Controlling Fine-Grain Sharing in Natural Language with a Virtual Assistant

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

This paper proposes a novel approach to let consumers share data from their existing web accounts and devices easily, securely, and with fine granularity of control. Our proposal is to have our personal virtual assistant be responsible for sharing our digital assets. The owner can specify fine-grain access control in natural language; the virtual assistant executes access requests on behalf of the requesters and returns the results, if the requests conform to the owner's access control policies.

Specifically, we allow a virtual assistant to share any ThingTalk command–an event-driven task composed of skills drawn from Thingpedia, a crowdsourced repository with over 200 functions currently. Access control in natural language is translated into TACL, a formal language we introduce to let users express for whom, what, when, where, and how ThingTalk commands can be executed. TACL policies are in turn translated into SMT (Satisfiability Modulo Theories) formulas and enforced using a provably correct algorithm. Our Remote ThingTalk Protocol lets users access their own and others' data through their own virtual assistant, while enabling sharing without disclosing information to a third party.

The proposed ideas have been incorporated and released in the open-source Almond virtual assistant. 18 of the 20 users in a study say that they like the concept proposed, and 14 like the prototype. We show that users are more willing to share their data given the ability to impose TACL constraints, that 90% of enforceable use cases suggested by 60 users are supported by TACL, and that static and dynamic conformance of policies can be enforced efficiently.

CCS Concepts: • **Human-centered computing** → **Personal digital assistants**; • **Security and privacy** → *Access control*; *Usability in security and privacy*; • **Computing methodologies** → Distributed programming languages;

Additional Key Words and Phrases: natural language interfaces, usable security, Web APIs, Internet of Things, remote program execution

ACM Reference Format:
Giovanni Campagna, Silei Xu, Rakesh Ramesh, Michael Fischer, and Monica S. Lam. 2018. Controlling Fine-Grain Sharing in Natural Language with a Virtual Assistant. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 3 (September 2018), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Today, consumers have a tremendous amount of digital information and capabilities, 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. We have a need to share these data and capabilities with other users easily, while controlling their access. Unfortunately, if we wish to share beyond the options provided by the service provider, we need to give out our account credentials, and hence full access to our accounts.

Authors' address: Giovanni Campagna; Silei Xu; Rakesh Ramesh; Michael Fischer; Monica S. Lam, Stanford University, Computer Science Department, 353 Serra Mall, Stanford, CA, 94305, USA, {gcampagn,silei,rakeshr,mfischer,lam}@cs.stanford.edu.

2474-9567/2018/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The goal of this research is to let consumers share *anything they can access with public APIs* and control *who, what, when, where, and how* the data is to be shared. The technique must be *secure*, but at the same time, *easy to use* by ordinary people. Our typical user cannot handle sophisticated access control languages originally designed for IT professionals. We want to let users just say what they want, in natural language, in everyday scenarios:

"*Allow my daughter to watch Netflix only before 8pm.*"
"*Allow my son to purchase any household item under $10 on Amazon.*"
"*My dad can access my security camera, only when I am not home.*"
"*Whenever I am out of town, let my secretary read email messages whose subject is marked 'urgent'.*"
"*Allow colleagues to add GitHub issues to my to-do list.*"
"*Authors can only read those files with their names in the title.*"

These examples show a need for 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 parental control, privacy protection, minimizing capability shared, and information disclosed. Typically, access control in computing systems needs to be set up ahead of time; consumers are unlikely to do so. It is desirable for consumers to respond to requests by specifying policies on the fly and saving them for subsequent requests. For example, an Airbnb Guest may ask, "*Can I open the front door?*" and the home owner can respond with "*Yes, but make sure the access expires at noon tomorrow.*" This illustrates that specific and seemingly detailed access controls can be specified naturally in an easily understandable context through an interactive dialog. In summary, we need our access control to be general, expressive, extensible, secure, and easy to use.

1.1 Sharing via a Virtual Assistant

The adoption of virtual assistants, such as Alexa, Google Assistant, Siri, and Cortana, has grown rapidly in recent years. Virtual assistants are the ideal agent to help consumers share their information with access control. They already have the users' credentials and the ability to perform all the skills in the platform. Also, they support a natural language interface, which we can extend to incorporate user-specified access control.

Our proposal is to have all sharing requests be sent to the owner's virtual assistant which, upon the approval of the owner, will perform the requested commands on behalf of the requester and return the results.

This design is *general and extensible*: the requester can potentially access anything the owner's virtual assistant can, which is defined by the open-world repository of virtual assistant skills. Sharing does not require the requester to get an account with the service provider; nor is the sharing limited by what is supported by the service itself. The design is *secure and supports privacy*. 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.

Requesters can send requests via any communication channel the virtual assistant listens to, and results are sent back in the same channel. Examples of channels include email, text messages, or voice if the virtual assistant is voice-activated and can differentiate speakers based on their voice.

1.2 Access Control in Natural Language

The requirements of an access control language for virtual assistant tasks are very different from previous access control languages.

Open-world commands and predicates. A virtual assistant is typically capable of a large number of skills stored in a repository [1, 16, 21]. Richer virtual assistants, such as Almond [12], let users specify, in natural language, compound tasks. Almond accepts any commands that can be mapped to a program in the *ThingTalk* programming language, which has a single control construct that can combine multiple functions together from the open *Thingpedia* repository of skills, along with predicates to filter the execution. Even though Thingpedia only has 210 functions currently, Almond can already execute hundreds of thousands of possible commands; this number

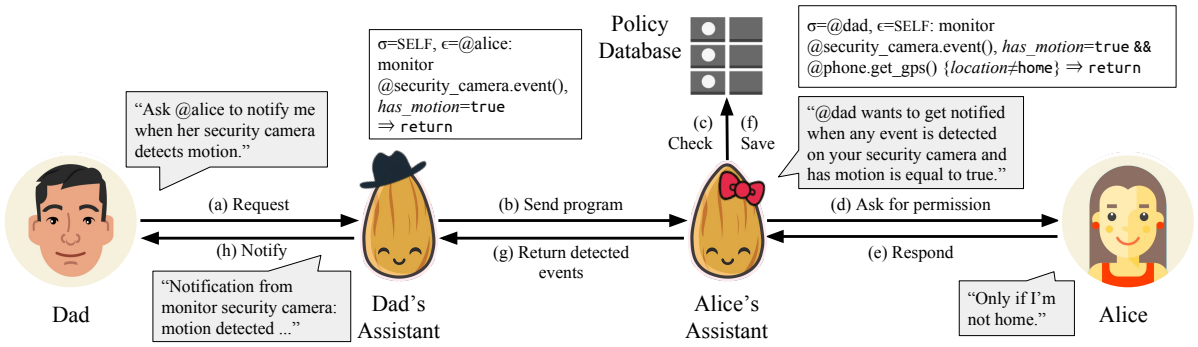


Fig. 1. Dad's request for Alice's security camera in natural language is translated into a ThingTalk program, which is executed by Alice's virtual assistant, according to Alice's access control, specified in natural language and translated into TACL.

scales quickly with the growth of Thingpedia entries. We require our access control language to support the full generality and extensibility of virtual assistant commands.

Synthesizable from natural language. The users are non-programmers rather than experienced 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 from natural language. The owner can specify who can execute what ThingTalk programs and impose constraints on input parameters and results, as well as external factors, such as the location or weather, that can be computed from any API in Thingpedia. TACL is expressive enough to support all the access control examples shown above. While TACL policies can be specified through natural language, they are formally represented as Satisfiability Modulo Theories (SMT) formulas. TACL conformance reduces to solving SMT, for which a provably correct algorithm exists.

While the semantic parser translating from natural language into TACL is outside the scope of this paper, we note that our prototype accepts natural-language commands with a parser we built with the same state-of-the-art methodology used by Almond. Our semantic parser is currently not accurate enough to be used in practice, as discussed in Section 6, but is expected to improve with the availability of more training data. We have developed a graphical user interface to complement the natural-language user interface in the meantime.

1.3 Communicating Virtual Assistants

To further simplify sharing, we enable a user to access others' data and resources in a similar manner as their own, through their own virtual assistant. Users only need to add the owner's name in their command. The requester's assistant can cooperate with the owner's assistant to accomplish the task, using the Remote ThingTalk Protocol we have developed. The protocol returns the result in rich structured form, allowing them to be used programmatically by the requester's assistant. In addition, users can share while using different virtual assistants, provided they run the same Remote ThingTalk Protocol. For example, the requester can be using the Brassau graphical virtual assistant [14], while the owner uses the text-based Almond assistant, both of which run the Remote ThingTalk Protocol.

We give a high-level overview of how users share resources in our model with an example in Fig. 1. The example shows a dad accessing his daughter's security camera through Almond. (a) Dad first makes a request to his virtual assistant in natural language, (b) Dad's virtual assistant translates the request into a ThingTalk program and sends it to Alice's assistant. (c) Alice's assistant checks if the program conforms to Alice's previously

specified TACL policies, possibly with the addition of run-time checks. If so, Alice is notified that the program will be run. If not, (d) Alice is consulted, and can provide additional constraints (e). If permission is given, the assistant (g) executes the program and returns the detected events to Dad's assistant; (h) Dad's assistant then notifies Dad of the result. The owner's assistant performs the task on behalf of the requester and only returns precisely the allowed result. We will discuss the technical details of this example in the rest of the paper.

1.4 Contributions

This paper proposes a novel approach to let consumers share existing web accounts and devices easily, securely and with unprecedentedly fine granularity of control. The contributions of this paper include:

- A design to provide secure, flexible sharing with privacy by having a virtual assistant execute requested commands, subject to owner's access control expressed in natural language. The design is secure because the owner is informed precisely of what is executed, by translating the commands back into natural language.
- TACL (Thing Access Control Language): A flexible, synthesizable access control language for virtual assistants that supports an open world of constraints. We find that 90% of the enforceable access control use cases suggested by 60 users are within the scope of TACL.
- A theoretically sound algorithm, based on Satisfiability Modulo Theories (SMT), to enforce TACL policies statically and dynamically. Experimental evaluation shows that the algorithm is practical.
- The Remote ThingTalk Protocol, which enables users to access their own and others' data in a similar manner, through their own virtual assistants. It also supports sharing without disclosure to a third party if the virtual assistants are run on users' own devices.
- A fully functional prototype of communicating virtual assistants, implemented as part of the open-source Almond assistant [12]. We will refer to this extended version simply as Almond, unless explicitly noted, in the rest of the paper. Almond has been released as an Android app and a web service[1].
- We show that people are interested in fine-grain access control. In a survey involving 200 people across a spectrum of 20 different sharing use cases, we find that, on average, adding access control to a use case makes sharing more comfortable for 28% of the people.
- We evaluate our prototype with a user study involving 20 users, and find that 18 of them like the concept, and 14 like the prototype.

1.5 Paper Organization

The rest of the paper is organized as follows. Section 2 introduces ThingTalk and TACL. Section 3 describes the user experience of data sharing with Almond assistants. Section 4 presents the algorithm to verify that ThingTalk programs conform to a set of TACL policies. Section 5 introduces the Remote ThingTalk Protocol, through which virtual assistants communicate. We present our evaluation in Section 6, and discuss the limitations of our approach and prototype in Section 7. Finally, we present related work and conclude.

2 THE TACL ACCESS CONTROL LANGUAGE

TACL is based on ThingTalk; here we first give an overview of ThingTalk, then TACL.

2.1 ThingTalk Overview

ThingTalk was introduced in the Almond virtual assistant [12] as a formal language that lets a user connects multiple web services and IoT devices in a single command. It is designed to be synthesizable from natural language. A ThingTalk program has the following syntax:

[1] https://almond.stanford.edu

Retrieval function: @security_camera.event	
Parameters	`out` *picture_url* : URL `out` *has_person* : Boolean `out` *has_motion* : Boolean
Example utterances	"get a snapshot of my security camera" "show me my security camera" "when there is a new event detected on my security camera" "when my security camera detects a person" "when my security camera detects motion"
Confirmation	"the current event detected on your security camera"
Returns a list	no
Can be monitored	yes

Fig. 2. The security camera entry in Thingpedia.

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

The program consists of one, two, or three of the clauses, separated by ⇒: a WHEN clause, a GET clause, and a DO clause. The WHEN clause specifies when the command will be triggered. In the syntax, *when* can refer to "`now`", a calendar or interval timer, or the monitoring of a *retrieval* function f with the syntax "`monitor`$f(\ldots)$"; in the latter case, the command is executed automatically whenever the result of $f(\ldots)$ changes. *get* in the GET clause calls a retrieval function, while the DO clause calls the *action* function indicated by *do*. If unspecified, the DO clause defaults to "`notify`", meaning that the results will be presented to the user.

For example, the following program continually monitors the user's Instagram profile for new pictures that have hashtag #cat, and copies them on the user's Twitter account:

```
monitor @instagram.get_pictures(), contains(hashtags, #cat)
⇒ @twitter.post_picture(url = instagram.url, caption = "cat")
```

Input parameters can be passed as keywords to each function, and output parameters can be referred to by name in later functions. Both the WHEN and the GET clause can include predicates, denoted by p_{WHEN} and p_{GET} respectively, which operate on the output parameters of the current and the previous functions. Information flows from the WHEN clause to the GET clause and then finally, to the DO clause. The DO function is executed only if p_{WHEN} and p_{GET} are satisfied. If the same retrieval function is invoked by both the WHEN and the GET clause with the same arguments, the function is evaluated only once, with each trigger of the WHEN clause.

The set of functions that can be used in ThingTalk is defined in *Thingpedia* [6]. Each function entry includes the API, with its list of input and output parameters and type information, its implementation, example utterances for natural language training, and a canonical confirmation sentence to ensure that the input command was parsed correctly. An example Thingpedia entry for a security camera is shown in Fig. 2. The represented function "@security_camera.event" has no input parameter and 3 output parameters.

2.2 Second-party ThingTalk Commands

We extend ThingTalk so commands can be executed on another user's virtual assistant. Our extension specifies a *source*, σ, who issues the command, and an *executor*, ϵ, whose assistant is to run the command:

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

By default, both the executor and the source are SELF, i.e. the person defining the program is also the owner of the assistant running it. Otherwise, we say the program is a *second-party ThingTalk program*, an example of which is shown in Fig. 1.

Upon receiving a second-party ThingTalk program, the executor virtual assistant validates it by checking its syntax and types, verifies that it conforms to the policies defined by its owner, and then executes it. The source of a program also retains the ability to stop the execution of the program at a later time.

We add one special DO function, `return`, which presents the result to the source. How this result is presented depends on the specific communication channel between the source and the executor; for example, the results could be presented in the form of an email to the source user. Section 5.2 discusses the implementation of `return` if the request is performed via a source virtual assistant.

2.3 TACL: ThingTalk Access Control Language

TACL, as a control policy language for ThingTalk, must be at least as expressive as ThingTalk. At the minimum, we should be able to specify that an individual is allowed to execute a certain ThingTalk command on our virtual assistants. Thus, we define TACL syntactically as a generalization of ThingTalk,

$$\hat{p}_\sigma, \epsilon = \text{SELF} : when, \hat{p}_{\text{WHEN}} \Rightarrow get, \hat{p}_{\text{GET}} \Rightarrow do, \hat{p}_{\text{DO}}$$

The formal grammar for ThingTalk and TACL is included in Appendix B. We have formalized the meaning of both ThingTalk and TACL with denotational semantics, and proven that our conformance algorithm presented in this paper is correct. Due to space limitation, we will only provide an informal language description and an intuitive explanation of the algorithm in this paper.

2.3.1 Source Constraints. Whereas in ThingTalk, the source of the request is a single identity, here we allow a predicate on the source, $\hat{p}_\sigma$. The user can specify a specific identity as before, a group membership predicate, or a logical combination of both. The constant predicate `true` can be used to allow anybody to request the action.

2.3.2 Function Constraints. The owner may grant a requester a family of functions. Besides the syntax allowed in ThingTalk, any function in the command can be replaced by a wildcard over any function in Thingpedia, denoted "_", or a wildcard over any function of a specific device, denoted "@*dn*._", where *dn* is the device name.

2.3.3 Input and Output Constraints. ThingTalk allows predicates on the outputs of WHEN and GET functions as a means to filter out results of no interest. Note DO functions perform a side effect and do not return a result that can be used in subsequent computation. In TACL, the predicates $\hat{p}_{\text{WHEN}}$, $\hat{p}_{\text{GET}}$, $\hat{p}_{\text{DO}}$ are a logical expression of constraints on the *input parameters* to restrict what the requesters can supply as inputs; $\hat{p}_{\text{WHEN}}$ and $\hat{p}_{\text{GET}}$ can also include constraints on any of the *output parameters* in functions executed to limit the results the owner wishes to share. The logical predicates supported for each parameter are type-based: for example, comparisons for numbers and strings, and containment for strings and arrays. Thus, Thingpedia function signatures automatically define the predicates allowed in access control.

2.3.4 Information Flow Constraints. By forcing the input of a function to be equal to the output of a previous function, we can also impose information flow constraints. For example, the flow constraint in "*allow @alice or @bob to play anything coming from Netflix on the TV, as long as it is 'G' rated*", can be expressed in TACL as:

$$(\sigma = \text{@alice} \;||\; \sigma = \text{@bob}) : \text{now} \Rightarrow \text{@netflix.search}, rating = \text{"G"} \Rightarrow \text{@tv.play\_url}, url = netflix.url$$

Another example of a flow constraint is "*allow students to post any ACM articles with tag 'access control' to our lab's Facebook account*":

$$\texttt{group}(\sigma, \text{@students}) : \text{now} \Rightarrow \text{@acm.search}, \texttt{contains}(tags, \text{"access control"}) \Rightarrow \text{@facebook.post}, url = acm.url$$

2.3.5 External Constraints. In addition, users can also restrict access based on external factors that can be computed using any of the retrieval functions in Thingpedia. For example, Alice can instruct 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. Here, the retrieval function is used as a condition for access; we refer to it as a GET predicate.

Any number of GET predicates can appear in each of the $\hat{p}_\sigma$, $\hat{p}_{\text{WHEN}}$, $\hat{p}_{\text{GET}}$, $\hat{p}_{\text{DO}}$ predicates. Syntactically, a GET predicate has the form

$$@dn.fn([pn = v]^*)\{p\}$$

which invokes function *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:

$$\sigma = \text{@dad}, \epsilon = \text{SELF} : \texttt{monitor}\ \ \text{@security\_camera.event}, \text{@phone.get\_gps}()\{\textit{location} \neq \texttt{home}\} \Rightarrow \texttt{notify}$$

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

2.4 Relationships between ThingTalk and TACL

We observe that a TACL policy that contains no wildcards and where all the required input parameters have constant values, has a well-defined execution semantics. We refer to such TACL policies as *TACL programs*. We observe that ThingTalk programs are a subset of TACL programs, as the former do not have GET predicates. The fact that *programs are just very specific policies* brings important benefits:

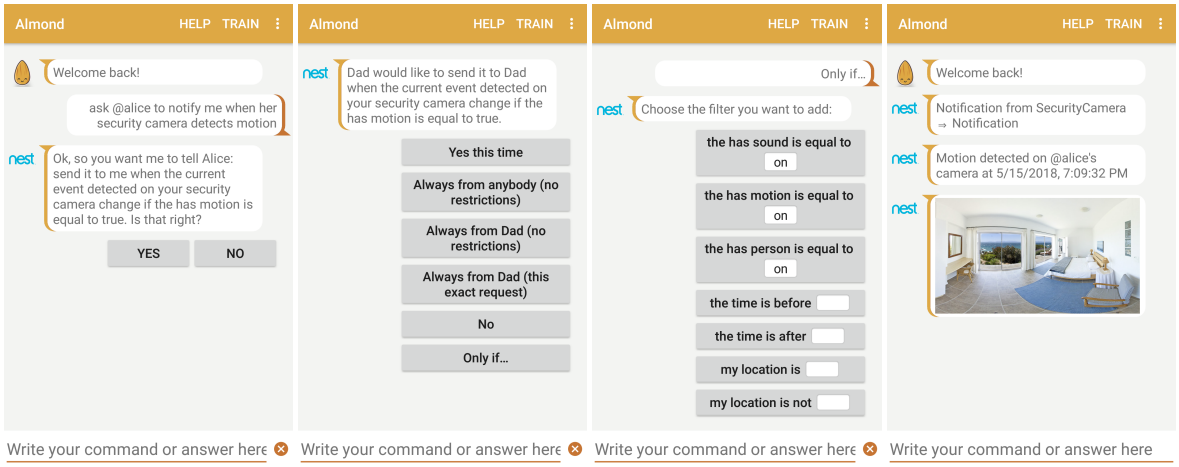
(1) *Users can understand and specify policies easily.* Users can specify the policies and issue commands in the same way, using natural language, and a similar algorithm can parse programs and policies. An incoming request can be converted into a policy trivially, allowing subsequent similar requests to execute without repeated approvals. Furthermore, users can incrementally define policies by adding constraints to a request.
(2) *Conformance can be understood intuitively.* A program *conforms* to a policy if its execution is a subset allowed by the policy. The source of the program must satisfy the source predicate in the policy; the functions in the program must either match those in the policy or be allowed by a wildcard; all the predicates in the policy must be implied by those in the program.
(3) *Conforming programs can be synthesized, possibly with run-time enforcement.* If the execution of ThingTalk a program is a superset of what the TACL policy allows, we can synthesize a conforming TACL program by adding the restrictions from the policy to the program. The restrictions can be enforced statically, or dynamically. Dynamic restrictions include all GET predicates and predicates on output results not implied already in the original program. Fig. 1(f) shows the synthesis of Dad's interest in motion and Alice's constraint, expressed as a TACL program.

3 USER EXPERIENCE

Specifying access control is hard and error prone even for professionals: how do we expect an average user to do so correctly? Furthermore, natural language itself is imprecise, and translation from natural language to code is far from perfect. We describe below how we attempt to address these issues.

3.1 On-Demand Approval and Access Control

In Almond, if a request does not conform to existing policies, the owner has a chance to approve the request on the fly. If they have reservations about a specific request, they can place constraints on it; for example, Alice wants to limit camera access only to when she is not home. Reacting to a specific request is much easier than specifying all the access control policies a priori. The users have the option to save approved programs as policies for the future, and they are also allowed to generalize some of the parameters. In this way, users can grow their database of policies gradually.



(a) Dad sends a request (b) Alice receives a request (c) Alice adds constraints (d) Dad receives a notification

Fig. 3. Screenshots of the Almond Android user interface

To get approval, the virtual assistant translates the incoming ThingTalk program into an 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 secure and guaranteed to match the code. However, the confirmation is currently generated by a deterministic rule-based system, and the result is often clunky and ungrammatical. In our running example, upon finding no usable policies in the database (Fig. 1(c)), the virtual assistant translates Dad's request as: "*Dad wants to send it to Dad when the current event detected on your security camera if the has motion is equal to true changes.*" (Fig. 1(d)). While the user can probably figure out what the request means, better confirmation generation is warranted.

The user can approve or deny this request and the requester is informed of the same. In this 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)). Her input is translated into formal TACL and saved in the database for future requests (Fig. 1(f)).

3.2 Multi-Modal User Interface

Almond lets users have a choice of either using natural language or a menu-driven graphical user interface to create second-party ThingTalk programs or specify access control policies. The requesters can click the "HELP" button at the top to get a menu of supported commands grouped by device categories. Then they can choose the desired command, fill in the parameters, and execute it. On the other side, if the grantor has not already specified a policy, they are presented different options when a request arrives. They can approve, deny, or add additional constraints to the request by either choosing the option provided or typing. As in our running example, Dad can issue the command by natural language to request access to Alice's security camera (Fig. 3(a)), whereas Alice can approve the request by clicking the prompt buttons and filling the blanks (Fig. 3(b) and 3(c)). Then Dad will get the notification from Alice once an event is triggered after approval (Fig. 3(d)).

4 ENFORCEMENT OF TACL POLICIES

Access control in Almond is expressed as a set of TACL policies. An input program *conforms* to a policy set as long as each of its execution instance conforms to some policy. Note that the policy satisfied may be different for

each execution. A program is *consistent* with a policy set if it can be made to conform with additional run-time constraints. A *conforming* program is trivially *consistent*.

The goal of the policy conformance algorithm is to determine if an input program is:

(1) *Conforming*: the program is allowed to run as is.
(2) *Consistent*: add the necessary run-time constraints to make it conforming.
(3) *Inconsistent*: reject the program.

Policy conformance can be reduced to the problem of Satisfiability Modulo Theories (SMT) [9]. 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 *satisfiable*.

We translate TACL predicates into SMT formulas and map conformance to satisfiability of various formulas. Doing so leverages previous work in making SMT solvers fast. Even though SMT is NP-hard, we will show in Section 6.4 that our algorithm scales well empirically.

4.1 Transforming TACL to SMT

To apply SMT, we define a transformation $\mathcal{L}$ from the space of programs and policies to logical formulas. We use SMT in the theory of strings, real difference logic, sets, algebraic data types and uninterpreted functions. We note that, while the general theory of strings is undecidable, the subset used by $\mathcal{L}$ guarantees that at least one parameter to the string predicates is constant and is thus decidable [22].

$\mathcal{L}$ 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 $\mathcal{L}$ is shown in Fig. 4. For clarity, we use positional input parameters in the figure, even though TACL uses keyword parameters like ThingTalk. Each retrieval function f is mapped to a multi-valued uninterpreted function $F_f(\bar{x})$, that returns a tuple of output parameters. This enables SMT to reason about the fact that calling the same function twice within a single trigger returns the same results. If the function returns a list, as indicated in its Thingpedia metadata, the input parameters include a fresh skolem variable, r, to refer to a particular instance. We introduce fresh Y variables for each function, to model passing parameters from one function to the next in the program or policy. We also introduce the variables $\overline{X}_{\text{W}}, \overline{X}_{\text{G}}, \overline{X}_{\text{D}}$ to represent the input parameters to the WHEN, GET and DO function, respectively. These variables unify the inputs passed to the functions between the program and the policies in conformance testing.

For predicates that have an exact correspondence in SMT, such as strings and numbers, $\mathcal{L}$ uses the exact SMT equivalent. Arrays are mapped to sets and set membership in SMT (the only ThingTalk operation supported on arrays is membership and duplicates are ignored). A GET predicate that invokes function f is converted in a similar way as a GET clause, except its input parameters are not unified.

For example, the program (from Section 2.1):

$$\sigma = \text{@bob} : \texttt{monitor}\ \ \text{@instagram.get\_pictures}(), \texttt{contains}(\textit{hashtags}, \#\text{cat})$$
$$\Rightarrow \text{@twitter.post\_picture}(\textit{url} = \textit{instagram.url}, \textit{caption} = \text{"cat"})$$

is converted to the formula:

$$\sigma = \text{@bob} \wedge \left(Y_{1,\text{url}}, Y_{1,\text{hashtags}}\right) = F_{\text{instagram.get\_pictures}}(r_1) \wedge\ \text{mkHashtag}(\text{“cat”}) \in Y_{1,\text{hashtags}}$$
$$\wedge\, X_{\text{D,url}} = Y_{1,\text{url}} \wedge X_{\text{D,caption}} = \text{“cat”}$$

Observe that the formula uses the skolem variable r_1 to indicate one of the many results returned by @instagram.get_pictures(). mkHashtag is a datatype constructor for the Hashtag type.

$$
\begin{aligned}
\mathcal{L}[\![\hat{p}_\sigma : w \ \Rightarrow g \ \Rightarrow d]\!] &\mapsto \mathcal{L}[\![\hat{p}_\sigma]\!] \wedge \mathcal{L}[\![w]\!] \wedge \mathcal{L}[\![g]\!] \wedge \mathcal{L}[\![d]\!] \\
\mathcal{L}[\![\_]\!] &\mapsto \text{true} \\
\mathcal{L}[\![\texttt{notify}]\!] &\mapsto \text{true} \\
\mathcal{L}[\![\texttt{now}]\!] &\mapsto \text{true} \\
\mathcal{L}[\![\texttt{monitor } g(\overline{v}), \hat{p}]\!] &\mapsto \left(Y_{pn_1}, Y_{pn_2}, \ldots\right) = F_g(r, \mathcal{L}[\![\overline{v}]\!]) \wedge \mathcal{L}[\![\overline{v}]\!] = \overline{X}_{\text{W}} \wedge \mathcal{L}[\![\hat{p}]\!] \text{ , fresh } r, Y \\
\mathcal{L}[\![g(\overline{v}), \hat{p}]\!] &\mapsto \left(Y_{pn_1}, Y_{pn_2}, \ldots\right) = F_g(r, \mathcal{L}[\![\overline{v}]\!]) \wedge \mathcal{L}[\![\overline{v}]\!] = \overline{X}_{\text{G}} \wedge \mathcal{L}[\![\hat{p}]\!] \text{ , fresh } r, Y \\
\mathcal{L}[\![d(\overline{v}), \hat{p}]\!] &\mapsto \mathcal{L}[\![\overline{v}]\!] = \overline{X}_{\text{D}} \wedge \mathcal{L}[\![\hat{p}]\!] \\
\mathcal{L}[\![f(\overline{v})\{\hat{p}\}]\!] &\mapsto \left(Y_{pn_1}, Y_{pn_2}, \ldots\right) = F_g(r, \mathcal{L}[\![\overline{v}]\!]) \wedge \mathcal{L}[\![\hat{p}]\!] \text{ , fresh } r, Y \\
\mathcal{L}[\![\hat{p}_1 \ \texttt{\&\&} \ \hat{p}_2]\!] &\mapsto \mathcal{L}[\![\hat{p}_1]\!] \wedge \mathcal{L}[\![\hat{p}_2]\!] \\
\mathcal{L}[\![\hat{p}_1 \ \texttt{||} \ \hat{p}_2]\!] &\mapsto \mathcal{L}[\![\hat{p}_1]\!] \vee \mathcal{L}[\![\hat{p}_2]\!] \\
\mathcal{L}[\![\texttt{!}(\hat{p})]\!] &\mapsto \neg \mathcal{L}[\![\hat{p}]\!] \\
\mathcal{L}[\![vn \ \ op \ \ v]\!] &\mapsto \mathcal{L}[\![vn]\!] \ \ op \ \ \mathcal{L}[\![v]\!] \\
\mathcal{L}[\![\texttt{substr}(vn, v)]\!] &\mapsto \text{StrContains}(\mathcal{L}[\![vn]\!], \mathcal{L}[\![v]\!]) \\
\mathcal{L}[\![\texttt{starts\_with}(vn, v)]\!] &\mapsto \text{StrPrefixOf}(\mathcal{L}[\![v]\!], \mathcal{L}[\![vn]\!]) \\
\mathcal{L}[\![\texttt{ends\_with}(vn, v)]\!] &\mapsto \text{StrSuffixOf}(\mathcal{L}[\![v]\!], \mathcal{L}[\![vn]\!]) \\
\mathcal{L}[\![\texttt{contains}(vn, v)]\!] &\mapsto \mathcal{L}[\![v]\!] \in \mathcal{L}[\![vn]\!] \\
\mathcal{L}[\![vn]\!] &\mapsto \begin{cases} Y_{vn} & \text{if } vn \text{ is an output parameter} \\ X_{vn} & \text{if } vn \text{ is an input parameter} \end{cases}
\end{aligned}
$$

Fig. 4. Definition of the $\mathcal{L}$ 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 and type constructors such as mkHashtag and mkLocation.

Similarly, the policy (from our running example):

$$\sigma = \text{@dad} : \texttt{monitor } \text{@security\_camera.event}, \text{@phone.get\_gps}()\{\textit{location} \neq \texttt{home}\} \Rightarrow \texttt{return}$$

is converted to the formula:

$$
\begin{aligned}
&\sigma = \text{@dad} \wedge \left(Y_{1,\text{picture\_url}}, Y_{1,\text{has\_motion}}, Y_{1,\text{has\_person}}\right) = F_{\text{security\_camera.event}}() \\
&\wedge \left(Y_{2,\text{location}}\right) = F_{\text{phone.get\_gps}}() \wedge \texttt{home} \neq Y_{2,\text{location}}
\end{aligned}
$$

Note that this time there is no skolem variable because both @security_camera.event and @phone.get_gps are marked in Thingpedia as returning only one result.

Given a program π, it holds that if $\mathcal{L}[\![\pi]\!]$ is unsatisfiable, the program will never have any visible side effect. This occurs if the predicates in π are contradictory, and in that case we say π is a *null program*. For reasons of space, we omit the proof of the correctness of $\mathcal{L}$.

4.2 Checking Conformance

We say that a program is *compatible* with the policy, if the source and functions match those in the policy. Clearly, a program can only conform to compatible policies. A program π conforms to a set of compatible policies Π if

$$\mathcal{L}\{\!\{\pi\}\!\} \models \bigvee_{\pi_i \in \Pi} \mathcal{L}\{\!\{\pi_i\}\!\}$$

That is, there does not exist an instance of π that is not covered by the union of the policies. This formula can also be expressed as a satisfiability query:

$$\neg \text{SAT}\left(\mathcal{L}\{\!\{\pi\}\!\} \wedge \neg \bigvee_{\pi_i \in \Pi} \mathcal{L}\{\!\{\pi_i\}\!\}\right)$$

4.3 Synthesizing a Conforming Program

If an input program does not conform as is, it may be possible to synthesize a more restricted version that conforms to the owner's policies.

Given a program π with components *when*, *get*, *do*, p_{WHEN}, p_{GET}, and compatible policies $\pi_i \in \Pi$ with predicates $\hat{p}_{i,\text{WHEN}}, \hat{p}_{i,\text{GET}}, \hat{p}_{i,\text{DO}}$, we create program π' as follows:

$$\begin{aligned} & \textit{when}, p_{\text{WHEN}} \;\&\&\; (\hat{p}_{1,\text{WHEN}} \;||\; \hat{p}_{2,\text{WHEN}} \;||\; \ldots) \\ \Rightarrow\; & \textit{get}, p_{\text{GET}} \;\&\&\; ((\hat{p}_{1,\text{WHEN}} \;\&\&\; \hat{p}_{1,\text{GET}} \;\&\&\; \hat{p}_{1,\text{DO}}) \;||\; (\hat{p}_{2,\text{WHEN}} \;\&\&\; \hat{p}_{2,\text{GET}} \;\&\&\; \hat{p}_{2,\text{DO}}) \;||\; \ldots) \\ \Rightarrow\; & \textit{do} \end{aligned}$$

This program checks that at least one $\hat{p}_{i,\text{WHEN}}$ of a policy is satisfied, in addition to the p_{WHEN} of the program. Then it checks that the $\hat{p}_{i,\text{GET}}$ and $\hat{p}_{i,\text{DO}}$ predicates of the same policy are satisfied. This ensures that, for *each* program 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 π' imposes the least constraints to make π conforming to the user policy Π. However, the program π' may not produce any results if the predicates contradict each other. We can test if π' is a null program by asking if $\mathcal{L}\{\!\{\pi'\}\!\}$ is satisfiable. The details of the conformance algorithm are included in Appendix A.

5 REMOTE THINGTALK PROTOCOL

Our overall design for sharing is based on submitting requests to the executor virtual assistants, which validates them, verifies their conformance and then executes them. To support requests coming through another virtual assistant, we introduce the Remote ThingTalk Protocol, by which the source assistant sends the requested program to the executor assistant, and receives the results back if applicable. The Remote ThingTalk Protocol allows users to own and share their data without involving a third party; furthermore, users can access their own and others' data with the same virtual assistant interface.

5.1 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. Messages to the same account are delivered in order.

5.2 Remote Execution Protocol

When the user makes a request for remote resources, their assistant, acting as the source, generates a second-party ThingTalk program and sends an Install message to the executor, containing: (1) *program*, the source code of the program to execute; (2) *identity*, the real-life identity of the source; (3) *progid*, the unique identifier of the program.

The executor virtual assistant first verifies that the identity claimed in the message corresponds to the sender. Then, if the program is consistent with the policy defined by the owner, the assistant runs the compliant program returned by the conformance algorithm (Section 4.3), otherwise the user is asked to approve (Section 3.1). If the program is denied, the assistant replies with an Abort(*progid*, *reason*). When the program on the executor terminates, the executor signals its successful end via an End(*progid*) message. 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.

The original ThingTalk request may require results to be returned to the source, rather than just an action to be performed. The communication of results is achieved by having the source and the executor virtual assistant coordinate through a pair of ThingTalk programs. The source virtual assistant, u_1, rewrites the original ThingTalk program with a "`return`" DO function as a pair of low-level programs:

$$\sigma = u_1, \epsilon = u_2 : \mathit{when} \Rightarrow \mathit{get} \Rightarrow \texttt{send}(\mathit{to} = u_1, \mathit{flow} = f, \text{“}\mathit{payload}\text{”})$$
$$\sigma = u_1, \epsilon = u_1 : \texttt{monitor receive}(\mathit{from} = u_2, \mathit{flow} = f) \Rightarrow \texttt{notify}$$

The first program is sent to the executor, u_2, for approval and execution; the second is executed by the source upon having the first program approved. Both programs are given the same unique program identifier, *progid*, so that they can be stopped at the same time. The *progid* is sent to the executor in the Install message.

`send` and `receive` are communicating functions between two virtual assistants. The two functions are connected by a *flow*, a unique identifier that pairs them; "*payload*" is a placeholder for the actual output parameters returned by the program. The definitions of these functions are entered into Thingpedia, no different from any other APIs, and the general ThingTalk implementation can execute these operations with no modification. When the `send` function is invoked, the executor sends a Data(*flow*, *payload*) message. This message is routed to the corresponding `receive` function based on the flow identifier and triggers the `notify` action, showing the result to the user. The source assistant stops the `receive` program upon receiving an End message, after processing all previous Data messages; it also stops the `receive` program upon receiving an Abort message.

5.3 Choice of Messaging Protocol

Our prototype uses the Matrix messaging protocol [23]. Matrix supports the requirements of the Remote ThingTalk Protocol: 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.

5.4 Security Considerations

Access to private user information makes Almond 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 Almond, 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 memory-safe dynamic language, which reduces the surface area for attacks. Communication with Thingpedia and the Almond 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. Almond 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 [13].

Remote Code Execution. Executing code received from an external source is dangerous. Almond can execute only programs written 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.

Spamming and Denial of Service. Almond relies on the messaging service to prevent spamming. We expect the messaging service to rate-limit the messages each user can send and receive, and to offer the ability to block all messages from certain users. Additionally, because the conformance algorithm is NP-hard, a potential denial of service can occur if an adversary crafts a program that requires exponential time to verify against the policy. We will show in Section 6.4 that all legitimate requests can be checked in a short amount of time, so this attack can be mitigated with strict timeouts on the conformance algorithm.

6 EXPERIMENTATION

We have developed a full prototype of our design, as part of the Almond open source project. We extended the Almond natural language parser to support second-party ThingTalk programs and TACL policies. We added to the existing training set 3,577 second-party ThingTalk commands and 4,285 TACL policies, collected using the *paraphrasing* technique [32]. The resulting parser achieves an accuracy of 61% for second-party commands and 74% for policies, a result similar to those previously reported for Almond. Details of this parser are outside the scope of this paper, but we note that it differs from the one discussed by Campagna et al. [12].

Here we present studies to address the following questions: Do people have a need for fine-grain access control? Do people like the concept of sharing with virtual assistants? Do they like the Almond prototype? Is TACL expressive enough to support typical uses? And finally, is our policy conformance algorithm efficient enough to handle many policies?

6.1 The Need for Access Control

The goal of this study is to evaluate whether adding access controls, such as those supported by Almond, would make people more comfortable to share data. We conduct a survey with 20 use cases, described in Fig. 5, chosen to cover most of what Almond supports, or could support with new devices in Thingpedia. With each use case, we present a *baseline* policy with only role-based access control, i.e., giving out full permission to users playing

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

Fig. 5. 20 sharing use cases in the survey

certain roles. Immediately following that, we show them two examples of attribute-based access control. 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 constraint that "*only between 7 and 9 pm*" and then with another constraint "*only free G or PG rated movies*.". We ask the user to rate the comfort level of each use case; each attribute-based constrained policy was rated individually. We use a five-point Likert scale, labeled with: "very uncomfortable, I would not do that", "uncomfortable", "neutral", "comfortable", "very comfortable, no problem at all". In reality,

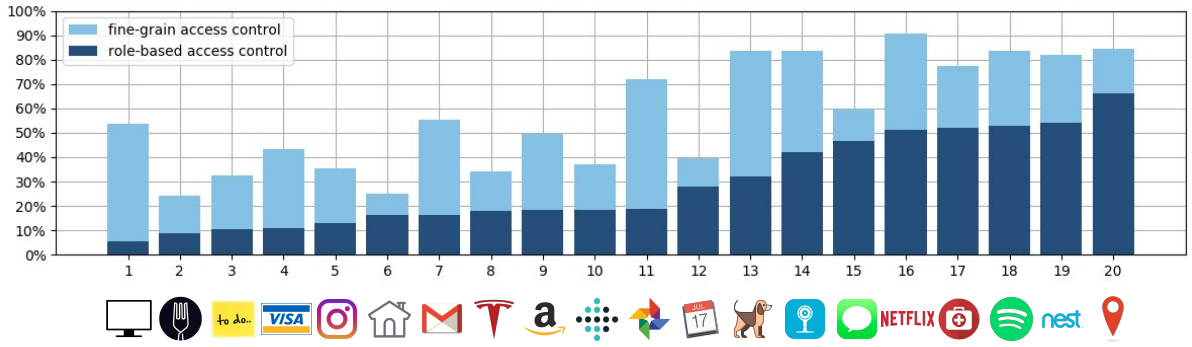


Fig. 6. Percentage of people who find the use cases in Fig. 5 "comfortable" or "very comfortable."

users can specify any constraints they wish; we approximate this capability by finding the maximum of the ratings a user assigned to the two constraints for each use case.

The survey is taken by 200 Amazon Mechanical Turk workers, all residing in the United States. Each is compensated $5 for taking part in the survey, which on average requires 30 minutes. The study is approved by our university's IRB. The workers are evenly split between men and women, and they span all age groups and education levels. 29% of the workers are active users of virtual assistants, and 67% are familiar with them.

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 6% to 66% 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 17 and 20. When allowed finer-grain 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, emails, Fitbit information, photos, a dog's location, security cameras, SMS, blood pressure, Spotify playlists, and current location. The grantors are less motivated to share private 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. In the case of one's current location, the grantor is willing to share only with a highly trusted person.
- *Liability and need-to-act for actions.* 9 of our scenarios involve granting rights to accounts to perform actions, from UberEats, 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 (UberEats, 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

#	Requester: Bob	Grantor: Alice
1	Monitor @alice's security camera	Only when I'm not here
2	Get @alice's recent Instagram pictures	Only for those with caption containing "trip"
3	Tweet "hello - from alice" on @alice's Twitter	Only if the tweet body contains "from bob"

Fig. 7. Three scenarios used in the end-user evaluation study.

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.

These results suggest that fine-grain access control can greatly improve the general public's comfort level in sharing. On average, an additional 28% of the population becomes comfortable with the sharing, with an increase of 40% of the population in 4 cases. While originally only 5 scenarios are found comfortable by 50% or more of the users, the number increases to 11 with constraints. Of the remaining 15 scenarios, the percentage of people finding it comfortable doubles in 10 cases.

6.2 End-User Evaluation of Almond

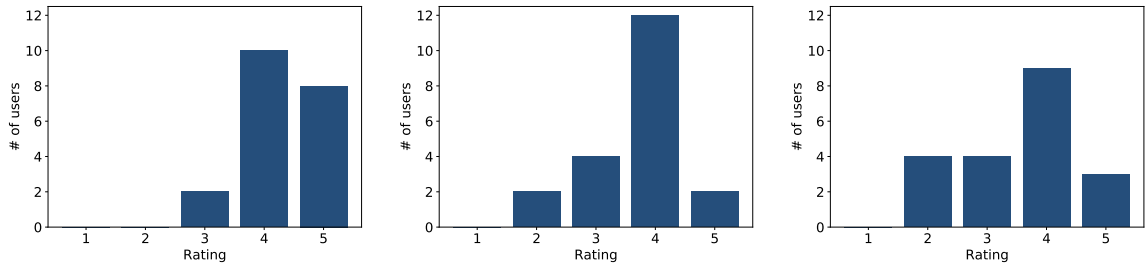
Next, we perform an in-person study to evaluate the prototype and gather user feedback. This design of this study is informed by an earlier pilot that identified issues which we fixed. For this study, we recruit 20 users: 2 staff members and 18 students, of which 2 are from Computer Science and the rest from other departments. 6 participants are women and 14 are men. Users receive $15 in Amazon Gift Cards as a token of appreciation.

Users are shown two tablets, both running the Almond app, one acting as a requester and one as a grantor. We show them an example interaction, similar to the one shown in Fig. 3. Users get to see the full experience, including the result of their actions.

6.2.1 Natural Language vs GUI. To start, we ask the users to perform the 3 scenarios in Fig. 7, both in natural language (by typing the command) and with the menu-driven GUI. For the natural-language task, the users are provided with an example command that they can copy, but they are informed they can paraphrase the command if they would like. The users perform the tasks successfully in natural language, without intervention, in 58 out of the 60 cases. Starting from the menu, however, they complete the request only in 38 cases without help. 5 users need help in the first scenario, as they are not familiar with the interface yet. For 7 users, intervention is needed in the third scenario, because they stop scrolling the menu before the correct command and think they have made a mistake. This confirms the concern that it is hard to find one of many possible commands in a menu-driven GUI.

We then present the users with a cheat sheet that shows all the 42 services/devices and 361 commands. We ask them to choose at least 2 scenarios they find generally useful. They perform the request for those scenarios using their preferred interface, then approve the request with their choice of policy. The users try 48 commands, 7 of which fail due to missing or erroneous functions in Thingpedia; this issue is outside the scope of this paper. 22 of the remaining 41 commands are completed using the GUI, where they just pick some combination presented to them. The rest are attempted in natural language; 74% of these cases run to completion. The failure rate is consistent with the accuracy reported for the natural language parser. Common errors include missing quotes around textual parameters and missing @-signs in names. This is a preexisting limitation of the Almond natural language parser, and we expect it will be lifted in the future; in the meantime, users need to learn to avoid it.

After users try out their individual scenarios, we ask them which interface they prefer. They are evenly split between natural language and the GUI. We also ask the users if they would prefer a voice UI, if available: 14 users out of 20 say they would. The voice-to-text capability can be independently addressed; existing voice-to-text technology must be augmented with identification of textual parameters, which need to be quoted.



(a) Do you like the concept? (b) Do you like the app? (c) Would you use it?

Fig. 8. The distribution of the answers to our survey questions regarding the Almond prototype.

6.2.2 Feedback on Usage. Users choose a wide variety of resources to share, from social media sharing, IoT devices, calendar and cloud drive access. We are a bit surprised that a couple of the users find it useful to permit others to post on their social media account. Most of the time, users want to approve the request for just one time. One wants to impose an additional constraint to the request and still have it run just once; they are worried that they would forget that the permission is enabled. On-demand approval of specific requests is an important use case, and also the primary goal of Almond. It is also less susceptible to user error.

Users also show interest in imposing a wide variety of access control constraints, another important design goal of Almond. One user wants to grant everybody access to their LinkedIn profile. There are two uses of limiting access to an individual, three uses of output constraints, and one use of an external GET predicate. One user suggests letting people ask for permission to all the functions of a particular service, which is not yet supported. One user is concerned about spam requests, which are addressed by Almond's underlying messaging system.

6.2.3 Rating the Concepts and the Prototype. We ask the users to rate the concept and the Almond prototype on a 5-point Likert scale, from strongly dislike (1) to strongly like (5). The results are summarized in Fig. 8. Overall, most users like the concept of Almond, as embodied by the app, with 18 out of 20 rating it at least 4 out of 5. One user summarizes the purpose of the app succinctly as "*sharing without passwords*". A user comments on how it is useful for "*email, because calendar can be shared already but email is normally private*".

Users also comment positively on not having to log into their own account to get information from others' accounts. One user wants to "*ask @nasa for the latest asteroid data*". We first observe that users need not ask for permission to see public data, and that the users' virtual assistant can execute the command locally. This utterance makes us realize how natural the "ask ⟨entity⟩ to ⟨do a task⟩" syntax is; it is equally applicable to public data and data belonging to an individual. After all, this is the syntax used in Alexa for all third-party services, such as "*ask Bing to search for x*". In the future, the virtual assistant should accept the same syntax for both public and private access, hiding the distinction of first-party vs second-party execution from the user for simplicity.

Although Almond is just a prototype, and not a product, 14 users, 70%, indicate they like or strongly like the current implementation. One user comments that the app is "*a good research prototype*"; a second user thinks that the concept is "*totally new and great*" but finds the app to have a "*high barrier to entry*". Finally, 12 users, 60%, say they would use the app, 4 are neutral, and 4 would not use it.

Roommate:	"*I want to read your Gmail account*".
Me:	"*OK, but you can only read those with subject 'rent receipt'*"
Mom:	"*I want to remind you to buy apples when you are in a grocery store next time. Can you share your GPS location with me?*"
Me:	"*OK, but you can only know my location when I'm near a grocery store.*"
Girlfriend:	"*I got some nice photos for our trip in my phone, give me your Facebook account so that I can upload them for you.*"
Me:	"*Sure, but you can only upload pictures, no read.*"

Fig. 9. The 3 use cases shown to MTurk workers as examples, before asking for more use cases.

6.3 Expressiveness of TACL

TACL is designed to support a wide range of access controls on an open and extensible world of commands. Our next experiment crowdsources use cases and relevant access control constraints beyond the limitation of the current prototype, and analyzes them to evaluate the expressiveness of TACL.

6.3.1 Diversity of Access Controls. To solicit use cases, we first show workers three example use cases (Fig. 9) where a user puts constraints on a requested access. Without revealing anything about our system at all, we then ask them to suggest new use cases similar to the ones presented (not necessarily use cases they would use themselves). We hire 60 Amazon Mechanical Turk workers, from all around the world, and they are compensated $2 for 4 use cases; we collect altogether 220 valid use cases. 20 responses are invalid: 4 are blank, and 16 do not involve an interaction between two users. Each task takes 20 minutes on average. No IRB review is necessary since no personal questions are asked.

The suggested use cases involve 85 unique devices and services, covering a large variety of situations. They are classified below, along with the number of mentions 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): streaming services, 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.

The constraints used in the 220 cases fall mainly into five categories, whose representative examples are shown in Fig. 10:

- *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.
- *External constraint* (25): The execution of the command depends on external conditions, such as time, location, and weather.

Constraint	Examples		Resource Type
Function	Brother:	"*Let me access your calendar to add a reminder so that you won't forget our parents anniversary.*"	Personal data
	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.*"	Personal data
	Husband:	"*Yes, but only to draft and send a message.*	
Input	Daughter:	"*Mom, can I send a text to grandma about this weekend?*"	IoT device
	Mom:	"*Sure, but only text grandma and no one else.*"	
	Mom:	"*You need to follow this guy on twitter, give me your twitter account.*"	Social media
	Me:	"*OK, add him but don't follow any other twitter user.*"	
Output	Friend:	"*Can I access non-shared files on your Google Drive so I can read some books?*"	Personal data
	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?*"	Personal data
	Me:	"*Yes, but not the sections tracking my psychological states and moods.*"	
External	Tenant:	"*I believe there may be a hurricane coming soon. Can I put down the window shutters?*"	IoT device
	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.*"	IoT device
	Me:	"*Sure, but only if I'm there to help you.*"	
Aggregate	Friend:	"*I forgot my gmail account can I have your gmail password to send a mail?*"	Personal data
	Me:	"*Yes, but only send one email.*"	
	Son:	"*Mom I need to use your uber account until I receive my new phone.*"	Service
	Mom:	"*Ok but use 4 rides max.*"	

Fig. 10. Representative use cases and access controls created by Mechanical Turk workers.

- *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.

Of the 220 use cases, 118 involve a close family relationship (parental, spousal, sibling) between the requester and the grantor, which implies a high degree of trust already present in the request. 61 cases are between friends, and 39 cases involve some kind of work or business relationship (such as classmates, business partners, secretaries, and tenants. Finally, only 2 cases out of 220 involve strangers, such as AirBnb guests, and no cases at all are "open to all" access. Thus, access control is mostly applied to sharing with trusted relations, and it is widely applicable to diverse assets in real life.

6.3.2 Applicability of TACL. We deliberately provided the workers, drawn from the general population, 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, and did not expect 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. Nevertheless, the majority of the 220 use cases fall within the scope of TACL, as described 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 adding the publicly available APIs to Thingpedia.
- 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 and service providers 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. 3 more 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 their promise.

This study shows that TACL covers 90% of the 206 enforceable cases suggested by our workers, provided that the APIs are available. Since the workers are unaware of the expressiveness of TACL, the results suggest that TACL has a good coverage of the kind of access control that laymen want.

6.4 Policy Conformance Evaluation

Because SMT is NP-hard, it can require exponential time to solve 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 [7] version 1.5, with the SMT-LIB language front end [8]. We enable all supported quantifier-free theories, and enable the "experimental string" option. We also find that enabling the "string-guess-models" option, which controls some heuristics in the theory of strings, improves the performance noticeably.

6.4.1 Test Suite. For this experiment, we generate 4,000 programs from 48 device classes and 192 functions in Thingpedia. To avoid oversampling devices that have more APIs, we sample in order the kinds of clauses (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 is 3, while the maximum is 65. These unrealistically large numbers of predicates are useful for evaluating the worst-case behavior of the algorithm.

As discussed in Section 4, 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.

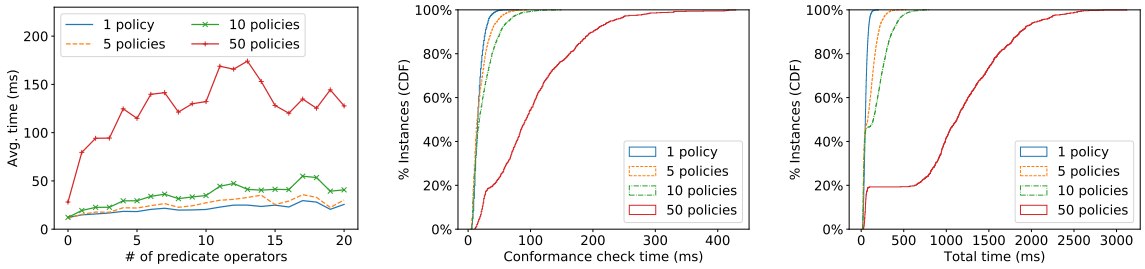
We run 4 experiments, where we try to generate 1, 5, 10, 50 compatible policies, respectively. We expect a program to have no more than 5 compatible policies in practice; we experiment with 50 compatible policies to understand scalability. The number of possible compatible policies is limited by the 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.

6.4.2 Conformance of Test Suite. We run the test suite through our policy conformance algorithm, and obtain the results shown in Fig. 11. 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

# policy	# program	Null	Inconsistent	Consistent	Conforming
1	4000	9.2%	26.8%	48.1%	15.9%
5	3786	9.1%	2.1%	47.3%	41.4%
10	3282	8.9%	0.6%	48.7%	41.8%
50	1067	7.1%	0%	75.0%	17.9%

Fig. 11. The results of our tests on the SMT algorithm.



(a) Average time to check a program with the given number of predicates.

(b) CDF of the time to check conformance of a program to the policies.

(c) CDF of the total time to check and synthesize a program.

Fig. 12. Evaluation of policy conformance algorithm.

0 even with 5 randomly generated policies. 48% of the programs are consistent for sets with 10 or fewer policies, whereas 75% are consistent for sets with 50 policies.

6.4.3 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. 12(a) 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. 12(b). 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 fewer

compatible policies, and up to 3 seconds for 50 compatible policies (Fig. 12(c)). From Fig. 11, we see that the synthesis step is needed for about 50% of programs with 10 or fewer 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.

7 LIMITATIONS AND FUTURE WORK

Our evaluations suggest that people recognize the need for access control and its broad applicability over many different services and devices. However, adoption of existing access control platforms is limited in practice. Previous work has shown that authoring access control policies in a traditional user interface can be cumbersome and error-prone [24, 26, 29].

Our proposal stands a higher chance of adoption than previous solutions because of two reasons: virtual assistants are already commercially viable, and they already have access to the credentials of users. Additionally, our solution is general since we are leveraging the same repository of device APIs used by the virtual assistant.

Almond is also a departure from existing access control systems, because it does not require the owner to grant direct access to the underlying service. This is achieved by having the owner's virtual assistant perform the allowed operation on behalf of the requester. The owner also has fine granularity of control because they can approve a specific program, controlling the functions executed, the values of the parameters, as well as the information flow from one function to another.

7.1 Natural Language

The main limitation in the system arises from the natural language parser, which, while achieving state-of-the-art accuracy, is still not completely usable. At the current stage, only educated and motivated users can overcome its limitations, in particular the requirement to tag parameters with quotes and @-signs. We expect that as natural language technology matures, these requirements will go away in time.

The reverse problem of converting from the parsed ThingTalk program back to natural language, used to confirm the user's action and to request permissions interactively, is equally hard and is currently imperfect. The rule-based system employed by Almond is effective for basic policies with function and role constraints, but becomes less understandable as the complexity of the policy increases. We hypothesize that users will err on the side of rejecting the request when it is not clear. At the same time, future work will explore how to generate understandable and precise confirmation sentences.

7.2 Complexity of Access Control

Fine granularity of access control is hard even for professionals, and it can be difficult to have a precise, global view of what access has been granted, and to whom [26]. Bauer et al. report that on-demand approval is at the top 3 of desired functionality in access control [10], and Smetters et al. in particular highlight that users prefer *contextual* access control policies [29]. In Almond, by using the on-demand approval interface, the user is shown the entire detailed context and is able to make informed decisions at the time of access rather than a priori. If desired, the user can accept a requested program as a policy for the future, and add incremental constraints before approval. Based on the results of our user study described in Section 6, we expect a typical user to approve one-off requests most of the time, and keep only a small number of outstanding access control policies at any one time.

One common source of complexity in access control systems is interactions between different rules. In Almond, the user can only specify what is allowed, versus what is not allowed, in each policy. The user may provide multiple policies, and these policies are implicitly combined as a disjunction. That is, a program is allowed only if it is allowed by the union of the policies. This design reduces complex interactions between policies.

If the users wish to impose extra constraints, they need to be aware of corner cases, especially when dealing with adversaries. Our system is intended to help users share with trusted relationships, like friends, family, and colleagues, where recourses exist. For example, a father, wishing to give his son a $10 gift, asks his virtual assistant to let his son purchase anything on Amazon within $10. Instead, the son gets around the constraint by making multiple $10 purchases. The son's action may satisfy the letter of the policy, but not its intent. Such a breach of trust can be dealt with out-of-band, and would most likely result in a reprimand and reduced privileges in the future. In Almond, users are alerted of all requests, including those approved with a policy, so corrective action can be taken before the effects become severe. We could also inform users of potential unintended consequences when they enable a policy, but as in real life, iterative refinement is often needed. Nonetheless, Almond offers a better alternative than sharing one's account credentials and relying on a gentleman's agreement to limit access, which would expose access to more data and capability than necessary.

8 RELATED WORK

8.1 OAuth

OAuth [17] is an authentication and access control technology that is commonly used to allow third-party applications to access web services. In the OAuth framework, an application wishing to access the user's account on a web service will ask the user for a limited set of permissions, called *scopes*, and will receive in return an access token; given the token, the application can now communicate directly with the web service to execute operations under the requested scope.

The set of possible scopes is defined by the web service and usually corresponds to different levels of access, such as read-only or read-write. It is therefore a more coarse sharing policy than what can be implemented by TACL. Furthermore, the scopes requested by a particular application are defined by its developer: the user only has the option to either approve or deny the whole set of scopes, with no option to add restrictions.

8.2 Access Control Systems

Access control systems can be divided in two major classes: Role-based Access Control (RBAC) [27] and Attribute-based Access Control (ABAC) [34]. 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 [18–20]. 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 by constraining the source of the program. 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 [25]. 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, such as disjointness and conflict [3, 4, 31]. 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. propose an algorithm to modify access control requests to be policy-conforming, based on a manually defined optimality condition [2]. This is a different synthesis problem than the TACL one, because TACL modifies the program by restricting the execution while preserving the intent.

8.3 Policy Languages for Programs

There are several languages proposed in the literature that express policies on the behavior of programs, rather than single functions [5, 11, 33]. 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.

8.4 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. Limited parental control support is available in "Echo Dot Kids", a version of the device that can only access a subset of skills chosen by the parent. For the full version of Alexa or Google Home, users cannot restrict actions nor control their data.

Our work is built on top of the open Almond virtual assistant platform [12]. The previous version of Almond does not support multi-user interactions. We extend it in several major ways: we introduce the TACL language, present a new access control conformance algorithm, and we augment the execution engine with the Remote ThingTalk Protocol.

8.5 Peer-To-Peer Data Sharing

Many systems offer the ability to share personal data in a peer-to-peer fashion [15, 28, 30]. Contrail is a federated social network with a pub-sub model [30]. Users can add filters on the subscribed data, and evaluation occurs on the publisher nodes. Unlike in Almond, 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 is a federated architecture where brokers mediate queries to existing data sources, such as Facebook or email [28]. 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.

9 CONCLUSION

With the rise of the virtual assistant, consumers will have a smart agent that holds all their credentials and can carry out natural language commands. We see this as an opportunity to greatly improve how people 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 them. The execution of the requests is made secure by having the owner's assistant translate the command into natural language for approval.

Moreover, users can use natural language to assert fine-grain 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. The Remote ThingTalk Protocol lets users access their own and others' data through their own virtual assistant, while enabling sharing without disclosing information to a third party.

We found that the general public finds a need for access control for sharing, with all our 20 sharing scenarios seeing an increase in comfort once constraints are applied. 90% of the 20 users in our study say that they like the concept proposed, and 70% like the prototype. Our prototype has been merged in the Almond open source project and released online and on the Google Play Store. 90% of enforceable access controls of interest proposed by 60 users can be expressed in TACL. Finally, our algorithm for policy conformance is found to be efficient.

Today, users are at the mercy of the service providers on how they can share their data. With this proposal, users can share any device or information available to their virtual assistants easily, exactly, and securely according to their preference.

ACKNOWLEDGMENTS

Support for this work was provided in part by the Stanford MobiSocial Laboratory, sponsored by AVG, Google, HTC, Hitachi, ING Direct, Nokia, Samsung, Sony Ericsson, and UST Global.

REFERENCES

[1] Amazon. 2017. Amazon Alexa. https://developer.amazon.com/alexa.

[2] Konstantine Arkoudas, Ritu Chadha, and C Jason Chiang. 2011. An Application of Formal Methods to Cognitive Radios.. In *First International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS@FMCAD 2011)*.

[3] Konstantine Arkoudas, Ritu Chadha, and Jason Chiang. 2014. Sophisticated Access Control via SMT and Logical Frameworks. *ACM Trans. Inf. Syst. Secur.* 16, 4, Article 17 (April 2014), 31 pages. https://doi.org/10.1145/2595222

[4] Alessandro Armando and Silvio Ranise. 2011. Automated Symbolic Analysis of ARBAC-Policies. In *Security and Trust Management*. Springer Berlin Heidelberg, 17–34. https://doi.org/10.1007/978-3-642-22444-7_2

[5] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2465106.2465121

[6] Various Authors. 2017. Thingpedia - knowledge for your virtual assistant. https://thingpedia.stanford.edu

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.smt-lib.org.

[9] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. *Handbook of satisfiability* 185 (2009), 825–885.

[10] Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. 2008. A User Study of Policy Creation in a Flexible Access-control System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 543–552. https://doi.org/10.1145/1357054.1357143

[11] Lujo Bauer, Jay Ligatti, and David Walker. 2005. Composing Security Policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 305–314. https://doi.org/10.1145/1065010.1065047

[12] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. 2017. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. ACM Press, New York, New York, USA, 341–350. https://doi.org/10.1145/3038912.3052562

[13] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 451–466. https://doi.org/10.1109/eurosp.2017.27

[14] Michael Fischer, Giovanni Campagna, Silei Xu, and Monica S. Lam. 2018. Brassau: Automatically Generating Graphical User Interfaces for Virtual Assistants. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI 2018)*. https://doi.org/10.1145/3229434.3229481

[15] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy. 2007. Homeviews: peer-to-peer middleware for personal data sharing applications. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/1247480.1247508

[16] Google. 2018. Google Assistant - Just Say "Hey Google" and Make Google Do It. https://assistant.google.com/.

[17] Dick Hardt. 2012. *The OAuth 2.0 authorization framework*. Technical Report. https://tools.ietf.org/html/rfc6749

[18] Jingwei Huang, David M. Nicol, Rakesh Bobba, and Jun Ho Huh. 2012. A Framework Integrating Attribute-based Policies into Role-based Access Control. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT '12)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2295136.2295170

[19] Sun Kaiwen and Yin Lihua. 2014. Attribute-Role-Based Hybrid Access Control in the Internet of Things. In *Web Technologies and Applications*. Springer International Publishing, 333–343. https://doi.org/10.1007/978-3-319-11119-3_31

[20] D. Richard Kuhn, Edward J. Coyne, and Timothy R. Weil. 2010. Adding Attributes to Role-Based Access Control. *Computer* 43, 6 (jun 2010), 79–81. https://doi.org/10.1109/mc.2010.155

[21] Anjishnu Kumar, Arpit Gupta, Julian Chan, Sam Tucker, Björn Hoffmeister, and Markus Dreyer. 2017. Just ASK: Building an Architecture for Extensible Self-Service Spoken Language Understanding. *CoRR* abs/1711.00549 (2017). arXiv:1711.00549 http://arxiv.org/abs/1711.00549

[22] Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2015. A decision procedure for regular membership and length constraints over unbounded strings. In *International Symposium on Frontiers of Combining Systems*. Springer, 135–150.

[23] Matrix.org Foundation. 2017. Matrix – An open network for secure, decentralized communication. https://matrix.org.

[24] Roy A. Maxion and Robert W. Reeder. 2005. Improving user-interface dependability through mitigation of human error. *International Journal of Human-Computer Studies* 63, 1-2 (jul 2005), 25–50. https://doi.org/10.1016/j.ijhcs.2005.04.009

[25] Tim Moses et al. 2005. Extensible access control markup language (xacml) version 2.0. *Oasis Standard* 200502 (2005).

[26] Robert W. Reeder, Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, Kelli Bacon, Keisha How, and Heather Strong. 2008. Expandable Grids for Visualizing and Authoring Computer Security Policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 1473–1482. https://doi.org/10.1145/1357054.1357285

[27] Ravi S. Sandhu. 1998. Role-based Access Control. (1998), 237–286. https://doi.org/10.1016/s0065-2458(08)60206-5

[28] Seok-Won Seong, Jiwon Seo, Matthew Nasielski, Debangsu Sengupta, Sudheendra Hangal, Seng Keat Teh, Ruven Chu, Ben Dodson, and Monica S. Lam. 2010. PrPl: A Decentralized Social Networking Infrastructure. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond (MCS '10)*. ACM, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/1810931.1810939

[29] D. K. Smetters and Nathan Good. 2009. How Users Use Access Control. In *Proceedings of the 5th Symposium on Usable Privacy and Security (SOUPS '09)*. ACM, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/1572532.1572552

[30] Patrick Stuedi, Iqbal Mohomed, Mahesh Balakrishnan, Z. Morley Mao, Venugopalan Ramasubramanian, Doug Terry, and Ted Wobber. 2011. Contrail: Enabling Decentralized Social Networks on Smartphones. In *Proceedings of the 12th International Middleware Conference (Middleware '11)*. International Federation for Information Processing, Laxenburg, Austria, Austria, 40–59. http://dl.acm.org/citation.cfm?id=2414338.2414343

[31] Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. 2017. Formal analysis of XACML policies using SMT. *Computers & Security* 66, Supplement C (2017), 185 – 203. https://doi.org/10.1016/j.cose.2017.01.009

[32] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a Semantic Parser Overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 1332–1342. https://doi.org/10.3115/v1/p15-1129

[33] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/1629575.1629604

[34] Eric Yuan and Jin Tong. 2005. Attributed based access control (ABAC) for Web services. In *IEEE International Conference on Web Services (ICWS'05)*. 569. https://doi.org/10.1109/ICWS.2005.25

A POLICY CONFORMANCE ALGORITHM

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. The proof of correctness for this algorithm, derived from denotatonial semantics, is also omitted for space reasons.

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

Data: program π: $w, p_{\text{WHEN}} \Rightarrow g, p_{\text{GET}} \Rightarrow d$, a set of compatible policies Π
Result: synthesized program π' that makes π conform to Π
if $\Pi = \emptyset$ **then return** NULL // π has no compatible policies
if $\neg\text{SAT}(\mathcal{L}[\![\pi]\!])$ **then return** NULL // π is null
if $\neg\text{SAT}(\mathcal{L}[\![\pi]\!] \wedge \neg \bigvee_{\pi_i \in \Pi} \mathcal{L}[\![\pi_i]\!])$ **then return** π // π 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 π_i is relevant
  if $\text{SAT}(\mathcal{L}[\![\pi]\!] \wedge \mathcal{L}[\![p_{\text{WHEN},\pi_i}]\!] \wedge \mathcal{L}[\![p_{\text{GET},\pi_i}]\!] \wedge \mathcal{L}[\![p_{\text{DO},\pi_i}]\!])$ **then**
    // check whether p_{WHEN,π_i} is redundant
    if $\neg\text{SAT}(\mathcal{L}[\![\pi]\!] \wedge \neg\mathcal{L}[\![p_{\text{WHEN},\pi_i}]\!])$ **then**
      $p'_{\text{WHEN}} \leftarrow$ `true`
      $p_{\text{WHEN},\pi_i} \leftarrow$ `true`
    end
    else $p'_{\text{WHEN}} \leftarrow p'_{\text{WHEN}}$ `||` p_{WHEN,π_i}
    // check whether p_{GET,π_i} and p_{DO,π_i} are redundant
    if $\neg\text{SAT}(\mathcal{L}[\![\pi]\!] \wedge \mathcal{L}[\![p_{\text{WHEN},\pi_i}]\!] \wedge \neg(\mathcal{L}[\![p_{\text{GET},\pi_i}]\!] \wedge \mathcal{L}[\![p_{\text{DO},\pi_i}]\!]))$ **then** $p'_{\text{GET}} \leftarrow p'_{\text{GET}}$ `||` p_{WHEN,π_i}
    else $p'_{\text{GET}} \leftarrow p'_{\text{GET}}$ `||` $(p_{\text{WHEN},\pi_i}$ `&&` p_{GET,π_i} `&&` $p_{\text{DO},\pi_i})$
  end
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 $p'_{\text{WHEN}} =$ `false` $\vee\ p'_{\text{GET}} =$ `false` **then** // π is inconsistent
  return NULL
end
return π' // π is consistent

B FORMAL GRAMMAR

B.1 ThingTalk Language

Program π:	`source` $= \sigma$, `executor` $= \epsilon : w\ [\Rightarrow g]? \Rightarrow d$
WHEN clause w:	`now` | `monitor` $f([pn = v]^*), p$ | `timer(interval` $= v$`)` | `attimer(time` $= v$`)`
GET clause g:	$f([pn = v]^*), p$
DO clause d:	$f([pn = v]^*)$ | `notify` | `return`

Function f:	$@cn.dn$
Class name cn:	identifier
Function name fn:	identifier
Predicate p:	`true` | `false` | `!`p | p `&&` p | p `||` p | $vn\ op\ v$ | `contains(`vn, v`)` | `substr(`vn, v`)` | `starts_with(`vn, v`)` | `ends_with(`vn, v`)`
Source σ:	SELF | v
Executor ϵ:	SELF | v
Value v:	literal | vn
Parameter name pn:	identifier
Variable name vn:	identifier

B.2 TACL Language

Policy $\hat{\pi}$:	$\hat{p_\sigma} : \hat{w}\ [\Rightarrow \hat{g}]? \Rightarrow \hat{d}$
WHEN clause $\hat{w}$:	`now` | `monitor` $f, \hat{p}$ | `timer,`$\hat{p}$ | `attimer,`$\hat{p}$ | _ | $@cn._$
GET clause $\hat{g}$:	$f, \hat{p}$ | _ | $@cn._$
DO clause $\hat{d}$:	$f, \hat{p}$ | `notify` | _ | $@cn._$
Function f:	$@cn.dn$
Class name cn:	identifier
Function name fn:	identifier
Predicate p:	`true` | `false` | `!`$\hat{p}$ | $\hat{p}$ `&&` $\hat{p}$ | $\hat{p}$ `||` $\hat{p}$ | $vn\ op\ v$ | `contains(`vn, v`)` | `substr(`vn, v`)` | `starts_with(`vn, v`)` | `ends_with(`vn, v`)` | gp
GET predicate gp:	$f([pn = v]^*)\{\hat{p}\}$
Value v:	literal | σ | vn
Variable name vn:	identifier

Received February 2018; revised May 2018; accepted July 2018