G2C: Cryptographic Protocols From Goal-Driven Specifications

Michael Backes[1,2], Matteo Maffei[1], Kim Pecina[1], and Raphael M. Reischuk[1]

[1] Saarland University, Saarbrücken, Germany
[2] Max Planck Institute for Software Systems (MPI-SWS)

Abstract We present G2C, a goal-driven specification language for distributed applications. This language offers support for the declarative specification of functionality goals and security properties. The former comprise the parties, their inputs, and the goal of the communication protocol. The latter comprise secrecy, access control, and anonymity requirements. A key feature of our language is that it abstracts away from *how* the intended functionality is achieved, but instead lets the system designer concentrate on *which* functional features and security properties should be achieved. Our framework provides a compilation method for transforming G2C specifications into symbolic cryptographic protocols, which are shown to be optimal. We provide a technique to automatically verify the correctness and security of these protocols using ProVerif, a state-of-the-art automated theorem-prover for cryptographic protocols. We have implemented a G2C compiler to demonstrate the feasibility of our approach.

1 Introduction

Designing cryptographic protocols is tremendously difficult and error-prone. Protocol designers struggle to keep pace with the variety of possible security vulnerabilities, which have affected early authentication protocols such as the Needham-Schroeder protocol [19,31], carefully designed de facto standards like SSL and PKCS [46,8], and even widely deployed products such as Microsoft Passport [21] and Kerberos [11]. The task of designing cryptographic protocols is made more and more challenging by the dimension and complexity of modern distributed architectures (e.g., collaborative platforms, content sharing applications, social networks) and the number of security properties that have to be simultaneously fulfilled (e.g., user anonymity, access control, secrecy, and authentication). There are only few suitable guidelines [3] or automated tools [36,14,47,14,22,6,40] to assist system designers and, at present, the development of cryptographic protocols is mostly carried out by relying on common practice and on the creativity and experience of designers, rather than on rigorous and formal design techniques.

Recent research has started to address this problem by providing techniques to compile high-level protocol specifications into concrete cryptographic protocols [6,22,40] or to strengthen existing cryptographic protocols and make them resistant to sophisticated threat models [4]. These approaches, however, take as input a detailed specification of the structural aspects of the protocol: one has to describe in depth which messages are exchanged between which participants and, in some cases, even which cryptographic primitives are used. In general, these techniques require expert knowledge in current security research and, arguably, they are hardly accessible to system designers. Ideally, designers should be required to solely state in a simple, yet precise, manner *which* functionality should be realized and *which* security properties should be guaranteed, without necessarily having to think *how* this can be achieved.

1.1 Contributions

Inspired by the increasingly popular approach of declarative networking [30,29,48,34] — a high-level programming paradigm to conveniently describe and implement distributed systems — we propose G2C, a concise, *goal-driven specification language* for distributed applications. G2C allows the designer to specify the functionality of the protocol and the desired security properties (secrecy, access control, and anonymity) without specifying the actual communication patterns or the cryptographic infrastructure, in the spirit of "say what you want, not how to do it." Only the following information has to be specified: the protocol input (given facts like information from some customers), the desired protocol functionality (the survey institute obtains a statistical analysis of the customer's review) and the desired security properties (the customers' review should not leak out and customers should stay anonymous).

We present a *compilation technique* from G2C specifications into Dolev-Yao-style protocols expressed in the applied π-calculus [1]. This compilation is achieved using a combination of standard public-key encryptions and signatures, and, if necessary to achieve anonymity properties, broadcast encryptions [20,9,10] and ring signatures [38,26,12]. Our compiler first generates several candidate protocols and then automatically selects the protocol that minimizes the structural complexity.

We finally present an *automated validation technique* to check the correctness and the security of the synthesized cryptographic protocols using ProVerif [7], a state-of-the-art theorem prover based on Horn-clause resolution that yields security proofs for an unbounded number of protocol sessions. Our compiler embeds ProVerif annotations in the synthesized applied π-calculus code. These annotations allow for the validation of functional correctness, secrecy, and access control. The compiler additionally generates ProVerif bi-processes that capture the intended anonymity properties. In general, this translation validation approach has the advantage that even if we apply drastic optimizations, or completely reimplement the transformation, we do not need to redo any proofs. While a direct proof of correctness of the translation would provide stronger guarantees for any generated protocol implementation without relying on any validator, this far-from-trivial proof would need to be redone every time the transformation is changed, e.g., to apply optimizations or to consider additional security properties. We believe that the added benefits of having such a direct proof are greatly outweighed by the amount of work necessary to create it and keep it up-to-date as the transformation evolves.

1.2 Related work

In declarative network systems, which our approach is inspired by, such as P2/Overlog [30], NDlog [29], SeNDlog [48], the actions for each network node have to be specified. Similarly, in process calculi [32,1,2], it is necessary to specify both source and destination as well as the content of network messages. This holds true also for the number of languages for the specification of multiparty sessions that have been proposed in the last years [17,16,6,22]. In all these approaches, protocol designers have to specify concrete actions for each principal. As shown in the figure below, our approach provides a higher level of abstraction that lets the designer focus on *what* goals should be achieved, without specifying *how* this can be done, i.e., omitting the protocol details.

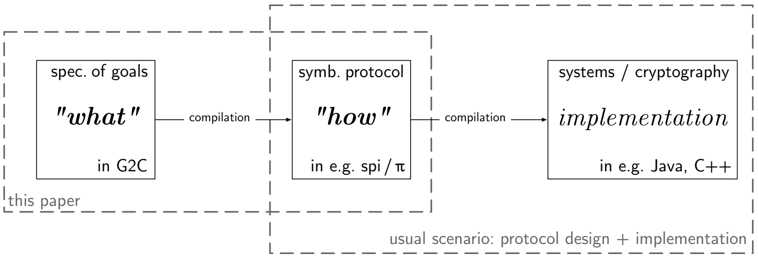


Figure 1: Positioning of our work. While existing approaches concentrate on designing and reasoning about symbolic protocols and corresponding implementations, we provide a more abstract layer for the specification functionality goals and security properties, together with a compilation procedure to symbolic protocols.

A data model [5] that resembles the model we use in G2C has been used to specify the HIPAA privacy rule [43], which regulates the transmission of protected health information by hospitals, doctors, insurance companies, etc. These privacy provisions, stated in terms of logical formulae, can be expressed in our language. Moreover, our framework is fine-grained enough to support both roles and groups of principals. In the HIPAA data model, each statement is dedicated to a single principal. In contrast, our framework allows statements to represent information of several principals (e.g., the result of the statistical analysis of customers' reviews). Jif [33] is a version of Java that incorporates a type system for the enforcement of secure information flow; Jif/Split [14] is an extension of Jif that automatically partitions programs to run securely on a distributed system. Fabric [28] is an extension of Jif/Split that allows new nodes to join the system and supports consistent, distributed computations over shared persistent data. Jif/Split and Fabric deal with confidentiality and integrity, but they do not address anonymity. AURA [27] is a typed language for authorization and audit that includes mechanically verified proofs of the decidability. AuraConf [44] is an extension of Aura that deals with confidentiality properties. In contrast to our approach, neither Aura nor AuraConf focus on the automated generation of cryptographic protocols and do not address anonymity properties. An approach in the area of trust-negotiation protocols [13] is close to our approach of minimizing the cost of a DAG, i.e., minimizing the amount of necessary prerequisites in order to achieve a given goal.

Translation validation [37] is an accepted technique for detecting compiler bugs and preventing incorrect code from being run. Since the validator is usually developed independently from the compiler and uses very different algorithms, translation validation significantly increases the confidence of the user in the compilation process. The validator can use a variety of techniques, ranging from program analysis and symbolic execution [35,42] to model checking and theorem proving [37]. We use ProVerif to validate the results of our compilation.

1.3 Outline

The remainder of this paper is organized as follows. For the sake of exposition, Section 2 introduces our framework by means of an illustrative example, Section 3 presents the compilation to symbolic protocols. Section 4 explains how anonymity is achieved and verified. Finally, Section 5 concludes.

2 Illustrative Example

The language G2C is best explained by means of an illustrative example of a goal-driven G2C specification. (We refer to Appendix A for a formal grammar of the G2C syntax.) In this example, information about different topics is collected from a set of customers (collection phase). This information is evaluated, i.e., using some statistic analyses by a manager (evaluation phase), and then sent to a survey institute that can publish the final document (publication phase). Among other functional goals and security constraints, the goal of this protocol is to ensure (1) anonymity of the customers who initially have some private or confidential information and (2) anonymity of the manager who evaluates the collected information. In this case, the final document can only be signed by a member of a pool of trustworthy managers, but the survey institute shall not know who the responsible manager is who actually signed the document.

Principals. The G2C specification for such a protocol defines a set of principals $\mathcal{P}$ occurring in the system. Each principal is assigned one or more tags $t_i \in \mathcal{T}$. By default, each principal $p \in \mathcal{P}$ is implicitly tagged by a tag with his own name p, i.e., the set of tags $\mathcal{T}$ is implicitly extended to comprise $\mathcal{P}$ if necessary.

For this example, there are some customers tagged `customer`, and some managers tagged `manager`, and the survey institute tagged `government`:

```
Principals:
  cust1 : customer
  cust2 : customer
  ...
  mng1  : manager
  mng2  : manager
  ...
  surveyinstitute : government
```

Tags. Both principals and statements (see below) are tagged. These tags can be related via a partial-order relation, a *tag lattice* with a least element `public`, which provides an access control mechanism for statements: principal p tagged t_p may only access statements tagged $t_s \leq t_p$. Intuitively, the higher the position of a tag $t_s \in \mathcal{T}$ in the lattice, the more confidential are the statements tagged by t_s. The usage of tags allows us to build upon several role-based access control mechanisms [18,45,39]. The presented example does not use an explicitly specified lattice, only the implicit relation $\forall\, t \in \mathcal{T}.\, \mathtt{public} \leq t$.

Statements. The specification defines a set $\mathcal{S}$ of statements that occur in the system. Statements can be considered as place-holders for the actual values in a protocol run. At specification time, these values are irrelevant in the sense that they do not affect the protocol construction. For this reason, we symbolically abstract values as statements. The syntax of a statement specification is of the form `s : t`, where $\mathtt{s} \in \mathcal{S}$ and $\mathtt{t} \in \mathcal{T}$. Arguments can be constants (lowercase strings and numbers), variables (strings beginning with an uppercase letter) or wildcards (*). The tags on the right-hand side of the colon can be either constants or variables that are bound in the argument list of the specified statement.

```
Statements:
  document(2011) : manager or government
  info(*)        : customer or manager
  manager_pwd()  : manager
```

The statements in this case are: `document(2011)`, which represents the final document that is created by the managers for the year 2011. It is accessible for all principals that are tagged `manager` or `government`. The statements called `info(topic)` contain the information that is collected in the collection phase for a specific topic. These statements are accessible for customers and managers. They can be parameterized to specify which particular information the statement contains. The statement `manager_pwd()` is a password that is necessary to compute and trustworthily sign the document. Its content is only accessible for those principals that are tagged `manager`.

Inputs. Initially, each principal $\mathtt{p} \in \mathcal{P}$ has some input statement $\mathtt{s} \in \mathcal{S}$ to the system. These inputs are captured in the input section. The syntax is `s @ p`. In our example, the customers have information about certain topics, and all managers have the manager password.

```
Input:
  info(*)        @ $PRINCIPALS_TAGGED(customer)
  manager_pwd()  @ $PRINCIPALS_TAGGED(manager)
```

The `G2C` language supports some syntactic sugar. The input specification expression `s @ $PRINCIPALS_TAGGED(t)` is evaluated to a list of input specifications $\mathtt{s}\,@\,\mathtt{p}_1$, $\mathtt{s}\,@\,\mathtt{p}_2, \ldots$, where $\mathtt{p}_i \in \mathcal{P}$ are the principals that are declared to have tag `t`. Statements may be parameterized by constants and wildcards.

Functional goals. A *functional* goal of the protocol is to make the document available to the customers. The goal section states these goals by listing statements at principals. It is syntactically similar to the input section.

```
Goals:
  document(2011)      @ surveyinstitute
```

Rules. In order to enforce the stated functional goals, a set of computation rules $\mathcal{R}$ has to be specified. In the section for rules, computations are abstractly specified using arbitrary function symbols like `create_document`. The intuition behind rules is that anyone can compute the head of a rule (in this case the statement `document`) whenever all computation arguments are available (in this case the manager password and the information about the topics).

```
Rules:
  document(2011) :- create_document[
                     info(topic1), info(topic2), ...,
                     manager_pwd()
                    ]
```

More formally, a computation rule $r \in \mathcal{R}$ is a variant of a Horn clause of the form h `:-` $f[b_1, \ldots, b_n]$. The *head* h of r is a statement possibly parameterized by constants or by variables. The right-hand side of the rule operator `:-` contains the *body* of r. The body comprises a function symbol f and a list of comma separated statements b_i. These statements can be parameterized by constants and variables. The variables must be bound as parameters of h. We stress that, for the reason of abstraction, the specification does not take the semantics for the specified function symbols into account. Any principal p can compute h whenever p knows *all* statements b_i. For each rule, our compiler automatically derives the set of principals that can apply that rule.

Anonymity. The *security* goals supported by G2C are secrecy and access control (as specified by the statement tags) and anonymity. The latter is captured in the anonymity section. An anonymity specification is a tuple $(s, \mathcal{A}, \mathcal{F})$. The first component $s \in \mathcal{S}$ is a statement. The second component $\mathcal{A} \subseteq \mathcal{P}$, the *among-set*, is a set of principals that shall be anonymous among each other. The third component $\mathcal{F} \subseteq \mathcal{P}$, the *for-set*, is a set of principals that shall not be able to distinguish who in the among set is involved in the computation of s. This notion of anonymity is similar to the concept of k-anonymity [41,15], which states the impossibility to determine who among k users is active. Our definition is more fine-grained in that it specifies the action with respect to which the user should stay anonymous and also the intended distinguishers (implicitly, the for-set additionally includes external observers eavesdropping the communication).

```
Anonymity:
  document(2011) among { cust1,cust2,... } for { surveyinstitute }
  document(2011) among { mng1,mng2,... }   for { cust1,cust2,... }
  document(2011) among { mng1,mng2,... }   for { surveyinstitute }
```

Intuitively, the first of the above specifications means that in the final document, all customers shall be anonymous *among* each other *for* the survey institute. This implies that the survey institute shall not be able to distinguish whether `cust1` or whether `cust2` was involved in the final document. The second and third specifications demand that the managers be anonymous for the customers and the survey institute, respectively. In other words, neither a customer, nor the survey institute shall learn who the actual manager is that has created and signed the document.

3 Compilation to Symbolic Protocols

In the first step (Section 3.1), an intermediate representation of the specification, a so called data flow graph, is generated. This graph is constructed based on the specified goals, the input patterns, and the corresponding computation rules. The access control specifications for the declared statements are also considered in order to prevent the graph from growing too fast. In the second step (Section 3.2), the data flow graph is condensed in that optimal nodes and edges are selected with respect to the overall communication complexity the final protocol would have. In the third step (Section 3.3), the paths of the condensed graph are translated into a cryptographic protocol expressed in the applied π-calculus.

3.1 Intermediate representation as data flow graphs

This section formally defines data flow graphs. Data flow graphs serve as an intermediate representation of the protocol specification where nodes represent knowledge of principals, and edges represent the flow of knowledge between principals (i.e., the communication patterns). This data structure provides all necessary information for generating a cryptographic protocol in a symbolic calculus. Data flow graphs are constructed by an iterative bottom-up procedure, which is best explained using the illustrative example of Section 2.

Example graph. The data flow graph for the example of Section 2 is illustrated in Figure 2. Data flow graphs are constructed in a bottom-up manner, starting with the specified *goal nodes* (doubly circled), which are added to an exploration queue of nodes that have to be explored further. For each node from the queue, the following three exploration steps are performed:

(1) Possible flows from other *knowledge nodes* (round shape) are considered in case the access control specifications permit these flows. In the example, there are flows from the managers' knowledge nodes, document(2011)@mng1 and document(2011)@mng2. These knowledge nodes, if not existent yet, are created within this first step and then added to the exploration queue.

(2) If the statement of the currently explored node is an instantiation of the head statement of a computation rule, a new *computation node* (rectangular shape) is created and added to the queue. Moreover, the inputs for such a computation, again knowledge nodes, are created and added to the queue.

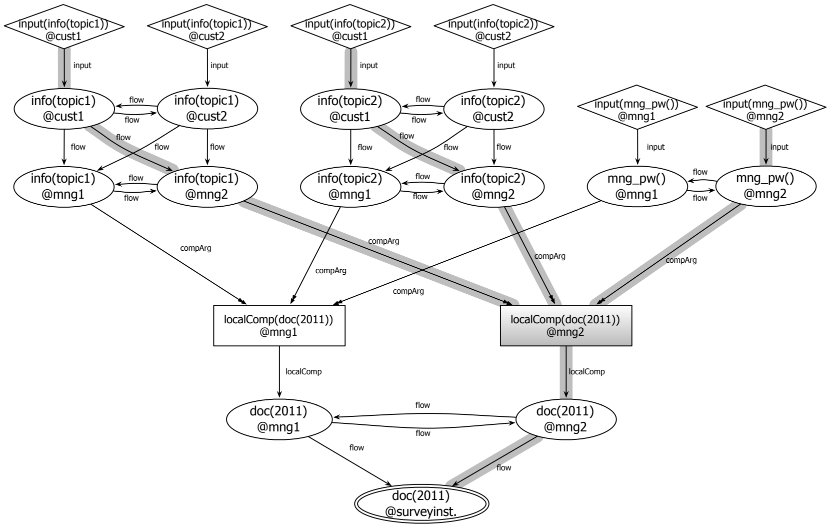


Figure 2: Data flow graph for the example of Section 2. For the sake of readability, the graph contains only two customers, two managers, and only information about two topics; some statements are abbreviated; the costs for edges and computations are omitted. An optimal selection of edges is colored in gray (cf. Section 3.2).

In the example above, the computation nodes are localComp(document(2011)@mng1) and localComp(document(2011)@mng2). The inputs to those computation nodes are knowledge nodes representing information about certain topics and the manager password.

(3) The statement of the currently explored node is matched against the input patterns that are provided in the specification. If an instantiation of an input pattern matches the node statement, a new *input node* (diamond shape) is created. This node is not added to the queue as it is not explored further.

Due to this bottom-up construction, and to the immediate instantiations of variables, nodes with neither preceding computation node nor input node might be added to the queue. Therefore, in the end, subgraphs that have no input node as ancestors are removed from the graph.

The edges of the data flow graph express possible *communication structures* of the synthesized protocol. Edges labeled *flow* connect two knowledge nodes that consist of the same statement, but at different principals. These edges correspond to messages that are sent in the synthesized cryptographic protocol. Edges labeled *localComp*, *compArg*, or *input* are virtual edges — they do not represent actual network messages.

Data flow graphs, formally. A data flow graph consists of a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$. The nodes are split into three disjoint sets: *input* nodes $\mathcal{N}_i \subseteq \{\text{input}(S@P)\}_{S\in\mathcal{S},P\in\mathcal{P}}$, *knowledge* nodes $\mathcal{N}_k \subseteq \{S@P\}_{S\in\mathcal{S},P\in\mathcal{P}}$, and *computation* nodes $\mathcal{N}_c \subseteq \{\text{localComp}(S@P)\}_{S\in\mathcal{S},P\in\mathcal{P}}$. A subset of the knowledge nodes $\mathcal{N}_{goal} \subseteq \mathcal{N}_k$ is called *goals*. The set of edges is split into the disjoint sets of input edges $\mathcal{E}_i$, flow edges $\mathcal{E}_f$, computation argument edges $\mathcal{E}_a$ and computation result edges $\mathcal{E}_r$.

Nodes. A data flow graph for a given specification is the smallest graph satisfying the following rules. The numbers refer to the aforementioned explorations.

$$\frac{\mathrm{input}(S^*@P) \in \texttt{SPEC} \qquad S \mathrel{\overset{\triangleleft}{\sim}} S^*}{\mathrm{input}(S@P) \in \mathcal{N}_i}\ \textsc{NInput}^{(3)} \qquad\qquad \frac{\mathrm{input}(S@P) \in \mathcal{N}_i}{(S@P) \in \mathcal{N}_k}\ \textsc{NKnowInput}^{(3)}$$

$$\frac{\begin{array}{c}\mathrm{rule}(S^* \leftarrow f(S_1^*, \ldots, S_n^*)) \in \texttt{SPEC} \\ \forall i.\, (S_i@P) \in \mathcal{N}_k \qquad \langle S, S_1, \ldots, S_n \rangle \mathrel{\overset{\triangleleft}{\sim}}_{\mathcal{R}} S^* \leftarrow f(S_1^*, \ldots, S_n^*)\end{array}}{\mathrm{localComp}(S@P) \in \mathcal{N}_c}\ \textsc{NComputation}^{(2)}$$

$$\frac{\mathrm{goal}(S@P) \in \texttt{SPEC}}{(S@P) \in \mathcal{N}_{goal}}\ \textsc{NGoal}^{(init)} \qquad\qquad \frac{\mathrm{localComp}(S@P) \in \mathcal{N}_c}{(S@P) \in \mathcal{N}_k}\ \textsc{NKnowComp}^{(2)}$$

$$\frac{(S@P') \in \mathcal{N}_k \qquad \mathrm{may\_access}(P, S)}{(S@P) \in \mathcal{N}_k}\ \textsc{NKnowFlow}^{(1)}$$

NInput introduces input nodes for instantiated statements $S@P$ that match an input pattern $S^*@P$ from the specification. $S \mathrel{\overset{\triangleleft}{\sim}} S^*$ denotes that S is a statement instantiation of S^*, i.e., wildcards `*` and variables in S^* are consistently instantiated, and $\mathrel{\overset{\triangleleft}{\sim}}_{\mathcal{R}}$ denotes a rule instantiation The details of the instantiation procedure are described in the long version [23]. We stress that the instantiation in NInput is minimal in the sense that only those statements are generated that are necessary to reach the goals (this prevents the graph from growing more than required). Hypotheses of the form $X \in \texttt{SPEC}$ assume X to occur in the `G2C` specification. NKnowInput creates knowledge nodes from input nodes. NComputation creates computation nodes for existing knowledge nodes $S_i@P$ and a computation rule occurring in the specification. NGoal directly introduces goal nodes from the specification. NKnowComp creates knowledge nodes from computation nodes. NKnowFlow creates knowledge nodes from existing knowledge nodes if the access control specification permits this step, i.e., the premise may_access(P, S) holds.

Edges. We define the edges $\mathcal{E}$ as the smallest set satisfying the following rules:

$$\frac{n_i = \mathrm{input}(S@P) \in \mathcal{N}_i \qquad n_k = (S@P) \in \mathcal{N}_k}{(n_i, n_k) \in \mathcal{E}_i}\ \textsc{EInput}^{(3)}$$

$$\frac{n_1 = (S@P_1) \in \mathcal{N}_k \qquad n_2 = (S@P_2) \in \mathcal{N}_k}{(n_1, n_2) \in \mathcal{E}_f}\ \textsc{EFlow}^{(1)}$$

$$\frac{\begin{array}{c}n_k = (S@P) \in \mathcal{N}_k \qquad n_c = \mathrm{localComp}(S'@P) \in \mathcal{N}_c \\ \mathrm{rule}(S^* \leftarrow f(S_1^*, \ldots, S_n^*)) \in \texttt{SPEC} \qquad S' \overset{\triangleleft}{\sim} S^* \qquad \exists i : S \overset{\triangleleft}{\sim} S_i^*\end{array}}{(n_k, n_c) \in \mathcal{E}_a}\ \textsc{ECompArg}^{(2)}$$

$$\frac{n_c = \mathrm{localComp}(S@P) \in \mathcal{N}_c \qquad n_k = (S@P) \in \mathcal{N}_k}{(n_c, n_k) \in \mathcal{E}_r}\ \textsc{ECompRes}^{(2)}$$

Given an input node and a knowledge node, EInput creates an input edge (labeled *input*). EFlow creates flow edges between knowledge nodes (labeled *flow*). Computation argument edges (labeled *compArg*) from knowledge node n_k to computation node n_c are introduced by ECompArg in case the statement of n_k is a valid instantiation of an argument of the computation rule contained in n_c. The knowledge node representing the result of a computation is connected via ECompRes.

We stress that data flow graphs are finite since there are only finitely many statements, finitely many principals, and finitely many constants.

3.2 Condensed data flow graphs – selection of the protocol skeleton

The idea behind condensing a data flow graph is to find a minimal subset $E \subseteq \mathcal{E}$ of edges, such that all goal nodes are *active* in E. Informally, a knowledge node n_k is active if at least one direct predecessor of n_k is active, a computation node n_c is active if all predecessors of n_c are active, input nodes are always active. In Figure 2, both computation nodes localComp(document(2011)@mng1) and localComp(document(2011)@mng2) are active since all necessary inputs are available and hence, all predecessor knowledge nodes are active. For the same reason, the goal node document(2011)@surveyinstitute is active as well. The *selected* subset of edges is depicted with a gray background. There are several other subsets of edges that make the goal node active. For example, principal cust2 could also give his inputs; or principal mng1 could provide the password.

Besides the condition that, in the final synthesized protocol, all goal nodes must be active, we require that the final protocol be minimal, i.e., the message complexity of the final protocol must not exceed the complexity of the synthesized protocol that has lowest complexity. This minimal subset of edges is referred to as *protocol skeleton* or *condensed data flow graph.* It constitutes the communication structure of the synthesized protocol.

Message complexity. In order to optimize the communication and computation complexity of synthesized protocols, we show a measure of the complexity for sent messages and for computed statements. Computing the minimal number of messages sent around the network requires to take reuse of computed information into account. The problem can be formulated as the task of finding a spanning tree, an acyclic subset of the edges from the data flow graph, such that all goal nodes can be computed with minimal cost. This problem is a classical planning problem [25], mostly investigated by the AI community.

The precise optimization problem *MinMsgCplx*$(\mathcal{G})$ is defined as follows. Given a directed acyclic flow graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which the nodes are split into two disjoint sets: $\mathcal{N} = \mathcal{N}_k \uplus \mathcal{N}_{ci}$ with $\mathcal{N}_{goal} \subseteq \mathcal{N}_k$ and $\mathcal{N}_{ci} = \mathcal{N}_c \uplus \mathcal{N}_i$. Intuitively, a node $n \in \mathcal{N}_k$ depends on only one of its predecessor nodes, whereas $n \in \mathcal{N}_{ci}$ depends on all predecessor nodes. Each edge $e \in \mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is assigned a cost $c(e) \in \mathbb{N}$. The goal is to compute a *valid* set $E \subseteq \mathcal{E}$ such that $\sum_{e \in E} c(e)$ is minimal. A set $E \subseteq \mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is valid (i.e., it fulfills the *graph constraints*) if

1. $\forall\, n \in \mathcal{N}_{goal} : n$ is active in E.
2. $\forall\, n \in \mathcal{N}_k$ active in E, $\exists\, m \in \mathcal{N} : (m, n) \in E$.
3. $\forall\, n \in \mathcal{N}_{ci}$ active in E, $\forall\, m \in \mathcal{N} : (m, n) \in \mathcal{E} \Rightarrow (m, n) \in E$.

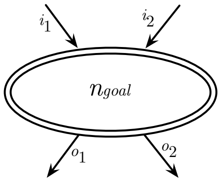
where $n \in \mathcal{N}$ is *active* in E whenever $\exists\, m \in \mathcal{N}$ with $(n, m) \in E$ or $(m, n) \in E$.

Define the decision problem *MsgCplx*$(\mathcal{G}, c^*)$ that asks whether there is a subset of edges $E \subseteq \mathcal{E}$ satisfying $\sum_{e \in E} c(e) \leq c^*$.

Theorem 1. *MsgCplx is* $\mathcal{NP}$*-complete.*

We refer to Appendix B for the proof.

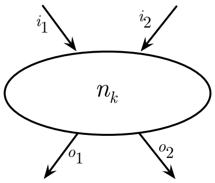
Graph constraints in SAT. In order to select nodes and edges, thereby enforcing the graph constraints, the data flow graph is translated into a satisfiability problem with clauses in disjunctive normal form. Each variable v_e of the SAT problem represents an edge $e \in \mathcal{E}$. Iff in a satisfying assignment of variables, variable v_e is true, then e is selected. From all satisfying assignments of variables, the assignment that minimizes $\sum_{e \in \mathcal{E}} e \cdot c(e)$ is eventually chosen as communication structure for the final protocol. Our framework uses a state-of-the-art constraint solver, *Gecode* [24], in order to obtain an optimal solution.



The translation of the graph constraints corresponds closely to the above definition of valid edge sets. For demonstration issues, we assume only two incoming edges and two outgoing edges per node. (1) For goal nodes with incoming edges i_1 and i_2, we post the constraint

$$i_1 \vee i_2$$

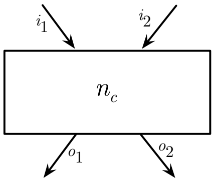
since at least one of i_1 and i_2 must be active in order to activate n_{goal}. The outgoing edges are not necessary to activate n_{goal}.



(2) The incoming edges of a knowledge node n_k shall only be active if at least one of the outgoing edges is active. For each outgoing edge o_j, we post the constraint

$$o_j \rightarrow (i_1 \vee i_2)$$

which represents the clause $\overline{o}_j \vee i_1 \vee i_2$.



(3) The scenario for computation nodes is slightly more complex. For a computation node n_c to be active, *all* incoming edges, hence all direct predecessors of n_c must be active. For each outgoing edge o_j, we post the constraint

$$o_j \rightarrow (i_1 \wedge i_2)$$

which is equivalent to the formula $\overline{o}_j \vee (i_1 \wedge i_2)$. This formula is translated to two clauses $\overline{o}_j \vee i_1$ and $\overline{o}_j \vee i_2$.

Cycles. Consider goal nodes u and v that are connected via two flow edges (u, v) and (v, u). The constraints for knowledge nodes are satisfied if both theses edges are active. Such cycles, of course, do not solve the problem of activating goal nodes. In order to prevent these cycles in the SAT instance, we post more constraints: Given knowledge node n_k with incoming flow edge i, for each outgoing flow edge o, we add the constraint $i \rightarrow \overline{o}$, which corresponds to the clause $\overline{i} \vee \overline{o}$. This prevents node n_k from acting as "knowledge forwarding" node. Edges from input nodes are not considered, nor are edges from computation nodes.

3.3 Synthesizing cryptographic protocols for the applied π-calculus

We now detail the translation from a condensed data flow graph into a cryptographic protocol in the applied π-calculus [1]. We build a process for every principal P involved in the protocol. The final protocol consists of the parallel execution of all principal processes, i.e., all principals run concurrently. We assume all principals to be semi-honest, i.e., principals follow the protocol properly, but are curious in that they attempt to learn additional information.

First, we describe the translation of the specified functional goals (together with their annotations for the ProVerif validation) and the security properties. Second, we show how the obtained processes can be modified in order to validate the desired anonymity requirements in ProVerif.

- ∗ An *input* with corresponding input node input(s@p) is expressed in the process of principal p as a restriction of a fresh name s. This captures the intuition that, initially, the statement s is only known to principal p. Moreover, the event event s is raised, which states that the input has taken place. Events will be used later in order to show that a computation can only take place if all proper inputs are available.
- ∗ A *computation* at node localComp(s@p) with function symbol f and arguments arg1,arg2,... is translated for the process of principal p into a constructor application f(arg1,arg2,...). Every such application is followed by the event event s in order to track that computation.
- ∗ A *goal* with corresponding goal node s@p raises the event event s. This is necessary to verify the reachability property of functional goals (see below).

For the purpose of validation, we insert correspondence queries for each computation: every computation `h :- func[`b_1`,...,`b_ℓ`]` in the G2C specification is translated into a ProVerif query of the form query ev:h ==> ev:b_1 & . . . & ev:b_ℓ. to validate the functional goals of our protocols. Since event h and the corresponding symbolic term must be preceded by all the events b_1 to b_ℓ along with the corresponding symbolic terms, such queries ensure that all computations are executed only with the expected inputs. For each goal s@p, we insert a query of the form query ev:s. If ProVerif successfully validates all queries, then all computations are well-formed and all goal nodes are reachable.

Flow edges are the only edges that are explicitly modeled in the symbolic protocol. These flows represent the actual communication over a public network. Loosely speaking, the sender first signs the message to ensure its integrity and then encrypts the resulting signature for the recipient to protect the statement's confidentiality. If the G2C specification includes anonymity requirements, then the implementation of the flow edges relies on more sophisticated cryptographic primitives, as detailed in the next section. For the validation of secrecy for statements, we use ProVerif's standard secrecy queries.

4 Anonymity

Anonymity is a security property that reasons about *knowledge* that principals have or do not have. Intuitively, a principal p is anonymous while performing some action α if no other principal p' has any knowledge about that action. If p' has the knowledge that action α took place, but if p' is not able to determine which principal from a set $\mathcal{A}$ of principals actually performed that action, then we say that p is anonymous in action α *among* $\mathcal{A}$ *for* p'. If any subset of principals $\mathcal{F}$ is not able to distinguish individual principals from $\mathcal{A}$ in action α, we say that principal p is anonymous in α among $\mathcal{A}$ for $\mathcal{F}$. In the following, we call $\mathcal{A}$ the *among*-set and we call $\mathcal{F}$ the *for*-set.

4.1 Anonymity as symmetric paths in the graph

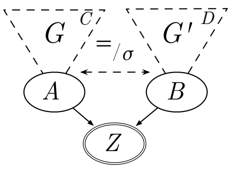
Intuitively, the anonymity requirement $(s, \mathcal{A}, \mathcal{F})$ is fulfilled if for each pair of principals p, p' in $\mathcal{A}$, there exist two subgraphs, both leading to goal s, which are equal up to the identities of p and p'. More precisely, if p is active in goal s, i.e., p contributes to the goal nodes with statement s, then for each other principal $p' \in \mathcal{A}$, there must be another subgraph, such that, after replacing all principals not in the among-set, nor in the for-set by a special symbol $\sharp$, and after replacing the two compared principals p and p' by a special symbol $\bullet$, the corresponding subgraphs are equal.

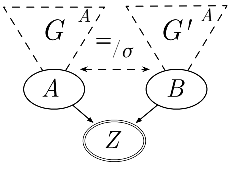
For a more formal definition, let $\mathcal{N}_{goal(s)}$ be the set of goal nodes in which statement s occurs. Let G be a minimal subgraph such that all goal nodes are active (as described in Section 3.2). For all goal nodes $n \in \mathcal{N}_{goal(s)}$, for all $p, p' \in \mathcal{A}$ with $p \neq p'$, we check that one of the following cases is satisfied:

1. p is inactive in n, i.e., for all edges $e_i = (u, v) \in G$ that are (not necessarily direct) ancestor edges of n, we have $prin(u) \neq p \neq prin(v)$, where $prin(s@p) = p$ is the principal for node $(s@p)$.
2. p is active in n and there exists a subgraph G' in which p' is active so that for $q_i \in \mathcal{P} \setminus (\mathcal{A} \cup \mathcal{F})$ the following subgraphs are equal:

$$G\left\{{}^{\sharp}/_{q_1}\right\}\ldots\left\{{}^{\sharp}/_{q_\ell}\right\}\left\{{}^{\bullet}/_{p}\right\} = G'\left\{{}^{\sharp}/_{q_1}\right\}\ldots\left\{{}^{\sharp}/_{q_\ell}\right\}\left\{{}^{\bullet}/_{p'}\right\}$$

The substitution $\mathrm{G}\{{}^{a}/_{b}\}$ replaces all occurrences of b in G by a. This affects statements and principals.





Example. Consider the examples depicted on the right. There are two ways of achieving the goal Z. If $\mathcal{A} = \{A, B\}$, we require that the subgraphs G and G' be equal up to the identities of A and B. For the first example, this requires among others that principals C and D do not occur in $\mathcal{F}$: they know whether they were involved in the protocol. In this case, the substitution σ replaces all occurrences of C and D by $\sharp$ before the subgraphs are compared. For the second example, as both A and B occur in $\mathcal{A}$, σ replaces only the occurrences of A and B by $\bullet$. This means that (although only A occurs in both G and G'), the resulting subgraphs are equal.

4.2 Cryptographic primitives for anonymity

Digital signatures and encryption schemes preserve the integrity and the privacy of data, respectively. In general however, these primitives do not suffice to enforce anonymity requirements. For instance, a digital signature immediately reveals the signer's identity and a ciphertext may reveal the intended recipient. We address these issues by deploying *ring signatures* [38,26,12] and *broadcast encryptions* [20,9,10].

A ring signature preserves the integrity of the signed message but the signer remains anonymous within a chosen group of people. More formally, the signer first decides on the ring he will use. A ring is an arbitrary group of people including the principal himself. He collects the (public) verification keys of all ring members and his own signing key. The resulting ring signature reveals only the fact that one person in the ring signed the message but does not reveal the actual signer. Ring signatures are thus a salient tool to protect the anonymity of principals in the among-set when sending messages to principals in the for-set (referred to as *forward anonymity* in the following). More formally, given an anonymity specification $(s, \mathcal{A}, \mathcal{F})$, whenever communication takes place between a principal p_A in $\mathcal{A}$ and a principal p_F in $\mathcal{F}$, p_A uses a ring signature where the ring consists of all principals from $\mathcal{A}$. We model ring signatures in the applied π-calculus with the constructor $\texttt{r}n\texttt{sign(m,sk}_1\texttt{,vk}_2\texttt{,}\ldots\texttt{,vk}_n\texttt{)}$ where $\texttt{(vk}_i\texttt{,sk}_i\texttt{)}$ denotes the i-th ring member verification key/signing key pair. The ring signature verification $\texttt{r}n\texttt{check(s,vk}_1\texttt{,vk}_2\texttt{,}\ldots\texttt{,vk}_n\texttt{)}$ succeeds and returns $\texttt{m}$ if and only if $\texttt{s} = \texttt{r}n\texttt{sign(m,sk}_1\texttt{,vk}_2\texttt{,}\ldots\texttt{,vk}_n\texttt{)}$. Here, the first principal signs the message. However, by extending the destructor reduction rules to handle permutations, we allow the actual signer to occur on arbitrary position.

Dually, we use broadcast encryption schemes to protect the anonymity of principals within the among-set when receiving messages from principals within the for-set (*backward anonymity*). More precisely, given an anonymity specification $(s, \mathcal{A}, \mathcal{F})$, when a principal $p_F \in \mathcal{F}$ communicates a message to a principal $p_A \in \mathcal{A}$, then p_F is required to use a broadcast encryption involving the public encryption keys of all principals in $\mathcal{A}$. This encryption ensures that the ciphertext is addressed to the correct set of principals and that the same plaintext is broadcast to all those principals. One might be tempted to simply require principals in the for-set to issue one encryption for each member of the among-set. A corrupted sender, however, could send only one encryption for a single principal in the among-set thus exploiting a termination channel. Alternatively, the sender could send different messages to different principals and then determine the active principal's identity by scrutinizing the output of a computation. In principle, it is also possible to use zero-knowledge proofs to counter the aforementioned attacks. However, broadcast encryptions entail a significantly lower computational overhead while achieving the same goal: the sender only creates one single ciphertext and the decryption will reveal if that ciphertext was indeed addressed to the proper group of people. For instance, the broadcast encryption scheme by Boneh et al. [9] requires that the encryption keys of all among-set members be available: we model broadcast encryptions with the constructor $\texttt{b}n\texttt{enc(m,ek}_1\texttt{,ek}_2\texttt{,}\ldots\texttt{,ek}_n\texttt{)}$ where $\texttt{(ek}_i\texttt{,dk}_i\texttt{)}$ denotes the i-th principal's encryption key/decryption key pair. The decryption $\texttt{b}n\texttt{dec(e,ek}_1\texttt{,dk}_2\texttt{,}\ldots\texttt{,ek}_n\texttt{)}$ succeeds and returns $\texttt{m}$ if and only if $\texttt{e} = \texttt{b}n\texttt{enc(m,ek}_1\texttt{,ek}_2\texttt{,}\ldots\texttt{,ek}_n\texttt{)}$. As seen for ring signatures, the destructor is applicable independently of the position of the party decrypting the ciphertext.

We finally remark that both ring signatures and broadcast encryptions are very efficient cryptographic schemes.

Example. Let us exemplify the notions of forward and backward anonymity on the anonymity specification from Section 2:

```
  Anonymity:
    document(2011) among { cust1, cust2 } for { surveyinstitute }
    document(2011) among { mng1, mng2 }   for { cust1, cust2 }
    document(2011) among { mng1, mng2 }   for { surveyinstitute }
```

For the first specification, as the customers never directly communicate with the survey institute, we do not take special precautions: we assume that only principals listed in the for-set are corrupted and thus the managers do not reveal the identity of the customers. Hence, the privacy offered by encryption schemes is sufficient to conceal the identity of the originator of a message.

The second specification requires the customers to use a broadcast encryption (backward anonymity): the manager will receive the input directly from the customers who should not be able to distinguish one manager from another. The simplified applied π-calculus code looks as follows:

```
Customer 1:
  ... new topic1;   (* input *)
  out(c, b2enc(sign(topic1,sk_cust1),ek_mng1,ek_mng2)); ...

Manager 2:
  ... in(c, esv_topic1);   (* signed-then-encrypted *)
  let sv_topic1 = b2dec(esv_topic1,ek_mng1,dk_mng2) in
  let  v_topic1 = check(sv_topic1,vk_cust1) in ...
```

Since the customer is not required to remain anonymous for the managers, it is sufficient for him to use a digital signature rather than a computationally more involved ring signature.

The third specification demands that the survey institute does not learn which manager collected the customer data (forward anonymity); the active manager uses a ring signature where the ring comprises all managers:

```
Manager 2:
  ... let document = create_document(topic1,topic2,manager_pwd) in
  out(c, enc(r2sign(document,vk_mng1,sk_mng2),ek_surveyinst)); ...

Survey Institute:
  ... in(c, esv_document);   (* signed-then-encrypted *)
  let sv_document = dec(esv_document,dk_surveyinst) in
  let  v_document = r2check(sv_document,vk_mng1,vk_mng2) in ...
```

4.3 Verification of anonymity in the synthesized protocol

To verify the given anonymity specification, we use ProVerif's so-called *choice operator*. Intuitively, this operator allows us to model two processes that are structurally equal but differ only in certain terms. The resulting process is called a bi-process. We check that the attacker cannot distinguish the two executions in such a bi-process. Therefore, we let all the principals in $\mathcal{F}$ be corrupted, i.e., we let them release all their secrets and let them take no further action. The attacker can thus act arbitrarily on behalf of those principals. We then pick two principals I, J from $\mathcal{A}$ and construct a bi-process where one choice corresponds to I's code and the other corresponds to J's code. The graph generation algorithm ensures that the two processes are structurally equal (cf. Section 4.1) and that they can be cast into a bi-process. As we verify an equivalence relation, for each anonymity specification, we only consider a chain of relations and use transitivity to obtain observational equivalence among all pairs of principals in $\mathcal{A}$.

Example. Let us consider again the above example: the two managers must remain anonymous for the survey institute. Thus, we cast both managers into a bi-process. The left process corresponds to manager 1 interacting in the protocol and the right process corresponds to manager 2 giving input to the survey institute. As the survey institute occurs in $\mathcal{F}$, we assume it to behave arbitrarily, and we hence let the attacker impersonate the survey institute.

```
Manager 1+2:
  let document = create_document(topic1,topic2,manager_pwd) in
  out(c, enc(r2sign(document,choice[sk_mng1,vk_mng1],
                             choice[vk_mng2,sk_mng2]),ek_surveyinst));

Survey Institute:
  out(c, (vk_surveyinst,sk_surveyinst));
  out(c, (ek_surveyinst,dk_surveyinst)).
```

5 Conclusions and Future Work

We have presented a novel high-level goal-driven specification language, called G2C, that offers support for the declarative specification of functionality goals and security properties. We have presented an automated compilation technique for transforming G2C specifications into corresponding cryptographic protocols, using a combination of public-key encryption, digital signatures, broadcast encryption, and ring signatures. The specified functionality goals as well as the secrecy and anonymity properties are automatically validated using ProVerif.

Drawing on ideas from the vast amount of existing work on authentication and authorization we plan to incorporate further features such as delegation and revocation mechanisms, which are notoriously difficult to combine with privacy and anonymity properties. This extension will naturally involve the usage of more sophisticated cryptographic primitives, such as zero-knowledge proofs. We additionally intend to consider further security properties, such as differential privacy.

References

1. M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *POPL'01*, 2001.
2. M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *CCS'97*. ACM, 1997.
3. M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Softw. Eng.*, 22(1), 1996.
4. M. Backes, M. P. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. In *CSF'09*. IEEE, 2009.
5. A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and Contextual Integrity: Framework and Applications. In *SP'06*. IEEE, 2006.
6. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF'09*. IEEE, 2009.
7. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW'01*, 2001.
8. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *CRYPTO'98*, volume 1462 of *LNCS*. Springer, 1998.
9. D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *CRYPTO'05*, volume 3621 of *LNCS*. Springer, 2005.
10. D. Boneh and B. Waters. A fully collusion resistant broadcast, trace, and revoke system. In *CCS'06*. ACM, 2006.
11. F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1), 2006.
12. N. Chandran, J. Groth, and A. Sahai. Ring signatures of sub-linear size without random oracles. In *ICALP'07*, volume 4596 of *LNCS*. Springer, 2007.
13. W. Chen, L. Clarke, J. Kurose, and D. Towsley. Optimizing cost-sensitive trust-negotiation protocols. In *INFOCOM'05*, pages 1431–1442. IEEE, 2005.
14. S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. *SIGOPS Operating System Review*, 41, 2007.
15. V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. k-Anonymity. *Secure Data Management in Decentralized Systems*, 33, 2007.
16. R. Corin and P.-M. Deniélou. A protocol compiler for secure sessions in ml. In *TGC'07*, volume 4912 of *LNCS*, 2007.
17. R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *CSF'07*, 2007.
18. D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5), 1976.
19. D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8), 1981.
20. A. Fiat and M. Naor. Broadcast encryption. In *CRYPTO'93*. Springer, 1994.
21. D. Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, 2003.
22. C. Fournet, G. L. Guernic, and T. Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *CCS'09*. ACM, 2009.
23. G2C website, `http://www.infsec.cs.uni-saarland.de/~reischuk/g2c/`, 2011.

24. Gecode: generic constraint development environment, `http://www.gecode.org`, 2011.
25. M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
26. J. Herranz. Identity-based ring signatures from RSA. *Theoretical Computer Science*, 389, 2007.
27. L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. AURA: A Programming Language for Authorization and Audit. In *ICFP'08*. ACM, 2008.
28. J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP'09*. ACM, 2009.
29. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD'06*. ACM, 2006.
30. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In A. Herbert and K. P. Birman, editors, *SOSP'05*. ACM, 2005.
31. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96*, volume 1055 of *LNCS*. Springer, 1996.
32. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
33. A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 2000.
34. J. A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In *PADL'09*. Springer, 2009.
35. G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5), 2000.
36. P. Ocenasek and M. Sveda. An approach to automated design of security protocols. *ICMLS'06*, 2006.
37. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*. Springer, 1998.
38. R. L. Rivest, A. Shamir, and Y. Tauman. How to leak a secret. *Communications of the ACM*, 22(22), 2001.
39. A. Sabelfeld and A. C. Myers. Language-Based Information Flow Security. *IEEE Journal on Selected Areas In Communications*, 21(1), 2003.
40. C. Sprenger and D. Basin. Developing security protocols by refinement. In *CCS'10*. ACM, 2010.
41. L. Sweeney. k-anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5), 2002.
42. J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *POPL'08*. ACM, 2008.
43. US Depart. of Health & Human Services. The Health Insurance Portability and Accountability Act of 1996 Privacy Rule. http://www.hhs.gov/ocr/privacy, 2009.
44. J. A. Vaughan. A confidentiality extension to the Aura programming language. In *TLDI'11*. ACM, 2011.
45. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3), 1996.
46. D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *EC 96*, 1996.
47. H. Xue, H. Zhang, and S. Qing. A schema of automated design security protocols. In *CISW'07*, 2007.
48. W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *ICDE'09*. IEEE, 2009.

A Syntax of G2C

		Terms
$\mathcal{P}$	::= $\mathcal{C}$	Principal
$\mathcal{T}$	::= $\mathcal{C}$	Tag
$\mathcal{S}$	::= $\mathcal{C}$ ‘(’ *arglist* ‘)’	Statement
$\mathcal{K}$	::= $\mathcal{S}$ ‘@’ $\mathcal{P}$	Knowledge
arg	::= $\mathcal{C} \mid \mathcal{V} \mid$ ‘$*$’	Argument
arglist	::= [*arglist* ‘,’] *arg*	Argument list
tagvarlist	::= [*tagvarlist* ‘or’] *arg*	Tags/variable list
plist	::= [*plist* ‘,’] $\mathcal{P}$	Principal list
statlist	::= [*statlist* ‘,’] $\mathcal{S}$	Statement list
knowlist	::= [*knowlist* ‘or’] $\mathcal{K}$	Knowledge list
		Expressions
princ	::= [*princ*] $\mathcal{P}$ ‘:’ $\mathcal{T}$	Principals
stats	::= [*conf*] $\mathcal{S}$ ‘:’ *tagvarlist*	Statements
tags	::= [*tags*] $\mathcal{T}$ ‘<’ $\mathcal{T}$	Tags
	$\mid$ [*tags* ‘<’] $\mathcal{T}$ ‘<’ $\mathcal{T}$	
	$\mid$ [*tags* ‘<’] $\mathcal{T}$ ‘<’ $\mathcal{T}$ ‘<’ $\mathcal{T}$	
rules	::= [*rules*] $\mathcal{S}$ ‘:-’ $\mathcal{C}$ ‘[’ *statlist* ‘]’	Rules
input	::= [*input*] $\mathcal{K}$	Input
goals	::= [*goals*] $\mathcal{K}$	Goals
		Specification
spec	::= ‘Principals:’ *princ*	Specification
	‘Input:’ *input*	
	‘Rules:’ *rules*	
	‘Goals:’ *goals*	
	‘Tags:’ *tags*	
	‘Statements:’ *stats*	

Figure 3: Formal grammar for the specification language G2C.

A formal grammar for the syntax of G2C is shown in Figure 3. Optional parts are enclosed in square brackets [$\cdot$], literal character sequences are enclosed in single quotes ‘$\cdot$’, choice is denoted by $\cdot|\cdot$. We assume $\mathcal{C}$ to be constants (strings consisting of $\{a,\ldots,z,0,\ldots,9\}$) and $\mathcal{V}$ to be variables (strings consisting of $\{a,\ldots,z\}$, beginning with a capital letter).

We call a specification *syntactically well-formed* iff all of the following hold.

1. The argument lists of rules do not contain wildcards.
2. Rules are safe: the argument lists of head statements of rules do not contain unbound variables, i.e., all variables are bound in the body statements.
3. Rules do not introduce constants, i.e., all constants occurring in the head statement must also occur in the body statements.
4. The argument lists of input statements contain only constants and wildcards.
5. The argument lists of goal statements contain only constants.
6. The arity n of a statement $s(s_1,\ldots,s_n)\in\mathcal{S}$ is the same for all occurrences of s.
7. All principals $P\in\mathcal{P}$ are declared in the section *Principals*.

We call a specification *consistent* if the following holds for all $P\in\mathcal{P}$, $S, S_i\in\mathcal{S}$:

1. $\mathrm{input}(S@P) \Rightarrow \mathrm{may\_access}(P,S)$
2. $\mathrm{goal}(S@P) \Rightarrow \mathrm{may\_access}(P,S)$
3. $(S \leftarrow f[\mathrm{S}_1,\ldots,\mathrm{S}_n]) \in \mathcal{R} \Rightarrow \forall i\neq j : S_i \neq S_j$
4. $((S \leftarrow f[\mathrm{S}_1,\ldots,\mathrm{S}_n]) \in \mathcal{R}\ \wedge \forall i : \mathrm{may\_access}(P,S_i))\ \Rightarrow\ \mathrm{may\_access}(P,S)$

The intuition behind the last requirement is that whenever there exists a rule $r\in\mathcal{R}$ for which P knows all necessary arguments, then P can compute the function value of r and hence the specification should permit the access to the computed value.

B Proof for Theorem 1

Proof. *MsgCplx* $\in \mathcal{NP}$ since any $E \subseteq \mathcal{E}$ with $\sum_{e \in E} c(e) \leq c^*$ serves as poly-length witness, verifiable by a poly-time bounded Turing machine.
The remainder of the proof is a poly-time Karp reduction from 3-SAT. Let a 3-SAT formula F in conjunctive normal form (CNF) with r variables and m clauses be given:

$$F = \bigwedge_{i \in \{1,\ldots,m\}} K_i \quad \text{where } K_i = x_{i_1}^{\alpha_{i_1}} \vee x_{i_2}^{\alpha_{i_2}} \vee x_{i_3}^{\alpha_{i_3}}$$

The superscript $\alpha_{i_j} \in \{+,-\}$ denotes whether variable x_{i_j} occurs positive or negated.

Construct the graph $\mathcal{G}_F$ as follows (see Figure 4 for illustration). For each variable $x_i \in \{x_1,\ldots,x_r\}$ introduce two nodes $x_i, \overline{x_i} \in \mathcal{N}_k$ and a node $v_i \in \mathcal{N}_k$ with edges $(x_i, v_i), (\overline{x_i}, v_i) \in \mathcal{E}$. Introduce another node $v \in \mathcal{N}_c$ that has an incoming edge from every node v_i. For each clause K_i introduce a node $k_i \in \mathcal{N}_k$ with corresponding nodes $x_{i_j} \in \mathcal{N}_k$ for the literals of K_i. Add the edges (x_{i_j}, k_i) for $i \in \{1,\ldots,m\}$ and $j \in \{1,2,3\}$. Finally, create a node $c \in \mathcal{N}_c$ that has incoming edges from v and all k_i. According to F, whenever a variable x_i appears in a clause K_j, an edge from either x_i or $\overline{x_i}$ is drawn to the corresponding literal of K_j (see the two dashed edges as examples). Note that there is exactly one incoming edge for each literal.

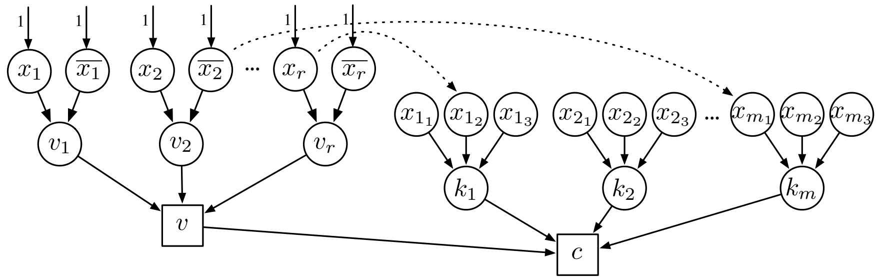


Figure 4: Graph $\mathcal{G}_F$ constructed from 3-SAT instance.

All edges ending in variable nodes $x_1, \overline{x_1}, x_2, \ldots, \overline{x_r}$ have a cost of 1, all other edges have cost 0.

In order to satisfy node v, for each variable x_i either the positive or the negative variant has to be selected. This gives a cost of exactly r. If the formula F is satisfiable, then any satisfying assignment will have the same cost r, since any variable x_i has a fixed assignment to either true or false. On the other hand, if F is not satisfiable, then there is at least one clause K_i for which a dotted edge is missing. This edge would satisfy the clause, incrementing the total cost to at least $r+1$.

Finally, we have that F is satisfiable iff $\mathcal{G}_F$ has cost r.