A user interface for a game-based protocol verification tool[⋆]

Peeter Laud[1] and Ilja Tšahhirov[2]

[1] Cybernetica AS and Tartu University
peeter@cyber.ee
[2] Institute of Cybernetics at Tallinn University of Technology
itshahhirov@gmail.com

Abstract. We present a platform that allows a protocol researcher to specify the sequence of games from an initial protocol to a protocol where the security property under consideration can be shown to hold using "conventional" means. Our tool represents the protocol in the form of a program dependency graph. A step in the sequence corresponds to replacing a local fragment in the current graph. The researcher interacts with the tool by pointing out the location of this fragment and choosing the applied transformation from a list. The tool guarantees the error-freeness of the sequence. By our knowledge, this is the first time where the aspects of user interaction have been seriously considered for a sequence-of-games-based protocol analyzer.

1 Introduction

The sequence-of-games-based approach is a method for giving security proofs for cryptographic protocols that is at the same time computationally sound and sufficiently organized for keeping track of all the details about the probabilities and conditional probabilities of various events. It is based on the fact that most cryptographic primitives have their security definitions stated as two experiments (or cryptographic games) that an adversary can interact with. A primitive is secure if the adversary cannot tell those two experiments apart. In this approach, the security proof of a cryptographic protocol (or a primitive) consists of two steps (which may take place simultaneously). The first step is the construction of a sequence of cryptographic games, the first of which is the original protocol and the last is a game that obviously fulfills the security property we want to prove (e.g. if the goal is the confidentiality of some value, the final game should contain no references to that value). The second step is the verification that to a resource-bounded adversary, each protocol in that sequence is indistinguishable from the one that immediately precedes it. To make such verification easy, the neighboring

[⋆] This research has been supported by Estonian Science Foundation, grant #6944, by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and by EU Integrated Project AEOLUS (contract no. IST-15964).

protocols in that sequence should syntactically differ only a little. For example, a protocol in that sequence may have been obtained from the previous one by locating one of the experiments from the definition of a cryptographic primitive in the code of this protocol, and replacing that part of the code with the code of the other experiment. Alternatively, the change from one protocol to the next could be a simple program transformation/optimization (e.g. copy propagation), done in order to make locating one of the aforementioned experiments easier.

A protocol researcher needs tool support for both steps of the proof done in the style of sequences of games. As a protocol in the sequence is constructed by applying a rather small change to the previous protocol, it makes sense to constrain the researcher in constructing the next protocol, thereby avoiding transcription errors. The verification of the proof (the second step) is also better left to an automated theorem prover.

The most recent results in this area mostly tackle the second problem — verifying the given sequence of games. Languages for cryptographic games have been proposed and certain program transformations have been proven (using proof assistants, such as Coq or Isabelle/HOL) to keep the games indistinguishable to an adversary [5, 9]. In contrast, we consider the first problem in this paper. We present a tool that helps a protocol researcher to *interactively* construct that sequence. So far, similar tools (Blanchet's CryptoVerif [12, 13] and the analyzer of Tšahhirov and Laud [37]) have worked almost fully automatically. An automatic generation of a game sequence is convenient, but not necessarily scalable. There are no guarantees that the set of transformations that the analyzer applies is convergent. Hence the analyzer may get stuck in a game that is not yet obviously secure but also cannot be transformed any longer, while a different order of transformations could have lead to a complete sequence. One may try to come up with heuristics for choosing the order of transformations, but this approach is certainly not complete and may not be worth the effort. Instead, one should rely on the knowledge of the protocol designer — he/she should have some idea why the protocol is secure, and be able to guide the analyzer.

Our tool is an extension of our protocol analyzer [37, 36]. The protocol is presented to the protocol researcher in a form (a dependency graph) that we believe is relatively easy comprehend and where, importantly, the location where one wishes to apply a certain transformation can be easily indicated. The researcher starts with the initial protocol representation (translated from a language similar to applied π-calculus) and applies one transformation after another until the protocol is easy to analyze. The tool makes sure that the researcher will not make invalid transformations. The form in which the protocols are represented has a well-defined semantics, hence it should not be too difficult to combine our tool with some of the verifiers of game sequences we have mentioned above to create a complete tool-chain for producing computationally sound proofs of protocols.

2 Related Work

The task of tool-supported computationally sound proving of security properties of cryptographic properties has received closer attention for almost a decade now. Starting from Abadi and Rogaway [4], a line of work [3, 29, 16, 22, 15] has attempted to show that the security of a protocol in formal model implies its security in the computational model, thereby leveraging the body of work on protocol analysis in the formal model. In parallel to that, program analyses have been devised that are correct with respect to the computational security definitions of cryptographic primitives [39, 23, 24, 27, 33, 20]. A somewhat similar line of work tries to axiomatize the computational semantics of protocols [28, 17, 18].

A somewhat different "formal model" with full computational justification was offered by Backes et al. [7] in the form of a *universally composable cryptographic library*. Various methods of protocol analysis (in the formal model) have been successfully carried over to this model, including type systems [26, 1], abstract interpretation [6] and theorem-proving [34, 35].

The consideration of the sequence of code transformations as a universally and automatically applicable method for protocol analysis first appeared in [25]. The method was generally popularized by Bellare and Rogaway [11] as the game-based method. It was quickly recognized as allowing automated or computer assisted analysis of protocols [32]. By now, the underlying principles of the method have been formalized, also in proof assistants [14, 30, 5, 9] and automatic analyzers have appeared [12, 37]. Active research is going on in this area.

3 Game-Based Protocol Analysis

A cryptographic *game* is the interaction between the adversary and its environment containing the protocol we want to analyze. A game is specified by describing the operations that the environment performs and values it makes available to, or receives from the adversary. The adversary's goal is to bring the game to a state that is considered as winning for it. For example, the adversary may win a game if it correctly guesses a bit generated by the environment.

To formally argue about a game, and to locate a game (or an experiment) in a larger game, it has to be expressed in a programming language with formal semantics. In the sequence-of-games-based protocol analysis, the initial game is transformed to a final game that is obviously secure. Each transformation step changes the game in a way that makes the adversary's winning probability larger or only negligibly smaller. In the latter case we have assumed that the adversary's running time is constrained to be polynomial; in the following we only consider probabilistic polynomial-time (PPT) adversaries. The obviousness of the security of the final game just means that it is easy to analyze and bound the adversary's probability of winning by using some conventional means. For example, if the adversary's goal is to guess a randomly generated bit, and the final game makes no references to that bit, then the adversary's winning probability is definitely no more than 1/2.

4 Protocol representation

We use *dependency graphs* as our intermediate representation of protocols [37, 36]. It has advantages with respect to abstract syntax trees / control flow graphs (used by CryptoVerif) in naturally allowing certain transformations one would like to invoke after applying a cryptographic transformation. Also, the dependency graph emphasizes the producers and consumers of different data items and henceforth appears to be a natural way to specify cryptographic games (despite the tendency to use imperative languages for that purpose in cryptographic literature).

The dependency graph is a directed graph, where each node corresponds to a computation, producing a *value* (either a bit-string or a Boolean). The edges of the graph indicate which nodes use values produced at another nodes. A computation happening at a node could be the execution of a cryptographic algorithm, an arithmetic or a boolean operation. The values produced are either bit strings or boolean values. The values produced outside of the graph (for example, random coin tosses, incoming messages, secret payloads) are brought into it via special nodes, having no incoming edges. Additionally, certain nodes (modeling the sending of messages) explicitly make their input values available to the adversary.

Program dependency graphs have originated as a program analysis and optimization tool [19], systematically recording the computational relationships between different parts of a program. Since then, several flavors of dependency graphs have been proposed, some of them admitting a formal semantics [8, 31], thus being suitable as intermediate program representations in a compiler. Programs represented as dependency graphs are amenable to aggressive optimizations as all program transformations we may want to apply are incremental on dependency graphs. The translation from an optimized dependency graph back to a sequence of instructions executable on an actual processor may be tricky as the optimizations may have introduced patterns that are not easily serializable. This is not al issue for us because we do not have to translate the optimized / simplified / analyzed protocol back to a more conventional form.

The formal definition and semantics of dependency graphs (DGs) can be found in [36]. Informally, DG is a directed, possibly infinite graph where each node v contains an operation $\lambda(v)$ and edges carry the values produced by their source node to be used in the computation at the target node. The nodes have *input ports* to distinguish the roles of incoming values. For each port of each node, there is exactly one incoming edge. The "normal" nodes of a DG are functional — same inputs cause it to produce the same output. Special nodes are used for inputs from and outputs to the outside world.

To represent scheduling information, most computational nodes of a DG have a special boolean input — the *control dependency.* A node can execute only if the value of its control dependency is true (initially, the value of all nodes is either $\perp$ (for nodes producing bit-strings) or false). During the execution of the dependency graph, the adversary can set the values of certain Boolean-valued input-nodes labeled Req to true and thereby initiate the execution of (certain parts of) the DG.

The execution of a DG proceeds in alteration with the adversary. First the adversary sets some Req-nodes and/or the values of some Receive-nodes (these nodes bring bit-string inputs to the DG). The setting of these nodes causes certain nodes of the DG to compute their values. If a value reaches some Send-node then such value is reported back to the adversary. The adversary can then again set some Req- and Receive-nodes and the process repeats, until the adversary decides to stop. The adversary then tries to output something related to secret values in the environment, made available to the DG through Secret-nodes.

Two dependency graphs G_1 and G_2 with the same set of input/output nodes (labeled Req, Receive or Send) are *indistinguishable* if for all PPT adversaries $\mathcal{A}$, the output of $\mathcal{A}$ running in parallel with G_1 is indistinguishable from its output if it runs in parallel with G_2. A dependency graph is *polynomial* if at any time the number of its nodes with values different from $\perp$ or false is polynomial in the number of its Receive- and Req-nodes that the adversary has set.

A *game transformation* is given by two *dependency graph fragments* (DGF). A DGF is basically a DG without the input/output nodes of a regular DG, but having some input/output nodes of its own (in principle: edges with one end inside and the other end outside of the DGF), for both Booleans and bit-strings. A DGF can be executed by the adversary, similarly to a regular DG. Again, the adversary can (iteratively) set the inputs to the DGF and learn the outputs. The indistinguishability and polynomiality for DGF-s is defined in the same way as for DG-s.

Definition. An *occurrence* of a DGF H in a DG G is a mapping φ from the input and internal nodes of H to the nodes of G, such that

- if v and w are input or internal nodes of H, then there is an edge from v to the port π of w iff there is an edge from $\varphi(v)$ to the port π of $\varphi(w)$;
- if there is an edge from $\varphi(v)$ to some node u in G, such that u is not the image of some internal node of H under φ, then there must be an edge from v to an output node in H.

If H and H' have the same inputs and outputs, and φ is an occurrence of H in G then we can replace this occurrence by H' by removing from G all nodes $\varphi(v)$, where v is an internal node of H, and introducing the internal nodes and edges of H' in their stead.

Theorem. If polynomial DGFs H and H' are indistinguishable, and DG G' is obtained from polynomial G by replacing an occurrence of H with H', then G and G' are indistinguishable and G' is polynomial, too [36].

Each node of the DG corresponds to a single operation that the system may perform. To model that some role of some protocol may be executed up to n times, we have to analyze a DG containing n copies of that role. To model that some role of some protocol may be executed an unbounded number of times, or that a party can take part in unbounded number of protocol sessions, requires *infinite* dependency graphs. In infinite dependency graphs, the set of nodes is countably infinite. Also, certain nodes (conjunction and disjunction) may have a countable number of predecessors. A dependency graph fragment can similarly be infinite.

As the infiniteness of a dependency graph is typically caused from the infinite repeating of certain finite constructs, the graph is regular enough to be finitely *represented*. Details can be found in [36]. Here we mention only that we are actually working with *dependency graph representations* (DGR-s) where each node may represent either a single, or countably many (identified with the elements of $\mathbb{N}^X$ for a certain finite set X, recorded in the DGR node) nodes in the actual DG. Similarly, DGFs generalize to DGFRs — dependency graph fragment representations.

On the choice of protocol representation We believe that the representation based on dependency graphs will be more convenient to use than the one based on abstract syntax trees as used by CryptoVerif. There are several reasons for that. First, the enabling conditions for transformations are more often locally represented in dependency graphs. Hence they should be easier to notice by the protocol researcher (but importantly, the visualizer also has to make it easy to locate interesting vertices and to explore their neighborhoods). Second, pointing at the to-be-transformed part of the protocol is very simple using a graph representation, while it may require doing a complex selection in a textual representation. Third and most importantly, all information is easily available in a dependency graph of the protocol, possibly annotated with nodes carrying the results of its static analysis. CryptoVerif contains not just the language for representing protocols, but also a language for *true facts* and *rewrite rules* it has collected for a protocol [13, App. C.2–C.5]. The user cannot control which facts are derived. In an interactive tool, these facts might be added to the textual representation of the program as annotations, but there does not necessarily exist an obvious location in the text for them. Fourth, the graphical representation allows certain natural transformations for which there is no equivalent in the syntax-tree-based representation.

5 The Tool

Our tool takes as an input a protocol specified in a language remniscient to the applied π-calculus [2], translates it into a dependency graph representation, presents it on the screen and allows the researcher to pick a particular transformation and the occurrence of the first DGFR. This occurrence is validated and then replaced by the second DGFR specified by the transformation, the result is again displayed and the researcher can choose the next transformation to apply. There is no obvious end-point to the analysis; at some moment the researcher can decide that the transformed protocol is now obviously secure.

We use the graph visualizer *uDraw(Graph)* [21, 38] as the front-end of our tool. It receives the commands to change the displayed graph from our tool, and sends back the actions of the user — the selected nodes and edges, as well as names of the chosen transformations. The visualizer allows the user to explore the graph and to change its layout. Fig. 1 shows a screenshot of the visualizer after loading a protocol that has just been translated from the π-calculus-like language to a DGR. We see that the translation procedure itself is straightforward and does not attempt to optimize the DGR. In the visualizer, the first row of a node

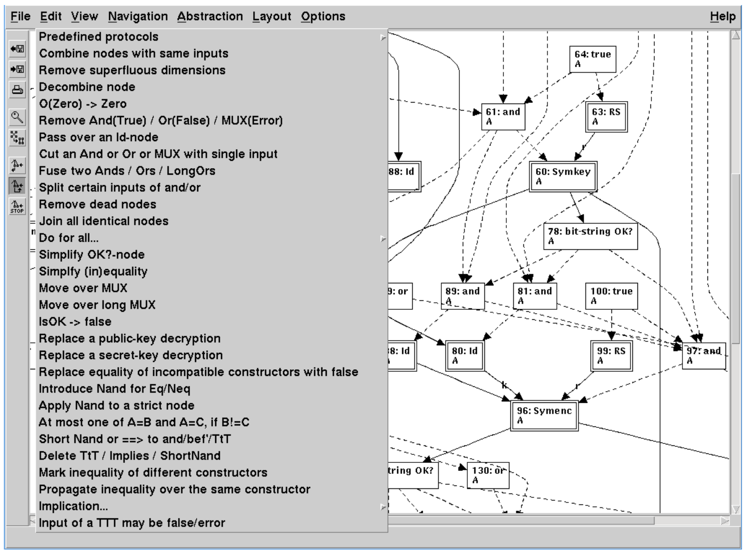
File Edit View Navigation Abstraction Layout Options Help
Predefined protocols
Combine nodes with same inputs
Remove superfluous dimensions
Decombine node
O(Zero) -> Zero
Remove And(True) / Or(False) / MUX(Error)
Pass over an Id-node
Cut an And or Or or MUX with single input
Fuse two Ands / Ors / LongOrs
Split certain inputs of and/or
Remove dead nodes
Join all identical nodes
Do for all...
Simplify OK?-node
Simplify (in)equality
Move over MUX
Move over long MUX
IsOK -> false
Replace a public-key decryption
Replace a secret-key decryption
Replace equality of incompatible constructors with false
Introduce Nand for Eq/Neq
Apply Nand to a strict node
At most one of A=B and A=C, if B!=C
Short Nand or ==> to and/bef'/TtT
Delete TtT / Implies / ShortNand
Mark inequality of different constructors
Propagate inequality over the same constructor
Implication...
Input of a TTT may be false/error
64: true A
61: and A
63: RS A
88: Id
60: Symkey A
78: bit-string OK? A
89: and A
81: and A
100: true A
80: Id A
99: RS A
97: and A
96: Symenc A
130: or A

Fig. 1. A screenshot of the visualizer with a loaded DGR

shows its ID and label, and the second row shows the elements of the (multi)set X of its *replication dimensions*. This node of the DGR corresponds to $\mathbb{N}^X$ nodes in the actual DG.

The tool has been implemented in OCaml. The components of the tool are its main loop (driving the interaction), the graph transformer and the various transformations. A transformation is specified as an OCaml module describing the initial and the final DGFR. Additionally, it contains a method for helping the user to choose the occurrence of the initial DGFR in the DGR. Instead of selecting all nodes and edges comprising a DGFR, as well as specifying the embedding of the DGFR in the DGR, the user has to select only a couple of fixed nodes/edges of the DGFR and this method will reconstruct the whole DGFR. The graph transformer is an OCaml functor receiving a graph transformation as an input and returning a module containing a function that takes as arguments a DGR and the names of the selected nodes / edges, and returns the transformed DGR (or an error message). The main loop receives the node and edge selection commands from the visualizer, as well as the name of the transformation menu element that the user has selected. It calls the correct graph transformation function, finds the difference between the original and the modified DGR, and sends that difference back to the visualizer.

5.1 Specifying a Family of Pairs of DGFRs

It makes sense to parameterize certain parts of DGFRs. For example, to express that tupling followed by projection just selects one of the inputs (*modulo* control dependencies and $\perp$-s): $\pi_i^n((x_1, \ldots, x_n)) \to x_i$, then there should not be a separate transformation for each n and i, but those should be the parameters of the transformation.

The DGFR pairs are specified as OCaml modules with a certain signature. In effect, our approach can be described as a *shallow embedding* of DGFRs into OCaml. A module conforming to that signature has to first define **an OCaml data type for variable names**. The variable names are used as a part in the type for variables; the variables map to the nodes and edges of DGFRs, but also to other values. As next, the module has to declare a **mapping from variable names to their types**. There are 5+1 possible types for a variable or a variable name — it can denote either an integer, a node, an edge, a set of dimensions, a map of dimensions (attached to edges going from one summary node in the DGR to another one; describing how the edges of DGR must be mapped to edges in the DG), or an array where all elements have the same type. The type "node" has three subtypes — a node can be either an input node, an internal node or an output node.

Given the datatype $\mathbf{X}$ of variable names, the **variables** of type $\mathbf{V}$ are defined either as *scalars* of type $\mathbf{X}$ or *elements* (v, i), where $v \in \mathbf{V}$ and i is a natural number. The module then has to define a number of functions that map the variables to their values, in effect describing a DGFR. The following functions have to be defined

– a map from variables of type "array" to their length;
– maps from variables of type "edge" to their source and target (variables of type "node"), dimension map (variables of type "dimension map") and the input port at the target node;
– maps from the variables of types "integer", "dimension", and "dimension map", giving their actual values;
– maps from variables of type "node" giving their label, and their dimension (and also the input dimension if it the node is a contracting node).

Importantly, all those functions can call each other if necessary. Circular dependencies will be detected.

The module has to specify two lists of variable names. The elements of these lists correspond to the nodes and edges in the initial and final DGFR, respectively. During the transformation, the nodes and edges only in the first list will be removed and those only in the second list will be added.

In the code of the functions that the module has to define, it is possible to ask for the actual parameters of the nodes and edges that are the values of the variables with the names in the list defining the initial DGFR. One can ask for the actual labels and dimensions of the variables of type "node", and dimension maps of the variables of type "edge".

The module has to define a **validation function** that tells whether the values of initial variables (variables whose names are in the first list) define a valid DGFR. The function can assume that the nodes and edges are connected in the way given by the functions we described before, but it still has to verify that the nodes have the correct labels. If the transformation depends on it, this function also has to verify that the dimensions and dimension maps of nodes and edges are suitable.

We see that our definition of DGFRs abstracts away from the actual DGR. Indeed, both the validation of the initial DGFR and the construction of the final DGFR are made in terms of DGFR variables. Only the **expansion function** that the module also has to define has access to the actual DGR. The task of this function is to assign values to the variables whose names are in the list of variable names for the initial DGFR. For variables of type "array", it also has to define the length of the array. The inputs to this function are the DGR, and the identities of certain nodes and edges of the DGR, which the expansion function will treat as the values of certain fixed DGFR variables.

5.2 The Graph Transformer

A graph transformer will be defined for each of the transformations specified as the pair of two DGFRs, but the definition is through an OCaml functor. Given the transformation module *TrM*, the transformer function will take a DGR and the node identities as arguments. First, it passes the arguments to the expansion function of *TrM* and receives a mapping φ from initial variables to nodes and edges of the graph. Second, it verifies that the φ indeed constitutes an occurrence of a DGFR in the given DGR — the internal nodes must have all their predecessor and successor nodes also as elements of the DGFR. Third, it invokes the validation function of *TrM* on the received DGFR. Fourth, it performs the actual change of the DGFR — it deletes the nodes and edges that occur in the initial DGFR, but not in the final. Then it adds new nodes and edges (corresponding to the variable names that occur in the list for the final DGFR, but not in the list for the initial DGFR). It calls the functions of *TrM* to find the parameters of those nodes and edges.

The graph transformer also makes sure that the values computed by the functions of *TrM* are memoized and that the computation of a certain function on a certain variable does not (possibly indirectly) invoke the same function on the same variable again. Additionally, the graph transformer verifies the outputs of the functions of *TrM*. For example, if the result of the function returning the dimension of a variable of type "node" is not a variable of type "dimension" then the transformation is immediately halted. Hence the typing of variables is enforced, albeit dynamically.

5.3 Example Analysis

Let us consider a situation where A and B have a long-term shared key K_{AB}, but whenever B wants to send a secret M to A, first A generates a short-term key k, sends it encrypted under K_{AB} to B who then uses k to protect M. In the conventional "arrow-notation" this protocol can be specified as follows:

$$\begin{array}{l} A \longrightarrow B : \{k\}_{K_{AB}} \\ B \longrightarrow A : \{M\}_k \\ A \longrightarrow \_ : \mathrm{OK} \end{array} \tag{1}$$

The initial protocol (game), directly corresponding to (1), but simplified from the output of the translator is depicted in Fig. 2. Here solid edges carry bit-strings and dashed edges booleans. The node 43 generates the key K_{AB}. The nodes 60–179 represent a session of A; those nodes have the (multi)set of dimensions $\{\mathsf{A}\}$, i.e. we are modeling an unbounded number of sessions. The first message is constructed by nodes 60 and 96 (the random coins for these operations are provided by special RS-nodes 63 and 99), and sent away by the node 118; the adversary can request it to be sent by setting (an instance of) the node 119 to true. The second message is received in node 136, decrypted in 142, and if it decrypts successfully (node 154) then the third message is sent in nodes 171 and 178. Similarly, nodes 196–245 represent a session of B — receiving the first and constructing and sending the second message.

As the node 43 is only used for encryption and decryption, we can apply a transformation corresponding to the IND-CCA- and INT-CTXT-security of symmetric encryption [10] to it. We select node 43 and choose "Replace a secret-key decryption" from the menu. The resulting graph is depicted in Fig. 3.

The transformation introduced the nodes 674–680. At first, it replaced the encryption node 96 with the node 676 labeled SymencZ. This operation encrypts a fixed bit-string ZERO using the random coins and the key that are given to it. The string ZERO cannot be the output of any node in the DG. The SymencZ-node does not use the plaintext argument (60) of the original node. Still, it should not produce output if the original plaintext has not been computed. Hence the test (node 674) whether it has been computed is part of the control dependency of node 676. At the decryption side, the ciphertext 196 is compared to all computed encryptions of ZERO in node 677 (note its set of dimensions). If one of them matches then it the corresponding plaintext (node 60) is selected as the result of decryption by nodes ("multiplexers") 679 and 680. The MUX-nodes have an arbitrary (finite) number of inputs, their output is the *least upper bound* of their inputs. I.e. if all inputs are $\perp$ then the output is $\perp$ and if exactly one input is different from $\perp$ then the output is equal to that input. If more than one input is different from $\perp$ (in this transformation, the number of inputs to MUX-nodes is equal to the number of SymEnc-operations) then the result of this operation is $\top$, denoting an inconsistency in the DG. A inconsistency means an immediate termination of the computation; this is visible to the adversary (i.e., if the transformations are correct, then this can happen only with negligible probability). Similarly, an IMUX is a "long MUX" — in a DG it is a node with infinite number of pairs of inputs (a bit-string and a boolean). If no boolean inputs are true, it returns $\perp$. If exactly one of the boolean inputs is true then it returns the corresponding bit-string input. In a DGR, a IMUX is a contracting node — in our example, node 679 contracts the dimension A.

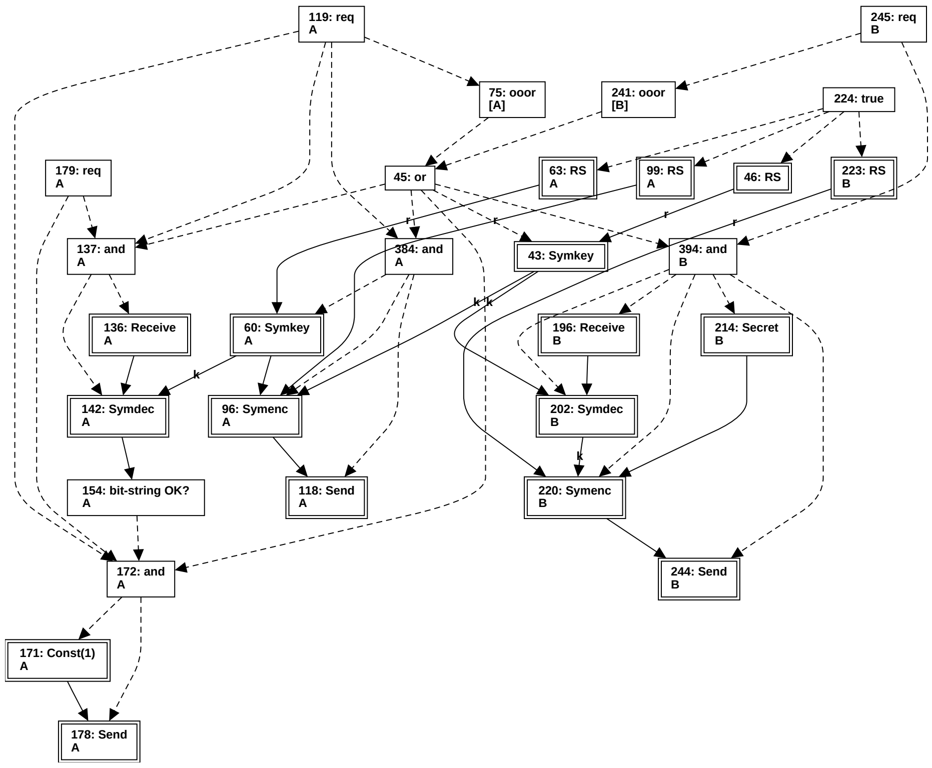

Fig. 2. Initial protocol

We would like to apply the symmetric encryption transformation also to node 60, but it has two uses forbidding that. We get rid of the use by node 674 by noting that SymKey succeeds if there are random coins incoming from RS, and if the control dependency is true. The RS-node 63 always produces coins because its control dependency is true. Hence the value of node 674 equals the value of node 384 and we get rid of this use of node 60.

The other use by node 679 eventually ends up in the SymEnc-node 220. As MUX- and IMUX-nodes do not change the values passing through them, we can swap them with the operations following them. Hence we can move the SymDec-node first to the other side of node 680 and then node 679, resulting in the graph depicted in Fig. 4. We see that the encryption node (with new ID 704) is now right next to the SymKey-node 60. The ability to do such swaps of operations with multiplexers is one of the **main advantages of dependency graphs**.

It is instructive to consider how the second message $\{M\}_k$ (sent by node 244) is computed in this graph. In the b-th round of B, this rounds secret message M_b is encrypted with keys k_a for all rounds a of A. The correct round a is then chosen by nodes 795 and 697 by comparing the first message received by B (node

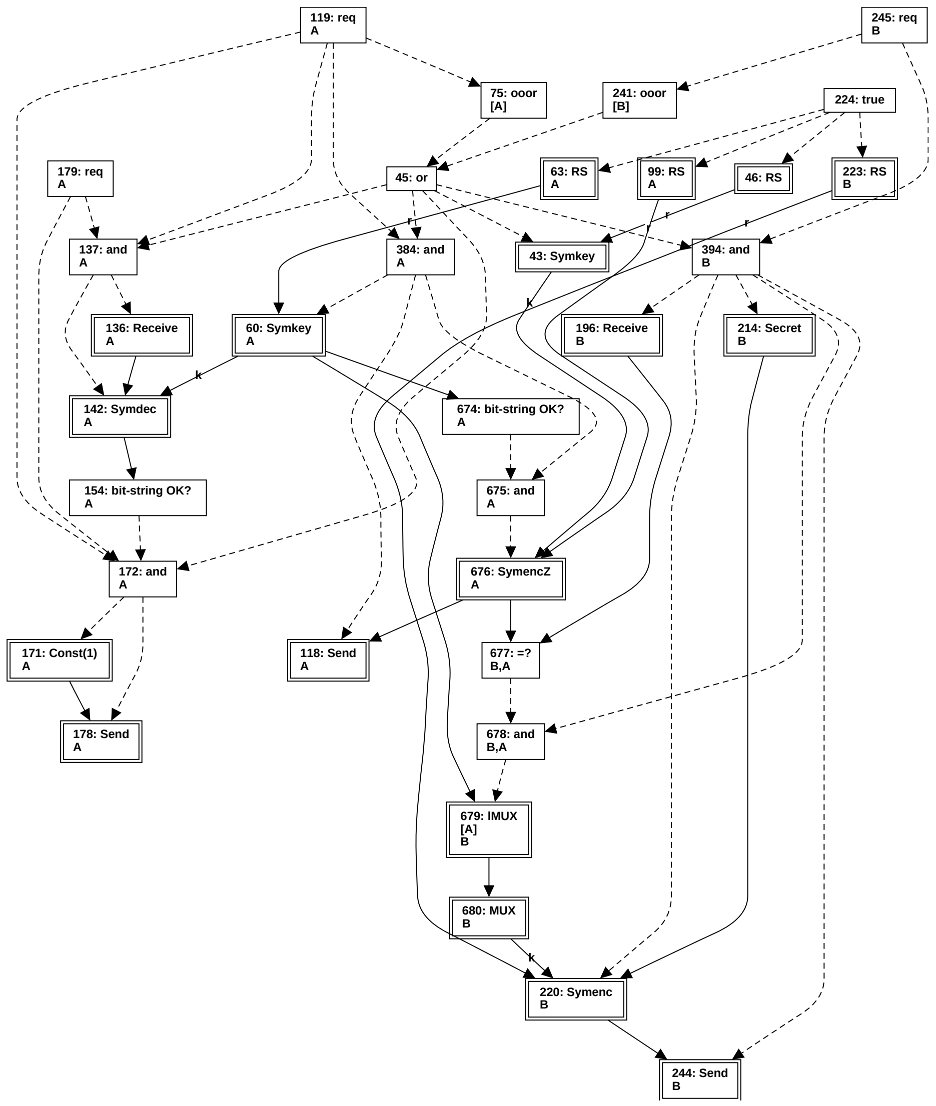

Fig. 3. Applying encryption transformation to K_{AB}

196) in this round with the first messages sent by A (node 118, sending the result of node 676) in all rounds.

The next transformation steps should be obvious. After getting rid of the node 674 (described above) we apply the symmetric encryption transformation to the key generation 60. This will get rid of the use of the Secret-node by node 704. The Secret-node will then be used by the nodes replacing the decryption node 142, as it has to be returned as the plaintext. It will be an input to a multiplexer whose output is only used by node 154. The position of node 154 will be swapped with multiplexers, making the OK?-node an immediate successor of the Secret-node. The use of Secret-node by an OK?-node can be transformed away and the Secret-node had no other uses. We are left with a dead Secret-node that can be removed. Thus the confidentiality of secrets is preserved.

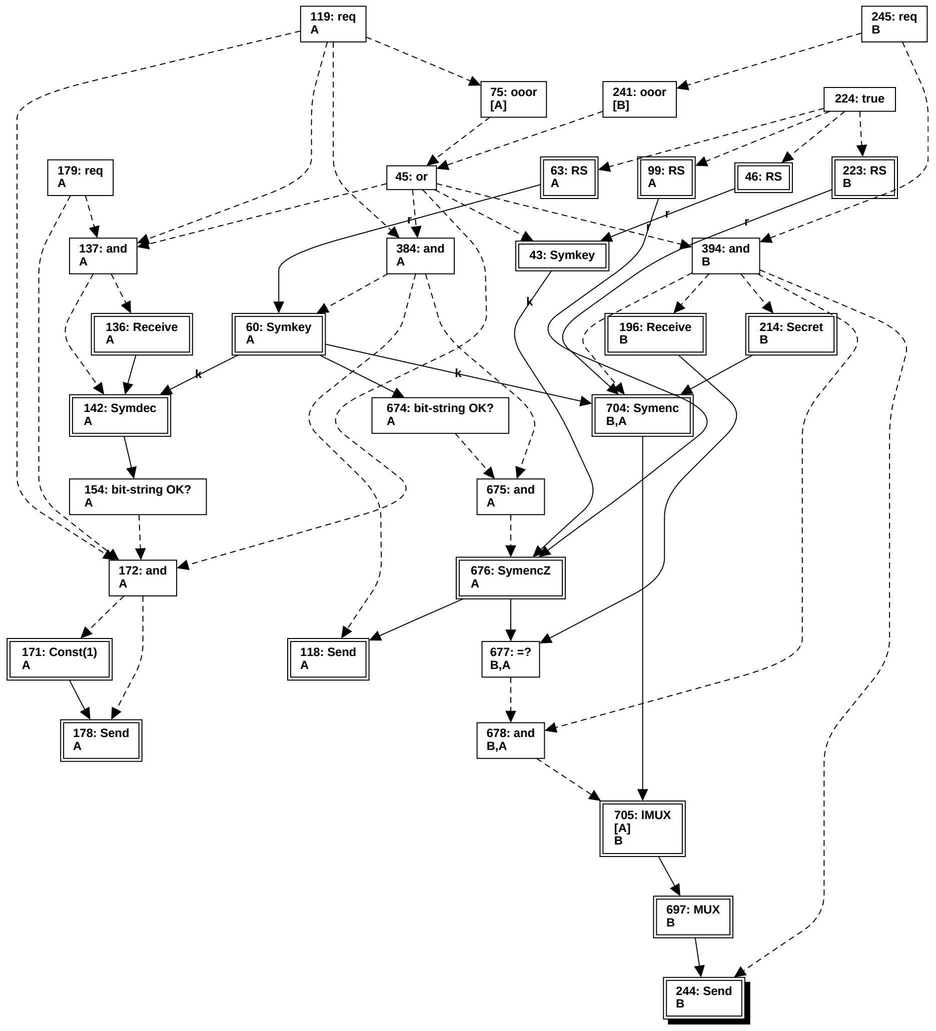

Fig. 4. Moving Symenc over MUX-s

Conclusions The presented example was very simple. More complex examples require the introduction of more and different kinds of nodes, as well as extending the definition of semantics of the dependency graph (but still keeping it mostly functional, i.e. without side effects) [36]. Nevertheless, we have managed to analyze several protocols from the secure protocols open repository (`http://www.lsv.ens-cachan.fr/spore`), including all protocols reported as success in [37]. The work on the analyser is still ongoing, it should be expanded with more cryptographic primitives, as well as different control structures (for the latter, we can rely on previous work on dependency graphs).

References

1. Martín Abadi, Ricardo Corin, and Cédric Fournet. Computational secrecy by typing for the pi calculus. *APLAS 2006*, LNCS 4279, pp. 253–269, 2006.
2. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *POPL 2001*, pp. 104–115, 2001.
3. Martín Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation. *TACS 2001*, LNCS 2215, pp. 82–94, 2001.
4. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 15(2):103–127, 2002.
5. Michael Backes, Matthias Berg, and Dominique Unruh. A formal language for cryptographic pseudocode. *LPAR 2008*, LNCS 5330, pp. 353–376, 2008.
6. Michael Backes and Peeter Laud. Computationally sound secrecy proofs by mechanized flow analysis. *ACM CCS '06*, pp. 370–379, 2006.
7. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. *ACM CCS '03*, pp. 220–230, 2003.
8. Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages. PLDI '90, pp. 257–271, 1990.
9. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *POPL 2009*, pp. 90–101, 2009.
10. Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *ASIACRYPT 2000*, LNCS 1976, pp. 531–535, 2000.
11. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. *EUROCRYPT 2006*, LNCS 4004, pp. 409–426, 2006.
12. Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE S&P 2006*, pp. 140–154, 2006.
13. Bruno Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. Cryptology ePrint Archive, Report 2005/401, Feb. 2nd, 2007.
14. Ricardo Corin and Jerry den Hartog. A probabilistic Hoare-style logic for game-based cryptographic proofs. *ICALP '06*, LNCS 4052, pp. 252–263, 2006.
15. Véronique Cortier, Steve Kremer, Ralf Küsters, and Bogdan Warinschi. Computationally sound symbolic secrecy in the presence of hash functions. *FSTTCS 2006*, LNCS 4337, pp. 176–187, 2006.
16. Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. *ESOP 2005*, LNCS 3444, pp. 157–171, 2005.
17. Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. *ICALP 2005*, LNCS 3580, pp. 16–29, 2005.
18. Anupam Datta, Ante Derek, John C. Mitchell, and Bogdan Warinschi. Computationally sound compositional logic for key exchange protocols. *CSFW 2006*, pp. 321–334, 2006.
19. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
20. Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. *POPL 2008*, pp. 323–335, 2008.

21. Michael Fröhlich and Mattias Werner. Demonstration of the interactive graph-visualization system *a vinci*. *Graph Drawing*, LNCS 894, pp. 266–269, 1994.
22. Romain Janvier, Yassine Lakhnech, and Laurent Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. *ESOP 2005*, LNCS 3444, pp. 172–185, 2005.
23. Peeter Laud. Semantics and program analysis of computationally secure information flow. *ESOP 2001*, LNCS 2028, pp. 77–91, 2001.
24. Peeter Laud. Handling encryption in an analysis for secure information flow. *ESOP 2003*, LNCS 2618, pp. 159–173, 2003.
25. Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. *IEEE S&P 2004*, pp. 71–85, 2004.
26. Peeter Laud. Secrecy types for a simulatable cryptographic library. *ACM CCS 2005*, pp. 26–35, 2005.
27. Peeter Laud and Varmo Vene. A type system for computationally secure information flow. *Fundamentals of Computational Theory '05*, LNCS 3623, pp. 365–377, 2005.
28. Patrick Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. A probabilistic poly-time framework for protocol analysis. *ACM CCS 1998*, pp. 112–121, 1998.
29. Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. *TCC 2004*, LNCS 2951, pp. 133–151, 2004.
30. David Nowak. A framework for game-based security proofs. *ICICS 2007*, LNCS 4861, pp. 319–333, 2007.
31. Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. *POPL 1991*, pp. 67–78, 1991.
32. Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/`.
33. Geoffrey Smith. Secure information flow with random assignment and encryption. *FMSE 2006*, pp. 33–44, 2006.
34. Christoph Sprenger, Michael Backes, David A. Basin, Birgit Pfitzmann, and Michael Waidner. Cryptographically sound theorem proving. *CSFW 2006*, pp. 153–166, 2006.
35. Christoph Sprenger and David A. Basin. Cryptographically-sound protocol-model abstractions. *CSF 2008*, pp. 115–129, 2008.
36. Ilja Tšahhirov. Security Protocols Analysis in the Computational Model — Dependency Flow Graphs-Based Approach. PhD thesis, Tallinn University of Technology, 2008.
37. Ilja Tšahhirov and Peeter Laud. Application of dependency graphs to security protocol analysis. *Trustworthy Global Computing '07*, LNCS 4912, pp. 294–311, 2007.
38. uDraw(Graph) graph visualizer, 2005. `http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html`.
39. Dennis M. Volpano. Secure introduction of one-way functions. *CSFW 2000*, pp. 246–254, 2000.