a flow, a unique identifier that pairs them. The definitions of communicating functions are entered into Thingpedia, no different from any other APIs. The general ThingTalk implementation can execute these operations without requiring any modification.

Both programs are given the same unique program identifier so that they can be stopped at the same time. The former program is sent to the executor for approval and execution; the latter is executed by the source upon having the program approved.

4.3 Program Execution

To execute a program, the source virtual assistant sends an Install message to the executor, containing: (1) program, the code to execute; (2) identity, the identity to use in the permission request; (3) progid, the unique identifier of the program.

The executor virtual assistant verifies that the identity claimed in message corresponds to the sender. If the program is consistent with the policy, the assistant runs the compliant program returned by Algorithm 1, otherwise the user is asked to approve. If the program is denied, the assistant replies with an Abort(progid, reason).

If the program uses the send function, the assistant sends a message of type Data(flow, payload) for each result. This message is routed to the corresponding receive function and triggers the notify action on the source side, showing the result to the user. When the program on the executor terminates, the executor signals the end of data messages via an End(flow) message. Upon receiving an End message, and after processing all previous Data messages, the source assistant stops the receive program.

At any point, either the source or the executor can stop the programs using an Abort(progid, reason) message. Upon receiving an Abort message, the assistant stops the program immediately. The protocol offers no guarantee that any action that the executor might have started will or will not be executed.

4.4 Choice of Messaging Protocol

Our distributed ThingTalk implementation uses the Matrix [3] messaging protocol. This protocol supports the requirements of the ThingTalk distributed system: it allows authentication based on well-known identities, it supports arbitrary payloads with end-to-end encryption, and it has reliable delivery in the face of network issues and long-time disconnection of one party.
Matrix exchanges messages (datagrams) of up to 64K in size. The header size of the ThingTalk protocol message is 80 bytes. For Data messages, only the raw values returned by the Thingpedia function are exchanged, and large objects such as pictures are passed by URL, thus each message is unlikely to be more than a few KBs. The size of an Install message depends on the size of the program, and in turn on the number of predicates. In our tests, more than 50% of the programs can be serialized in under 300 bytes. In the worst case scenario of an automatically generated program with 65 predicates, the program size is 1884 bytes, which indicates the protocol is sufficient for our use case. In our measurements, Matrix incurs a communication latency of at most 600 milliseconds, and that latency varies minimally with the payload size.

4.5 Security Considerations

Access to private user information makes Comma a security-sensitive system. In this section, we describe how the design protects users from known possible attacks.

Trust assumptions. We trust that there are no code errors in the assistant, and Thingpedia functions behave according to their metadata. First, the entirety of Comma, including the messaging client code, is open-source and thus subjected to public scrutiny. Furthermore, our design accepts only a small number of message types, and is implemented in a safe dynamic language, which reduces the surface area for attacks. Communication with Thingpedia and the Comma natural language service occurs through a secure channel (HTTPS). Finally, Thingpedia entries need to be approved by the administrators of Thingpedia for general use.

Phishing and Impersonation. Comma relies on the local address book and user knowledge of phone numbers and emails. Furthermore, all identities are validated through the messaging layer. We trust that the identities are not stolen, and we trust the verification provided by the messenger. In the future, this assumption could be lifted with an initial phase to verify keys, as customarily done by end-to-end encrypted messaging apps [15].

Remote Code Execution. Executing code received from an external source is extremely dangerous. Comma can execute programs written only in ThingTalk, which has a single control construct. The attacker can control the code, but they cannot control the approval request that is generated based on the trusted Thingpedia entries. This approval is explicit and unambiguous, mentioning all parts of the ThingTalk program, therefore the attacker cannot go undetected when doing something malicious. The system also strips any description or display name embedded in the ThingTalk code, so that no part of the confirmation is under the attacker’s control.

5 EXPERIMENTATION

We have created a fully working prototype of the concepts presented above, called Comma, on top of the open-source Almond platform. We implemented an Android version, a command line version and a web version.

Here we present several studies to address the following questions about access control and its conformance. We first conduct a small in-person study to see how users respond to Comma. We then broaden the study to people online focusing on access control. Do people have a need for access control? For what services and devices is access control useful? Is TACL expressive enough for an open-world of services and devices? And lastly, we explore if our policy conformance algorithm is efficient enough to handle many policies.

5.1 A User Study on Comma

First, we conduct an in-person user study to understand how users would react to our Comma prototype. We recruit 10 users in our study from a university; 7 are students and 3 are staff members. We show them two tablets running the Comma app, one acting as a requester and one as a grantor. Then each user is asked to complete the 5 scenarios shown in Fig. 4, in which they perform the request and then approve it with constraints, through the Comma user interface. They get to see the full experience, including the actual effects of the tasks performed.
Fig. 4. Five scenarios used in the user study.

The users completed the scenarios successfully without intervention, except in 9 instances. The problems arise mainly from missing error flows in the user interface, which can be addressed accordingly. Most users find the interface intuitive, and all but two of them prefer it over a typing or voice interface. Overall, all users like the concept with an average rating 4.4 out of 5. Only 1 user said they would not use it at all, despite liking the concept.

5.2 The Need of Access Control

To understand if people need fine-grained access control, we conducted a survey with 20 use cases. Each use case starts with a baseline policy with only role-based access control, i.e., giving out full permission based on the role of users. We then provide two examples of attribute-based access control. We ask the user to rate the comfort level of each use case, with and without the additional constraints. We use a five-level Likert scale, labeled with: "very uncomfortable", "uncomfortable", "neutral", "comfortable", "very comfortable". For example, we first ask if they would "allow your 10-year-old kid to use your Netflix account while you are not home". Then we ask the same but with the constraints that "only between 7 and 9 pm" and "only free G or PG rated movies".

The survey was conducted with the help of 200 Amazon Mechanical Turk workers. We summarize the result by reporting on the percentage of people who find the use case "comfortable" or "very comfortable". The results, ordered in increasing percentage for the baseline, are shown in Fig. 6. The bottom dark blue bar shows the baseline result, and the top light blue bar shows the increased percentage with the introduction of constraints.

First, we observe that the 20 use cases cover a wide spectrum of comfort level for sharing; people comfortable with sharing range from 7% to 70% in the baseline cases. The highest comfort rates are observed when the requester is highly trusted, such as a doctor or a significant other, as in scenarios 19 and 20. When allowed finer-grained access control, more people find sharing comfortable in every one of the use cases. Analyzing these scenarios suggests the reasons why access control makes people more comfortable to share, as discussed below.

- **Privacy and need-to-know for information.** Of the 20 cases, 11 of them request personal information, sorted in increasing order of comfort: the PC screen, the to-do list, Fitbit information, photos, emails, security cameras, a dog’s location, SMS, Spotify playlists, blood pressure, and current location. The grantors are less motivated to share privacy-sensitive information, except with trusted individuals. Limiting the information shared to what is beneficial and what is needed makes people more willing to share. For example, seeing the PC screen would reveal a lot of information about the grantor and following the dog would often reveal the grantor’s location. The grantor becomes willing to share the information only if the requester absolutely needs it, e.g., to provide IT support, or to help locate a missing dog.

- **Liability and need-to-act for actions.** 9 of our scenarios involve granting rights to accounts to perform actions, from UberEat, to credit cards, house doors, cars, Amazon accounts, Instagram, calendars, Netflix, thermostats. The responses suggest that people are more cautious with granting actions. The risk goes beyond losing privacy; it may have consequences on finances (UberEat, credit cards, house doors, cars, Amazon accounts, thermostats), safety (driving a car), or one’s image or reputation (posting on Instagram).
People are interested in using access control to limit the liability (e.g. setting a budget), and to increase the benefits (e.g. value of the delivered package). In addition, same as the need to know, people want to limit the access based on the need to act. As in the AirBnb guests’ request for access to the thermostat in scenario 18, people would rather risk a run-away electricity bill than to be contacted; yet, they still appreciate the ability to put limits on it.

<table>
<thead>
<tr>
<th>#</th>
<th>Role-based access control</th>
<th>Fine-grained access control</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allow others to see your PC screen online</td>
<td>only while you are gaming</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only if you are asking for IT support to solve a problem on your computer</td>
</tr>
<tr>
<td>2</td>
<td>Allow your roommate to order food with your UberEat account</td>
<td>with a $20 budget limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only to your current location</td>
</tr>
<tr>
<td>3</td>
<td>Allow your teenager daughter to access your credit card</td>
<td>with a $20 budget limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for restaurants only</td>
</tr>
<tr>
<td>4</td>
<td>Allow your colleagues to access your to-do list</td>
<td>only add to-dos labeled with &quot;work&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only see to-dos labeled with &quot;work&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Allow an Amazon courier to unlock your door and leave the package inside</td>
<td>only if the package is worth more than $1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only when your security camera is on</td>
</tr>
<tr>
<td>6</td>
<td>Allow your 17-year-old son to drive your Tesla</td>
<td>with its speed limited to less than 50 mph</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only between school and home</td>
</tr>
<tr>
<td>7</td>
<td>Allow your teenage son to access your Amazon account to make purchases</td>
<td>with a $20 budget limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only things you’ve put in your wish list or cart</td>
</tr>
<tr>
<td>8</td>
<td>Allow your friends to access your Fitbit account to see your activities</td>
<td>only see your steps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only see your steps when they are above 10000</td>
</tr>
<tr>
<td>9</td>
<td>Allow your friends to post photos on your Instagram</td>
<td>only photos with both of you in them</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only pictures of memes</td>
</tr>
<tr>
<td>10</td>
<td>Allow your friends to have access to your cloud drive to view/download photos</td>
<td>only photos with their faces in them</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only photos in a specific folder</td>
</tr>
<tr>
<td>11</td>
<td>Allow your secretary to read your emails</td>
<td>only emails with a certain label or subject you defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only when you are on vacation</td>
</tr>
<tr>
<td>12</td>
<td>Allow others to add events to your calendar</td>
<td>only events during working hours</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only who has an email account from a certain domain</td>
</tr>
<tr>
<td>13</td>
<td>Allow your parents or kids to have access to security cameras in your house</td>
<td>only if you are not at home</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only those cameras facing the front yard or the garage</td>
</tr>
<tr>
<td>14</td>
<td>Allow your friends to access your dog tracker to see your dog’s location</td>
<td>only when you and your dog are not at the same location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only if you lost your dog</td>
</tr>
<tr>
<td>15</td>
<td>Allow your significant other to read your sms</td>
<td>except messages between you and certain people you defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only if the messages come from a short phone number (e.g., UPS notification)</td>
</tr>
<tr>
<td>16</td>
<td>Allow your 10-year-old kid to use your Netflix account while you are not home</td>
<td>only between 7 PM to 9 PM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only free G or PG rated movies</td>
</tr>
<tr>
<td>17</td>
<td>Allow your friends to access your Spotify playlists</td>
<td>they can only view but cannot edit the playlists</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only playlists you marked as public</td>
</tr>
<tr>
<td>18</td>
<td>Allow your Airbnb guests to control the thermostats in the room</td>
<td>only within a temperature range</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only when they are inside the room</td>
</tr>
<tr>
<td>19</td>
<td>Allow your doctor to monitor your blood pressure from a smart device</td>
<td>only at 8 AM every morning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only when your blood pressure goes higher than normal</td>
</tr>
<tr>
<td>20</td>
<td>Allow your significant other to have access to your current location</td>
<td>only when you are driving to pick him or her up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only when you are near a Walmart so he or she can remind you what needs to be bought</td>
</tr>
</tbody>
</table>

Fig. 5. 20 sharing use cases in the survey
These results suggest that the general public is willing to share more if given fine-grained access control. On average, an additional 27% of the population becomes comfortable to share, with an increase of 40% of the population in 4 cases. While originally only 4 scenarios were found comfortable by 50% or more of the users, the number increases to 11 with constraints. Of the remaining 16 scenarios, the percentage of people finding it comfortable doubles in 11 cases.

### 5.3 The Diversity of Access Control

To better understand the real-world uses of TACL, we conduct another Mechanical Turk experiment to learn what people would use such a system for. We present each worker with three example use cases where a user puts constraints on a requested access, and ask them to suggest new use cases. In total, we ask 60 workers and collect 220 valid use cases.

The suggested use cases request access to 85 unique devices and services, which cover a large variety of situations, as shown below. The number of use cases is shown in parentheses.

- **IoT devices (59):** GPS, phone, computer, smart lock, shutter, thermostat, light, smart TV, gaming console, appliances, car.
- **Personal data (73):** messages, emails, contact list, calendar, cloud storage, financial data (bank statements, credit score, financial report, tax forms), driving records.
- **Social media (35):** Instagram, Facebook, Twitter, Snapchat, Flickr, Pinterest, Reddit, Steam, ClassDojo.
- **Services (46):** video and music streaming, Uber, TheKnot, online stores (Amazon, eBay), online recipes, bill payment.
- **Business accounts (2):** developer account.
- **Non-smart physical devices (5):** bike, pencil, and credit card.

Among the 220 use cases we collected, most of the constraints used fall into the following 5 categories, whose representative examples are shown in Fig. 7:

- **Function constraint (220):** Each device or service may have many functions; e.g. Twitter allows people to send tweets, read tweets, send direct messages, etc. A function constraint limits the functions a requester can use. All the use cases we collected have a function constraint; 70 of them have no other constraint.
- **Input constraint (24):** Only certain values are allowed as input parameters to a function; e.g. the recipient of messages.
- **Output constraint (74):** Only outputs satisfying the constraint are shown to the requester; e.g. only emails from a certain sender.
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Examples</th>
<th>Device/Service Type</th>
</tr>
</thead>
</table>
| Function  | Brother: Let me access your calendar to add a reminder so that you won’t forget our parents anniversary.  
Me: Okay, but only add notes, not read my other events.  
Wife: I need to send mortgage documents but they need to be sent by you can I have access to your email  
Husband: Yes, but only to draft and send a message | Personal data |
| Input     | Daughter: Mom, can I send a text to grandma about this weekend?  
Mom: Sure, but only text grandma and no one else.  
Mom: You need to follow this guy on twitter, give me your twitter account.  
Me: OK, add him but don’t follow any other twitter user. | IoT device |
| Output    | Friend: Can I access non-shared files on your Google Drive so I can read some books?  
Me: Yes, but only the PDF and ebook files.  
 Trainer: Can I access your Health Diary app to keep up with your well being during your training period?  
Me: Yes, but not the sections tracking my psychological states and moods. | Social media |
| External  | Tenant: I believe there may be a hurricane coming soon. Can I put down the window shutters?  
Owner: Yes, but only if it’s a Category 3 or above.  
Mother: I need to come into your room so I can clean it.  
Me: Sure, but only if I’m there to help you. | IoT device |
| Aggregate | Friend: I forgot my gmail account can I have your gmail password to send a mail?  
Me: yes, but only send one email.  
Son: Mom I need to use your uber account until I receive my new phone.  
Mom: Ok but use 4 rides max. | Service |

Fig. 7. Representative use cases by Mechanical Turk workers.

- **External constraint** (25): The execution of the command depends on external conditions, such as time, location, and weather.
- **Aggregate constraint** (16): This type of constraints aggregates over multiple invocations of the program. Examples include the frequency of execution or the sum of outputs across invocations.

This study suggests that the concept of access control is widely applicable to diverse assets in real life.

5.4 Applicability of TACL

The people answering the survey are drawn from a general population. We deliberately provided no guidance on the scope of allowable constraints to find out what constraints they would find useful. In fact, the workers thought that they would simply be asking the requesters to honor the constraints, not expecting them to be enforced. For example, one suggested constraint is "allow the use of my library card only if the book will be returned on time", which is of course not enforceable. Surprisingly, the majority of the 220 use cases fall within the scope of TACL, as shown below:

- Within the scope of TACL, with existing APIs (70%). Of the 153 cases in this category, 35 of them are already available in Thingpedia, and the rest can be supported by simply extending Thingpedia with the publicly available APIs.
Within the scope of TACL but requires new APIs (15%). There are 34 use cases in this category. For example, people want to limit access to specific functions on their laptop or their phone, such as making a call. Such APIs do not currently exist; hardware manufacturers may want to add the suggested functionality in the future.

Limitations of TACL (9%). TACL currently does not support aggregate constraints, thus cannot handle 16 cases. The remaining 3 use cases require information about the requester, which cannot be implemented by the owner’s assistant.

Unenforceable constraints (6%). The remaining 14 cases are not enforceable because they use non-smart devices or they rely on the requester keeping his promise.

This study showed that TACL covers 90% of the 206 enforceable cases suggested by our workers, provided that the APIs are available. Note that the workers are unaware of the expressiveness of TACL. Therefore, it suggests that TACL has a good coverage of the kind of access control that laymen are interested in.

5.5 Policy Conformance Evaluation

Because SMT is NP-hard, it can require exponential time in the worst case, but it has been shown to be fast enough for many tasks. Here we evaluate if our SMT-based conformance algorithm can handle large programs and many access control policies. Our experiment is conducted using the CVC4 SMT checker [9], with all supported quantifier-free theories enabled.

5.5.1 Test Suite. For this experiment, we generate 4,000 programs from the 48 device classes and 192 functions in Thingpedia. To avoid oversampling devices that have more APIs, we sample in order the type of functions (when, get, do) in the program, then the devices, and then the functions. Predicates are generated in conjunctive normal form, with the number of ‘and’ and ‘or’ clauses chosen by a geometric distribution with parameter 0.4. This keeps the median low but still generates a long tail of programs with many predicates. Predicate operators and constants are chosen uniformly. For input parameters, with fixed probability we choose either a variable in scope, a constant, or leave the value unspecified. The median number of predicates was 3, while the maximum was 65. These unrealistically large numbers of predicates are useful for evaluating the worst case behavior of the algorithm.

As discussed in Section 3.2, the cost of the conformance for a program depends primarily on the number of its compatible policies in the database. We generate compatible policies also randomly for each given program. We generate at most one wildcard function in each policy to make the problem harder but still realistic. The policy uses randomly generated predicates based on the function parameters, with a geometric distribution with parameter 0.5.

We run 4 experiments, where we try to generate 1, 5, 10, 50 compatible policies, respectively. We expect a program to have no more than 5 compatible policies in practice; we experiment with 50 compatible policies to understand scalability. The number of possible compatible policies depends on the number of parameters. We succeed in generating 5 compatible policies for 3786 of the 4000 programs, 10 for 3282 programs, and 50 for 1067 programs. The policy sets are generated independently for each experiment.
5.5.2 Conformance of Test Suite. We run the test suite through our policy conformance algorithm, and obtain the results shown in Fig. 8. First, 369 of the 4000 programs are null, meaning that the predicates in the program are inconsistent. Note the null programs are an artifact from random program generation; our conformance algorithm detects the inconsistency and disregards them.

As expected, the probability of getting rejected by the policies goes down as the number of policies increases. A rejection happens only if the input parameters or predicates in the program conflict with every policy in the set. We see a rejection rate of 27% when there is only one compatible policy. Otherwise, the rejection rate approaches 0 even with 5 randomly generated policies. 48% of the programs are consistent for sets with 10 or less policies, whereas 75% are consistent for sets with 50 policies.

5.5.3 Speed of the algorithm. We now study the speed of the algorithm. All our tests are conducted on a 6-core Intel Xeon CPU @ 2.50GHz, with 80 GB of RAM. Null programs are identified in less than 47 ms, and we omit them from the statistics shown below.

Our first experiment measures how long it takes to determine if a program conforms, without requiring the addition of run-time checks. Fig. 9a shows how the average conformance testing time varies with the number of predicates in the input program and the number of policies in the set. We only report averages for up to 20 predicates, since beyond that there are not enough samples.

The averages for each test case increase slowly, showing SMT is effective at analyzing even large formulas. The spikes and irregularities in the graph can be attributed to string operations being more expensive than numbers; programs with similar number of predicates use the same Thingpedia functions and types.

The increase in time as the number of policies increases is modest. Specifically, the slowdown is less than 2x from 5 to 10 policies, and less than 10x from 5 to 50, which suggests that our algorithm scales better than the exponential worst-case bound.

The conformance algorithm runs in less than 400ms for all programs, as shown in the cumulative density function in Fig. 9b. In fact, the SMT algorithm runs quickly, within 48ms, if the program is conforming. It only needs to find a policy that entails the predicates in the program. To conclude that the program is not conforming, however, the SMT algorithm needs to construct a case that violates all the policies. This effect can be easily observed in the test suite with 50 policies; the increase in execution time occurs at around 18%, coinciding with the number of programs that are conforming.

For programs not conforming to the policies, our algorithm attempts to synthesize a restricted program that conforms. With this additional step, the algorithm runs in less than 800 ms for programs with 10 or less compatible policies, and up to 3 seconds for 50 compatible policies (Fig. 9c). From Fig. 8, we see that the synthesis step is
needed for about 50% of programs with 10 or less policies, and about 80% for 50 policies. This explains the marked increase in run time observed. It is unlikely we can find as many as 50 compatible policies for a given input program, thus our results suggest that our algorithm is practical.

5.6 Discussion

Our survey suggests that many people recognize the need for access control and that access control has broad applicability over many different services and devices. However, adoption of existing access control platforms is limited. Previous work [23, 25, 28] has shown that authoring access control policies in a traditional user interface can be cumbersome and error prone.

Our proposal may have a higher chance of adoption than previous solution for several reasons. Virtual assistants are a commercial success [1, 2, 20], and many users will have a virtual assistant that has access to their credentials already. For generality, we are leveraging the same repository of device APIs, used by virtual assistants, albeit that we require a modest amount of extensions to support remote execution and access controls. The natural language interface will continue to improve, making it easy for users to express access controls.

Furthermore, Comma supports on-demand approvals and access control specifications. The requester can request access without first asking users for permission. Smetters et al. [28] in particular highlight that users prefer contextual access control policies, rather than policies based on the object being accessed, and Bauer et al. [11] report that on-demand approval is at the top 3 of desired functionality in access control, together with logging and notification. In Comma, by using the on-demand approval interface, the user is shown the most context and is able to make informed decisions at the time of access rather than ahead-of-time.

6 RELATED WORK

6.1 Access Control Systems

In research, there are two major classes of access control systems: Role-based Access Control (RBAC) [26] and Attribute-based Access Control (ABAC) [36]. RBAC restricts access by assigning users to roles, and defining the privileges of each role; ABAC expresses access as boolean predicate based on the user, resource, object, environment. Recently, many hybrid access control systems have been proposed [18, 19, 21]. This is because the number of roles in RBAC explodes with the size of the administrative domain, while ABAC policies become very complex as the predicate grows. TACL is intrinsically an ABAC language, with minimal support for RBAC-like policies through the use of groups. On top of that, TACL policies can be created collaboratively and on-demand, which reduces the cost of setting up the policy.

The most popular language for ABAC policies is XACML [24]. In XACML, a request is intended as a single function call with constant parameters. The policy can specify constraints on the parameters and on attributes such as time and user information. SMT solvers have been successfully used to check various properties of XACML policies [5, 6, 32], such as disjointness and conflict. Conformance in XACML is trivial, because the values of the parameters are known at verification time and can be substituted in the policy. On the other hand, conformance in TACL needs SMT because the input is a full program with predicates. TACL also supports runtime enforcement when necessary. Arkoudas et al. [4] proposes an algorithm to modify access control requests to be policy-conforming, based on a manually defined optimality condition. This is a different synthesis problem than the TACL one, because TACL modifies the program by restricting the execution while preserving the intent.

6.2 Policy Languages for Programs

There are several languages proposed in the literature [7, 12, 35] that express policies on the behavior of programs, rather than single functions. Their purposes are code inspection, bug finding and tracking leaks of secured data. These languages let users add arbitrary code to the programs. TACL differs from these languages because
it is higher level and end-user programmable. Traditional policies are written by programmers for a specific application, while TACL policies can be written by end users for an open world of tasks.

6.3 Virtual Assistants

Commercial virtual assistants like Google Home and Alexa provide very limited multi-user support. They use voice identities to associate users with different accounts on a single speaker device; they also allow restricting the use of certain skills with a PIN. Users cannot restrict actions nor control their data.

Comma is built on top of the open Almond virtual assistant [14] platform. Almond does not support multi-user interactions or access control. Comma extends Almond in several major ways. It supports distributed execution across devices owned by multiple parties, with flexible access control. The TACL language and access control conformance are new and introduced only in Comma. The Almond execution engine is augmented with a distributed protocol implementation. The Thingpedia specification is also made richer in Comma, by adding metadata for remote execution.

6.4 Peer-To-Peer Data Sharing

Many systems [17, 27, 29] offer the ability to share personal data in a peer-to-peer fashion. Contrail [29] is a federated social network with a pub-sub model. Users can add filters on the subscribed data, and evaluation occurs on the publisher nodes. Unlike in Comma, their filters are expressed in a full programming language, which cannot be analyzed like ThingTalk. Contrail has no access control mechanism to limit how the data is disseminated, making it suitable only for public social networks. The Prpl system [27] is a federated architecture where brokers mediate queries to existing data sources, such as Facebook or email. Prpl is not end-user programmable: users interact with a specific application that makes use of Prpl. Additionally, Prpl’s access control is rudimentary and static: users can only enforce read or write access to whole resources.

7 CONCLUSION

With the rise of the virtual assistant, individuals have a smart agent that holds all our credentials and can carry out natural language commands. We see this as an opportunity to dramatically improve how users share data and devices. We propose that the owner’s virtual assistant carry out commands on behalf of requesters and only share the need-to-know results with the requesters. The execution of the requests is made secure by having the owner’s assistant translate the command into natural language for approval. With the help of a distributed architecture that lets users run their virtual assistant on their own devices, users can share a wide range of assets, without disclosing information to any third party.

Moreover, users can use natural language to assert fine-grained and flexible access control over all their digital assets, with the help of the proposed TACL language. The user can even constrain an execution with external conditions derived from an open-world of virtual assistant skills. Our efficient and general SMT-based algorithm can enforce the access control statically and dynamically.

Our user study shows that the general public finds a need for access control and the majority of enforceable access controls of interest can be expressed in TACL. In the past, users were at the mercy of the service providers on how their data can be shared. Here, users can share any device or information available to their virtual assistants easily, exactly, and securely according to their preference.

REFERENCES


