Application of Dependency Graphs to Security Protocol Analysis

Ilja Tšahhirov[1] and Peeter Laud[2,3]

[1] Tallinn University of Technology
[2] Tartu University
[3] Cybernetica AS

Abstract. We present a computationally sound technique of static analysis for confidentiality in cryptographic protocols. The technique is a combination of the dependency flow graphs presented by Beck and Pingali and our earlier works – we start with the protocol representation as a dependency graph indicating possible flows of data in all possible runs of the protocol and replace the cryptographic operations with constructions which are "obviously secure". Transformations are made in such a way that the semantics of the resulting graph remains computationally indistinguishable from the semantics of the original graph. The transformed graphs are analysed again; the transformations are applied until no more transformations are possible. A protocol is deemed secure if its transformed version is secure; the transformed versions are amenable to a very simple security analysis. The framework is well-suited for producing fully automated (with zero user input) proofs for protocol security.

1 Introduction

A protocol is a convention that enables the connection, communication, and data exchange between several computing entities. A cryptographic protocol is a protocol, which performs some security-related function and applies cryptographic methods. Naturally, the cryptographic protocol is expected to satisfy certain security properties. Examples of those properties could be confidentiality, integrity, and so on. Therefore there is a need for methods for indicating whether the given protocol satisfies the given security property or not. The latter is simpler than the first – having an example of successful attack breaching the property of interest is sufficient to show that the protocol does not satisfy it. Showing that the protocol is secure with respect to certain properties is essentially convincing that no attack (known or unknown) breaching these properties exist.

In this paper we propose a framework for examining the protocol security properties based on the notion of computational indistinguishability, assuming certain properties of cryptographic operations. Currently the framework is used for checking the preservation of confidentiality and integrity properties in protocols. In principle, the framework should be suitable for verifying all properties whose fulfilment can be observed by the protocol participants and/or the adversary — the transformation process does not change the observable properties of protocols.

The basis of the analysis – protocol semantics and the set of assumptions on the cryptographic operations we used – is similar to [18, 27], while the protocol language is much more powerful, and the analysis methodology is significantly different. Our contribution is the introduction of the support for the replication to the analysed protocols, and using the approach based on the extensive usage of the dependency graphs, similar to those introduced in [26], for the analysis of the protocols' security. The developed framework is suitable for producing the automated, computationally sound proofs for the protocols' security.

This paper has the following structure. After reviewing the previous work in this field in Sec. 2, we go on with defining the protocol representation on which the technique is based in Sec. 3. The criterion for considering the protocol corresponding to a given graph secure is defined in Sec. 4, followed by Sec. 5, where more complex types of the dependency graphs, corresponding to the protocols with replication, are considered. Sec. 6 contains key rules for graph transformation preserving the semantics of the graph. In Sec. 7 we explain how the integrity properties can be verified in our framework.

2 Related Work

The research presented in this paper belongs to a body of work attempting to bridge the gap between the two main approaches for modeling and analyzing the cryptographic protocols — the Dolev-Yao model [15] and the complexity-theory-based approach [28]. The most related, and also a source of inspiration to the work reported in this paper has been the protocol analysis framework by Blanchet [9–11]. The difference of our approaches is in the choice of the program (protocol) representation. We believe that the use of dependency graphs instead of a more conventional representation allows our analyser to better focus on the important details of data and control flow. Both frameworks are based on the view of cryptographic proofs as sequences of games [8, 18]. In both frameworks, in order to prove a protocol correct, the automated analyser constructs such a sequence where the adversary's advantage diminishes only negligibly from one game to the next one, and where the adversary has obviously no advantage in the final game. Another difference with [9–11] is the degree of automation – the analyser from [9–11] still requires the human-produced hints on the set and order of transformations to apply for some protocols; the analyser prototype implemented based on our framework does not require any hints.

The work in this area has been started by Abadi and Rogaway [1], who considered the relationship of formal and computational symmetric encryption under passive attacks and provided a procedure to check whether two formal messages have indistinguishable computational semantics. The same primitive and class of attacks have been further considered in [2, 23, 17, 22, 3], in these papers the language has been expanded (the constraints have been weakened) and the security definitions have been clarified. Further on, the active attacks with the range of cryptographic primitives were considered in [13, 16, 21] (based on the translation of protocol traces from the computational to the formal model) and [19, 5] (based on the application of the universally composable [12, 25] cryptographic library [4]).

Another body of research the present work is based on is the static analysis on the intermediate program representations. We use the protocol representation close to [26]. Unlike some of the frameworks based on protocol rewriting – [18, 27], the protocol transformations we perform do not produce several sub-protocols which are to be analyzed separately; the representation chosen is capable of holding all the possible information flows and execution variants, therefore having better potential for analyzing the replication and data flows between different protocol runs.

3 Dependency Graphs

The analysis is performed over a dependency graph, representing the protocol inputs, outputs, operations performed during the execution of the protocol, and the data flows between them. Dependency graph is one possible representation of a protocol. It is universal – every protocol that can be specified in a WHILE-style language [24] can be represented by a dependency graph as well.

The dependency graph is a directed graph defined by a set of *operations* (vertices) and *dependencies* (edges). The operation is an equation with the left side consisting of a dependency (acting as a short-term name for the result of the operation), and the right side being the application of an operator to zero or more dependencies. Each operation is uniquely identified by its label from the *Lab* (set of operation labels). An operation is a *source* for the dependency on the left side of the equation and a *sink* for dependencies listed on the right side. A dependency has exactly one source but can have many sinks.

An operation, in general, is a deterministic computation from its inputs to the output. Some of the operations (e.g. encryption) use randomness in computation, but this randomness is taken as a data dependency, and the operation itself is still deterministic. The following operations can occur in the dependency graph: Constants ($i \in \mathbb{N}$, *true*, *false*, *error*), asymmetric encryption (*keypair*, *pubkey*, *pubenc*, *pubdec*), symmetric encryption (*symkey*, *symenc*, *symdec*), tupling and projecting ($tuple_m$, $proj_i^m$), conversion of random coins to numeric form (*nonce*), boolean operations (and_m, or_m), condition check (*isok*, *iseq*), and selection of one of the values (*mux*). Most operations' names give the general idea on what they do; additional details on some of the operations are provided throughout the article. Operations and_m, or_m, and $tuple_m$ are actually families of operations; each instance in the family having m arguments (for instance, and_2 is a conjunction of two dependencies). The operations not falling into this model are:

- *rs* — returns random coins, *secret* — returns the (also randomly generated) secret message. Let $Lab_* \subseteq Lab$ be the set of labels of all such operations; let $Lab_\bullet \subseteq Lab$ be the set of labels of all "normal" operations (i.e. those labels that are not included in $Lab_\leftharpoondown$ (introduced below) and Lab_*).
- *send*. The only output operation, making the information available to the adversary, is *send*. It has two dependencies – control dependency and the

value to be sent to the network. If the control dependency is *true*, the adversary learns the value of the other dependency. There can be no outgoing dependencies from *send*-nodes. Let $Lab_{\rightarrow} \subseteq Lab_{\bullet}$ be the labels of all *send*-operations.

– *receive*, *req* - return values set by adversary. *receive* denotes the reception of a message from the network (which is assumed to be under adversarial control). *req*-nodes are flags associated with *send*-nodes; they can be set by the adversary when it wants these *send*-nodes to produce a message. Let $Lab_{\leftarrow} \subseteq Lab$ be the set of labels of all such operations.

According to the way treated by operations, the dependencies (arguments) can be divided into two groups — data and control dependencies. Data dependency provides the operation with the argument required to perform the computation (for example, encryption key for the encryption operation). Control dependency indicates whether the operation is to be performed or not. The control dependency of an operation is *true* if it is necessary and possible to perform it. Necessity follows from the adversary's choice to evaluate the public output (i.e. set *req* for the corresponding *send* operation to *true*), or from necessity to evaluate an operation dependent from the given (all such chains terminate with *req*). Possibility to evaluate an operation follows from the possibility to evaluate each operation the current operation is dependent on. For operations having no data dependencies it is always possible to evaluate them (so the control dependency is just the necessity); for operations having one or more data dependencies, the control dependency is a conjunction of necessity to evaluate the current operation and possibility to evaluate each operation the current operation is dependent on. The *if* ... *then* ... construction introduces additional control dependency from the result of the evaluation of the condition to each operation under the *then* branch.

For example, let $\mathit{pubkey}(\mathit{keypair})$, $\mathit{secret}()$, $\mathit{rs}()$, and $\mathit{pubenc}(\mathit{key}, \mathit{text}, \mathit{random})$ be the operations returning public key, secret message, random coins, and encrypting the text using the key and random input, respectively. Let $\mathit{if}(\mathit{condition}, \mathit{statement})$ be the conditional execution construction. The dependencies for the program fragment
$x := \mathit{pubkey}(\ldots); y := \mathit{secret}(); \mathit{if}(a = b, z := \mathit{rs}(); w := \mathit{pubenc}(x, y, z))$; are illustrated in Fig. 1. The operation label is shown in the right part of the node. Data dependencies are drawn as solid lines, control dependencies – as dashed lines. Control dependency of the *pubenc* and *rs* operations specify that (among other conditions) the condition checked in *if* has to be *true* in order for the operation to be executed. Circles represent the nodes of the dependency graph connected to the fragment in question. In the following we present the dependency graphs or their fragments as listings of nodes of the form $\mathit{Operation}^{l}(l_1 \ldots l_m)$. Here l is the label (identity) of the current node, *Operation* is the operation performed in this node and $l_1, \ldots, l_m$ are the labels of the nodes that are the dependencies of the current node l — that produce the values consumed at the node l.

Note that some information on the execution order, not essential to the analysis, is lost. For instance, based on the data dependencies there's no difference in which order the operations with labels 1 and 2 are computed; in the original program, however, these 3 values are computed in a particular order. The evaluation of the operations on the graph could be made in any possible order, as long as the constraints defined by the control dependencies are met. Due to the properties of the graph evaluation semantics, every such evaluation will stop at the same point and give the same final result.

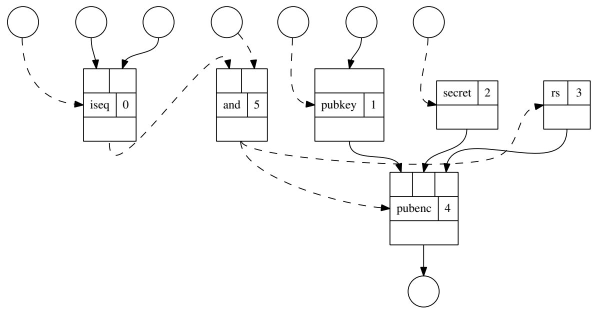

Fig. 1. Dependency graphs

The dependency graphs can be executed. The output of the execution is the set of values assigned to dependencies, given the adversary's input to the graph and the random coins used in the execution. During the execution of the graph the configuration of the graph (mapping of all dependencies to particular values) is modified. The configuration consists of three mappings: $\langle \rho, \phi, \psi \rangle$ returning the values of the "regular" dependencies, dependencies set by the adversary, and the random coins, respectively. For example $\rho[l]$ is the value assigned to the dependency with the source labeled l in a given configuration. The $\psi[secret]$ is the value of the secret message. The value of the dependency can be either a bit-string or $\perp$ (for data dependencies), or either true or false (for control dependencies).

The semantics of a dependency graph with a given initial configuration $C_0 = \langle \rho_0, \phi_0, \psi_0 \rangle$ is the least configuration $\langle \rho, \phi, \psi \rangle$ (order is defined pointwise, with $\perp \leq s$ for any bit-string s and $\mathsf{false} \leq \mathsf{true}$) greater or equal to C_0 that satisfies $\rho[l] = [\![Op]\!](\chi[l_1], \ldots, \chi[l_m])$ for all nodes $Op^l(l_1, \ldots, l_m)$, where χ may refer to either ρ, ϕ or ψ, depending on the type of l_i. The execution of the adversary together with a dependency graph consists of the adversary repeatedly setting certain inputs of the dependency graph (thereby changing its configuration) and learning the inputs to the *send*-nodes after the computation of the semantics of the graph. The details are given in Appendix A.

4 Security Definition

Let the adversary be an algorithm operating in the Probabilistic Polynomial Time (PPT), relative to the security parameter n (of the encryption system used by the cryptographic operations in the graph). The same formal definition of the security parameter as in [18] is used. The running time of the adversary is its total running time for all iterations performed during the execution of the graph.

For a given dependency graph, random coins ψ and the adversary $\mathcal{A}$ let the *adversary's view* $\mathsf{view}_\psi(\mathcal{A})$ be the distribution of $\alpha = \rho|_{Lab_\rightarrow}$ after computing the semantics of the graph with given random coins and adversary inputs. Let D be the distribution of ψ. In order to consider the protocol secure, we require that for all PPT adversaries, the adversary's view is independent from the secret message. Formally, the probability distribution of the pair of the secret message and the adversary's view produced while executing the protocol with this secret message should be indistinguishable from the probability distribution of the pair of the secret message and the adversary's view corresponding to a different secret message, taken according to the same distribution.

$$\begin{aligned}&\{\!|(\psi[secret], \alpha) \,:\, \alpha = Exec(\psi); \psi \leftarrow [\![D]\!]|\!\} \approx \\ &\approx \{\!|(\psi'[secret], \alpha) \,:\, \alpha = Exec(\psi); \psi, \psi' \leftarrow [\![D]\!]|\!\}\end{aligned}$$

The exact meaning of indistinguishability between two *families* of probability distributions $D = \{D_n\}_{n\in\mathbb{N}}$ and $D' = \{D'_n\}_{n\in\mathbb{N}}$, denoted $D \approx D'$, is following: for all PPT algorithms $\mathcal{A}$, the difference of probabilities

$$\mathbf{P}[b = 1 \,|\, x \leftarrow D_n, b \leftarrow \mathcal{A}(1^n, x)] - \mathbf{P}[b = 1 \,|\, x \leftarrow D'_n, b \leftarrow \mathcal{A}(1^n, x)]$$

is a negligible function of n.

Theorem. The protocol which does not include the operation $secret^l$ (the only operation which returns the value of the $\psi[secret]$), is secure (as the generated value of the secret message is not used in it).

This theorem is the "very simple security analysis" mentioned in the abstract.

5 Dependency Graphs with Replication

Normally during the execution of the protocol some parts of it are executed repeatedly (for instance, the execution of the participant could take place more than once), and there is a possibility of data exchange between the different executions of the same operations (e.g. replay attacks, when the adversary records the data exchanged during one run of the protocol, and uses this data to produce values sent to the participants during the next run). To model multiple runs of the protocol, the dependency graph must contain a node for each operation in each possible run of the protocol.

For example, let us suppose that the inner part of the conditional statement of the example program from Sec. 3 could be executed at most two times, using the same value of the conditional test, secret message and the public key. The dependency graph corresponding to it is illustrated on the Fig. 2.a. As the same conditional statement, public key, and secret message are used in both runs, nodes, corresponding to them, are present in the dependency graph only once. The rest of operations are executed two times, which is represented by two copies of each node.

In general, each operation performed in the replicated part of the protocol is represented by a number (equal to the number of runs) of nodes in the dependency graph. The adversary can establish information exchange between the different runs of the program by using the value of the *send*-node corresponding to one run to calculate the value to set to *req* and *receive*-nodes corresponding to another run.

Now let us consider the case when the number of the protocol runs is not limited. Each operation in the replicated part of the program is still represented by a set of nodes, the cardinality of which is equal to the number of possible protocol runs; hence the dependency graph (built using the same principles as the example with two runs) becomes infinite. Despite the fact that the graph is infinite, the constraints put on the adversary's execution time mean that only a finite subset of the nodes of the graph are evaluated.

The structure of the (infinite) dependency graph with the replications is regular enough to be finitely represented. Fig. 2.b contains the representation of the dependency graph from Fig. 2.a., but with an infinite number of replications. Each node in the representation has an additional attribute (the rightmost part of the node), showing whether it is replicated or not – the number of replication dimensions. A replication dimension corresponds to a possibility to execute a part of the protocol countable number of times. In the current example, some nodes in the representation have zero dimensions and the rest have one dimension. A node in the representation having k dimensions corresponds to a set of nodes in the actual dependency graph, the nodes in this set have a natural one-to-one correspondence with the elements of the set $\mathbb{N}^k$. Hence a zero-dimensional node corresponds to a single node in the real graph, and a node with one or more dimensions corresponds to countably many nodes in the real graph.

In this example we only saw nodes with at most one replication dimension. Two- or more-dimensional nodes naturally arise if the protocol specification contains nested replications. We see another case giving rise to more-dimensional nodes in Sec. 6.2.

The representation also specifies the dependencies present in the replicated dependency graph. If the source and the sink of a dependency edge have the same number of dimensions, then this edge in the representation means that each instance of the sink node depends on the the corresponding instance of the source node (but if the number of dimensions is greater than one then the dependency may permute the coordinates). If the dependency sink has more coordinates than the source (it is the case with *pubenc* depending on *secret* in our example) then the value of (a single instance of) dependency source is used in all the instances of the dependency sink. The third possible case – sink having less coordinates than the source – corresponds to each instance of the sink consuming an infinite number of arguments (a single argument from each instance of the source). In our dependency graph representations there are only two operations where this may happen — at logical *or*-s or at multiplexers.

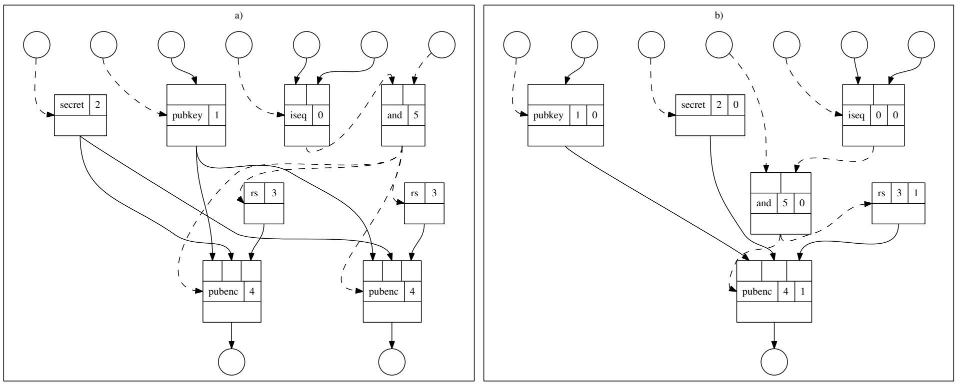

Fig. 2. Dependency graph with replications

6 Transformations

We use graph transformations to replace a potentially insecure graph with another graph, the adversary's view of which is computationally indistinguishable from the first. The meaning of the computational indistinguishability of two dependency graphs is analogous to the one specified in Sec. 4, with the difference that now the task is to distinguish two dependency graph with equal inputs (including secret value).

The transformations are designed in a way that each of them either reduces the complexity of the graph, makes the implicit result of some operations explicit, or eliminates data flows due to the properties to certain cryptographic operations. While performing possible transformations to the graph, it is gradually transformed to a form that makes more explicit which data flows really exist and which are "hidden" by the cryptographic operations. If at the end of the transformation sequence a secure graph is obtained, the very first graph is also secure.

The rest of the section contains the description of some of the transformations preserving the adversary's view on the graph.

6.1 Dead code removal

One special transformation is removal of dead code. Despite being trivial, it is important to consider, as it is the only transformation that makes the graph smaller (other transformations only add additional operations to the graph.)

A node in a dependency graph is *live* if it is a *send*-node or if the value produced by it is consumed by a live node. All nodes that are not live may be removed from the graph.

6.2 The replacement of encryptions

First, let us consider the asymmetric encryption operation. We require the encryption system to satisfy the IND-CCA2 property as defined in [6] – i.e. it should be impossible to distinguish two ciphertexts produced by the encryption oracle for two plaintexts given by the adversary with non-negligible probability in polynomial time even if the adversary is given the ability to decrypt everything except the challenge ciphertext. It means that encryption of the plaintext with the given public key could be replaced with the encryption of the string of zeroes (or any other constant) of equal length, and that latter ciphertext will be indistinguishable from the first for anyone, except for the one having the corresponding private key. On the decryption side we first check whether the ciphertext matches one of the ciphertexts already produced, and if it does, the corresponding plaintext is returned. If no match is found, the decryption operation is performed.

In terms of equivalent dependency graph fragments, it can be presented as follows. Suppose the graph contains the following operations.

$$
\begin{gathered}
rs^{l^{vrs}}(l^{crs}) \qquad keypair^{l^{vkp}}(l^{ckp}, l^{vrs}) \qquad pubkey^{l^{vpk}}(l^{cpk}, l^{vkp}) \\
rs^{l_i^{vrs}}(l_i^{crs}) \qquad pubenc^{l_i^{vect}}(l_i^{ce}, l^{vpk}, l_i^{vept}, l_i^{vrs}) \qquad (\text{where } 1 \le i \le n) \\
pubdec^{l_j^{vdpt}}(l_j^{cd}, l^{vkp}, l_j^{vdct}) \qquad (\text{where } 1 \le j \le m)
\end{gathered}
$$

The operations used are: rs is the generation of random coins, *keypair* and *pubkey* are the generation of the secret-public key pair and the extraction of the public key component from the pair, *pubenc* is the encryption of the given plain text using the given public key and random coins. All the operations have the control dependency as their first argument, and are evaluated only if it becomes *true*. The operation name is superscripted with the label of the operation (for instance, l^{vrs}). The abbreviations used in label names are: v stands for value, c for control, *kp* for key pair, *pk* for public key, *rs* for random seed, *e* for encryption, *d* for decryption, *pt* for plain text, and *ct* for cyphertext. The dependency is identified by the label of its source. We require that the dependencies l^{vrs}, l_i^{vrs} and l^{vkp} are not used in any operations except for the ones listed above.

By the definition of IND-CCA2, the semantics of the graph are indistinguishable from the semantics of the graph where all occurrences of the dependencies

l_i^{vect} and l_j^{vdpt} in the graph are replaced with the $l_i'^{vect}$ and $l_j'^{vdpt}$, respectively. These new dependencies are defined as follows:

$$pubencz^{l_i'^{vect}}(l_i^{ce}, l^{vpk}, l_i^{vrs}) \qquad \text{(where } 1 \leq i \leq n\text{)}$$
$$iseq^{l_{ji}^{cvar}}(l_j^{vdct}, l_i^{vect}) \qquad isok^{l_j^{cdok}}(l_j^{vdpt}) \qquad \text{(where } 1 \leq i \leq n,\ 1 \leq j \leq m\text{)}$$
$$mux_{n+1}^{l_j'^{vdpt}}(l_j^{cd}, l_{j1}^{cvar}, l_1^{vept}, \ldots, l_{jn}^{cvar}, l_n^{vept}, l_j^{cdok}, l_j^{vdpt}) \qquad \text{(where } 1 \leq j \leq m\text{)}$$

There are two operations that need to be clarified here: *pubencz* and *mux*. *pubencz* is the encryption of a special constant ("zero") that cannot be produced by any of the nodes in the graph. Decryption of the ciphertext produced by *pubencz* returns $\perp$. mux_n is the multiplexing operation. Beside the control dependency it takes n pairs of control and data dependencies as inputs. If the control dependency in exactly one of those pairs is true, the result of mux_n is the value of the data dependency. If none of the control dependencies are true, the result is $\perp$ (denoting error). If more than one of the control dependencies are true, the result is $\top$ (denoting inconsistency of the dependency graph; in this case the evaluation of the whole graph stops and this can be detected by the adversary). Note that among the above dependencies $l_{j1}^{cvar}, \ldots, l_{jn}^{cvar}, l_j^{cdok}$ there can indeed be at most one that is true — if l_j^{cdok} is true then the ciphertext l_j^{vdct} cannot be created by a *pubencz*-operation and hence all comparisons $iseq(l_j^{vdct}, l_i^{vect})$ return false. If one of these comparisons returns true then the ciphertext l_j^{vdct} was created by a *pubencz*-operation and its decryption l_j^{vdpt} will be false (by definition). Different *pubencz*-operations produce different values (the opposite event has negligible probability), hence at most one of $l_{j1}^{cvar}, \ldots, l_{jn}^{cvar}$ can be true.

If infinite replications are present then we have to consider the dimensions of the operations as well. Namely, the set of dimensions of the *iseq*-operation labeled with l_{ji}^{cvar} is the union of the sets of dimensions of the i-th encryption and the j-th decryption operation. The representations of the dependencies to l_{ji}^{cvar} include coordinate mappings telling which dimensions of l_j^{vdct} and l_i^{vect} correspond to which dimensions of l_{ji}^{cvar}. The multiplexer labeled with $l_j'^{vdpt}$ has just the same dimensions as the j-th decryption operation, hence it has to contract the rest of the dimensions of l_{ji}^{cvar}. In effect, the multiplexer will have an infinite number of inputs. The semantics stays the same — something different from $\perp$ or $\top$ is produced only if exactly one of (infinitely many) selectors is *true*.

Symmetric encryption is handled in a similar way. The requirements put on the symmetric encryption scheme are IND-CCA2, and ciphertext integrity (inability to generate a "valid" ciphertext without knowing the secret key) [7]. Similarly, based on these properties, the encryptions of the real text are replaced with the encryptions of the constant, and the decryptions are replaced with a set of comparisons with encrypted values. This can only be done if the encryption key is not used anywhere except as a key in encryption and decryption operations, and the random coins used in key generation and encryptions are used only there. The difference from the asymmetric decryption is that the decryption of the ciphertext does not have to be considered (due to the ciphertext integrity property). Formally, let the graph contain the following operations:

$$rs^{l^{vrs}}(l^{crs}) \qquad symkey^{l^{vsk}}(l^{csk}, l^{vrs})$$
$$rs^{l_i^{vrs}}(l_i^{crs}) \qquad symenc^{l_i^{vect}}(l_i^{ce}, l^{vsk}, l_i^{vept}, l_i^{vrs}) \qquad \text{(where } 1 \le i \le n\text{)}$$
$$symdec^{l_j^{vdpt}}(l_j^{cd}, l^{vsk}, l_j^{vdct}) \qquad \text{(where } 1 \le j \le m\text{)}$$

All occurrences of l_i^{vect} and l_i^{vdpt} in the rest of the graph are replaced with the $l_i'^{vect}$ and $l_j'^{dpt}$, defined as following (*symencz* is defined in the same way as *pubencz*, but uses the symmetric encryption primitive):

$$symencz^{l_i'^{vect}}(l_i^{ce}, l^{vsk}, l_i^{vrs}) \qquad \text{(where } 1 \le i \le n\text{)}$$
$$iseq^{l_{ji}^{cvar}}(l_j^{vdct}, l_i^{vect}) \qquad \text{(where } 1 \le i \le n,\ 1 \le j \le m\text{)}$$
$$mux_n^{l_j'^{vdpt}}(l^{cd_j}, l_{j1}^{cvar}, l_1^{vept}, \ldots, l_{jn}^{cvar}, l_n^{vept}) \qquad \text{(where } 1 \le j \le m\text{)}$$

6.3 The movement of multiplexers

After replacing the encryption and decryption operations with *mux*, it is usually possible to simplify the resulting graph. As the *mux* does not perform a computation on its input dependencies, an operation performed on the output of the *mux* can be "shifted" to all inputs. This movement may expose further simplification possibilities by putting next to each other the operations that cancel each other out. Formally if the graph contains the operations

$$mux_n^l(l^{cmux}, l_1^{cvar}, l_1^{vvar}, \ldots, l_n^{cvar}, l_n^{vvar})$$
$$Op^{l^2}(\ldots, l, \ldots)$$

then all occurrences of the dependency l^2 in the graph may be replaced with l'^2 after we have added the following operations to the graph:

$$Op^{l_i'^{vvar}}(\ldots, l_i^{vvar}, \ldots) \qquad \text{(where } 1 \le i \le n\text{)}$$
$$mux_n^{l'^2}(l^{cmux}, l_1^{cvar}, l_1'^{vvar}, \ldots, l_n^{cvar}, l_n'^{vvar})$$

If l'^2 is a control dependency (meaning that $l_1'^{vvar}, \ldots, l_n'^{vvar}$ are control dependencies as well) then the resulting multiplexer can be replaced with a composition of boolean operations expressing the semantics of multiplexers.

Another possible simplification is the replacment of a *mux* with always selected option with that option. Suppose the graph has the operation:

$$mux_n^l(l^{cmux}, l_1^{cvar}, l_1^{vvar}, \ldots, l_n^{cvar}, l_n^{vvar})$$

If it can be derived that $\rho[c_{mux}] = \mathit{true}$ implies for some i $\rho[l_i^{cvar}] = \mathit{true}$, $\forall j \ne i.\rho[l_j^{cvar}] = \mathit{false}$, then the only possible outcome of the *mux* is the i-th option (or *error* (always returning $\perp$), if the control dependency is not *true*). So, all the references to l can be replaced with l', defined as following: $id^{l'}(l^{cmux}, l_i^{vvar})$.

6.4 Other transformations

Due to space requirements less principal transformations cannot be described in length – a brief overview on them is given instead.

Some of the transformations follow from the properties of the data operations. Projection of a component from just constructed tuple can be replaced with the dependency used in tuple construction; control dependency of that operation is a conjunction of tuple and the original projection operation control dependencies. The same simplification is also possible for other corresponding constructor-destructor pairs. But if a constructor is followed by an incompatible destructor (for example, projection from a ciphertext) then the destructor may be replaced with the *error*-operation — we assume that all values produced in the graph are tagged with their type, so the type confusion is impossible.

Most of the operations return $\perp$ if the control dependency is *false*, or any of the arguments is $\perp$. If it is statically known that either of these cases is always true, the corresponding operation could be replaced with the *error*.

Another set of transformations is based on the boolean logic: an and_1 or or_1 (operation with a single dependency) can be replaced with its input dependency; an *and* or *or* operation whose result is statically known (one of the dependencies is *true* in case of *or*, and *false* in case of *and*) can be replaced with that result; two sequential boolean operations of the same kind can be combined; if it can be derived that one dependency implies (in the boolean logic sense) another, then *or* and *and* of these two dependencies can be formulated using just one of the dependencies. If it is possible to derive that at most one of two control dependencies can be true at any time (for example, the results $iseq(l_1, l_2)$ and $iseq(l_1, l_3)$ where l_2 and l_3 can be equal only with negligible probability — they might be the results of different *pubencz*-operations) then their conjunction can be replaced with *false*.

As the semantics of the operation depend only on the incoming dependencies of the operation (except for the operations returning random coins or adversarial input), two operations of the same kind with the same dependencies return the same result, so in the graph the result of the second operation can be replaced with the result of the first. If the cardinality of the operation is greater than that of its input dependencies (except for the control dependency), then the same values of the input dependencies are copied to each of the instances of the operation; each instance returns the same result, so the "extra" dimensions could be removed from the operation, and the result can be copied the required number of times.

In some cases it is possible to explicitly reflect the result of the computation in the graph. If two dependencies must always be equal at the input to some operation, they are replaced (on the input to the operation in question) with the special operation *merge*. It either returns the value of the dependencies if they are equal or $\perp$ if not. Two *merge* operations going in sequence can be replaced with a single *merge* operation having the union of the original operations' inputs. The operation *isok* returns *true* if the value of the input dependency is different from $\perp$ and *false* otherwise. If the input to the *isok* is statically known, the result is also constant. Some operations (e.g. random coins generation) never fail. The success of some of the operations performing computations only depends on whether its arguments are valid (so *isok* of the result could be replaced with a conjunction of *isok* of each argument). Similarly, if the arguments to the comparison operation *iseq* are statically known, the results are also known and the operation could be replaced with a constant *true* or *false*.

7 Proving integrity properties

While the secrecy property we defined in Sec. 4 stated the independence of adversary's view from the secret messages, the integrity properties state that only runs satisfying a certain predicate are feasible (i.e. the probablity that a protocol run does not satisfy this predicate is negligible). *Correspondence assertions* are a well-known means for specifying integrity properties of protocols, and our approach turns out to be well-suited for arguing about them as well.

To state a correspondence assertion, one inserts statements of the form $\mathit{begin}(E)$ and $\mathit{end}(E)$ to the protocol text, where E is some expression. The execution of such a statement means finding the value v of E and recording that the statement $\mathit{begin}(v)$ or $\mathit{end}(v)$ has been executed. The correspondence is satisfied if each execution of $\mathit{end}(v)$ is preceeded by an execution of $\mathit{begin}(v)$ with the same value v. Moreover, the correspondence is *injective* if there is a separate $\mathit{begin}(v)$ for each $\mathit{end}(v)$ [20].

Similarly, we can insert nodes $\mathit{begin}^l(l^c, l^v)$ and $\mathit{end}^l(l^c, l^v)$ to our protocol. Here l^c is the control dependency and l^v carries the value that appears in the executed *begin*- or *end*-statement. An execution of the dependency graph also gives rise to a sequence of executed *begin*- and *end*-nodes (if several of those nodes are executed simultaneously then we assume that *begin*-s happened before the *end*-s), allowing us to define correspondence and injective correspondence in the same way. The protocol modifications do no change the order of execution of those statements, hence the original protocol satisfies the correspondence property iff the final protocol does.

Our experience shows that arguing about the order of executions of *begin*- and *end*-nodes in the final, modified protocol is quite feasible. To show correspondence we have to locate a node $\mathit{begin}^{l^b}(l^{cb}, l^{vb})$ for each node $\mathit{end}^{l^e}(l^{ce}, l^{ve})$, such that the values of dependencies l^{vb} and l^{ve} are equal whenever both are defined. If $l^{vb} = l^{ve}$ then the equality is trivial, but the dependency graph also enables us to straightforwardly check their equality if l^{ve} is defined as a *merge* of l^{vb} and something else (or other similar patterns). We also have to show that l^{ce} implies l^{cb}. We have devised an analysis for determining the implications between control dependencies (the results of this analysis are also used in transformations described in Sec. 6.4) and this analysis turns out to be precise enough for determining that $l^{ce} \Rightarrow l^{cb}$. The injective correspondence can be proved by showing that among several control dependencies, controlling several alternative *end*-statements, at most one can be true (the respective analysis of control dependencies was also hinted in Sec. 6.4).

8 Results achieved

The dependency graphs-based approach, presented in this article, is a convenient tool for investigating the data flows taking place in the distributed computing systems. It is based on finding out the data and control flows on the protocol representation, and modifying the representation based on the semantics of the individual operations and connections between them.

The automatic protocol analyser has been implemented. The analyser has been applied to several protocols from the secure protocols open repository (`http://www.lsv.ens-cachan.fr/spore/`): Needham-Schroeder public key, Lowe's fixed version of Needham-Schroeder Public Key, Needham-Schroeder secret key, Kao-Chow Authentication, and TMN. TMN and Needham-Schroeder public key were not proved to be secure (corresponding graphs still contain secret message even after all the transformations applied); for both protocols there are known attacks. Graphs corresponding to Lowe's fixed version of Needham-Schroeder Public Key and Kao-Chow Authentication were transformed to a form not containing the secret message, thus indicating that secrecy property holds. The Needham-Schroeder secret key is considered secure under the condition that previously exchanged key is not compromised; the Denning-Sacco key freshness attack (if the adversary has obtained the key used in one of the previous sessions) was also successfully detected. Analysis of the Needham-Schroeder-Lowe public key protocol takes 10 minutes on Pentium M 1.60 GHz machine. The current implementation of the protocol analyser has some room for optimization (mostly – reusing the intermediate calculations between transformations), so the time of the analysis could be significantly reduced.

The difference with CryptoVerif [9–11] is that the transformations defined in our framework are not as sensitive to the order of applications as the CryptoVerif ones. The CryptoVerif, while being mechanized, is not automatic while operating on some protocols involving public key cryptography (e.g. Needham-Schroeder-Lowe public key). At some point in proof several transformations are "allowed", but only one of them leads to the successful proof. The human support is required to decide which transformation is the proper one. An example of transformations in question is *SArename* and the ones based on the properties of the cryptographic primitives . The *SArename* is performed to split the single variable defined multiple times (e.g. in different branches of *if*) into several separate definitions. Each use of such variable is then analysed in branches - one for each possible definition. Applying the transformation of the public key encryption "too early" in the analysis transforms the protocol to the form, where the *SArename* is no longer able to split the uses of the variable at the same fine granularity. Some of the information flows revealing the secret, present in the cases never executed, being combined with cases that are executed, are not removed from the further analysis, and the proof fails. When performed in the "right" order, same transformations separate the cases at sufficient granularity to produce a successful proof.

The analyser prototype we implemented is able to prove the secrecy of each secure protocol we tried it on by iterating the application of possible transformations until no more transformations are possible, without any hints from the user. So far we haven't observed the situation where the success of the proof depends on the order of transformations applied. We suspect the reason is in the usage of the "smaller-scale" transformations, and making all the conditions present at the point of definition (of each sub-case) explicit at the point of use, thus being able to delimit and remove the branches never executed with satisfactory precision without external hints. Another possible reason is that the multiplexer movement (described in sec. 6.3) is not present in CryptoVerif [9–11], nor seems there to be an easy way to add it. The information collection phase of CryptoVerif is able to compensate this omission to some extent, by propagating information about values through the multiplexer. This information can be used for validating certain simplifications, but not for doing cryptographic transformations.

The further research directions are quite "standard" — the application of the framework for determining other security properties, enriching the programming language with more cryptographic primitives, calculation of the exact (negligible) probability of distinguishing the semantics of the resulting graph from the initial one, and running the framework on more protocols. In particular, it would be interesting to verify protocols that are not so easily expressible in the Dolev-Yao model because of the cryptographic primitives they are using (e.g. verifyable secret sharing) and the security properties we are trying to prove. Voting schemes (for example, [14]) are obvious candidates for testing our framework.

9 Acknowledgement

We are thankful to the anonymous referees of TGC 2007 conference for their comments. They certainly have improved the readability of this paper.

This research has been supported by the Estonian Information Technology Foundation, the ESF Grants No. 6839 and 6944, and the EU FP6-IST project No. 15964 "AEOLUS".

References

1. Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). International Conference IFIP TCS 2000, LNCS 1872, pages 3–22, August 2000.
2. Martín Abadi and Jan Jürjens. Formal Eavesdropping and its Computational Interpretation. Theoretical Aspects of Computer Software, 4th International Symposium (TACS 2001), LNCS 2215, pages 82–94, September 2001.
3. Pedro Adão, Gergei Bana, Jonathan Herzog, and Andre Scedrov. Soundness of formal encryption in the presence of key-cycles. 10-th European Symposium on Research in Computer Security (ESORICS 2005), LNCS 3679, pages 374-396, September 2005.
4. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A Composable Cryptographic Library with Nested Operations. 10th ACM Conference on Computer and Communications Security (CCS 2003), pages 220–230, October 2003.

5. M. Backes and P. Laud. Computationally Sound Secrecy Proofs by Mechanized Flow Analysis. 13th ACM Conference on Computer and Communications Security (CCS 2006), pages 370–379, 2006.
6. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among Notions of Security for Public-Key Encryption Schemes. In proc of. CRYPTO '98 (LNCS 1462), pages 26–45.
7. M. Bellare and C. Namprempre. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. Asiacrypt 2000, LNCS 1976, pages 531–545, 2000.
8. M. Bellare and P. Rogaway. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. Eurocrypt 2006, LNCS 4004, pages 409–426, 2006. (also: Cryptology ePrint Archive, Report 2004/331, `http://eprint.iacr.org`)
9. B. Blanchet. A computationally sound mechanized prover for security protocols. Proc. 27th IEEE Symposium on Security & Privacy, 2006.
10. B. Blanchet and D. Pointcheval. Automated Security Proofs with Sequences of Games. CRYPTO 2006, LNCS 4117, pages 537–554, 2006.
11. B. Blanchet. Computationally Sound Mechanized Proofs of Correspondence Assertions. Proc. 20th IEEE Computer Security Foundations Symposium, 2007.
12. Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001), pages 136–145, October 2001.
13. Veronique Cortier and Bogdan Warinschi. Computationally Sound, Automated Proofs for Security Protocols. 14th European Symposium on Programming (ESOP 2005), LNCS 3444, pages 157–171, April 2005.
14. Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A Secure and Optimally Efficient Multi-Authority Election Scheme. EUROCRYPT'97, LNCS 1233, pages 103–118, 1997.
15. Danny Dolev and Andrew C. Yao. On the security of public key protocols. IEEE Transactions on Information Theory, IT-29(12), pages 198–208, March 1983.
16. Joshua D. Guttman, F. Javier Thayer, and Lenore D. Zuck. The Faithfulness of Abstract Protocol Analysis: Message Authentication. 8th ACM Conference on Computer and Communications Security (CCS 2001), pages 186–195, November 2001.
17. Peeter Laud. Handling Encryption in Analysis for Secure Information Flow. 12th European Symposium on Programming (ESOP 2003), LNCS 2618, pages 159–173, April 2003.
18. Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. 2004 IEEE Symposium on Security and Privacy, pages 71–85, May 2004.
19. Peeter Laud. Secrecy Types for a Simulatable Cryptographic Library. 12th ACM Conference on Computer and Communications Security (CCS 2005), pages 26–35, 2005.
20. Gavin Lowe. A Hierarchy of Authentication Specification. 10th Computer Security Foundations Workshop, pages 31–44, 1997.
21. Romain Janvier, Yassine Lakhnech, and Laurent Mazaré. Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries. 14th European Symposium on Programming (ESOP 2005), LNCS 3444, pages 172–185, April 2005.

22. Daniele Micciancio and Saurabh Panjwani. Adaptive Security of Symbolic Encryption. 2nd Theory of Cryptography Conference (TCC 2005), LNCS 3378, pages 169–187, February 2005.
23. Daniele Micciancio and Bogdan Warinschi. Completeness Theorems for the Abadi-Rogaway Logic of Encrypted Expressions. Workshop in Issues in the Theory of Security (WITS 2002), January 2002.
24. H. Riis Nielson, F. Nielson. Semantics with Applications: A Formal Introduction. Wiley, 1992.
25. B. Pfitzmann, M. Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. 2001 IEEE Symposium on Security and Privacy (IEEE S&P 2001), pages 184–200, 2001.
26. K. Pingali and M. Beck and R. Johnson and M. Moudgill and P. Stodghill. Dependence Flow Graphs: an Algebraic Approach to Program Dependencies. Advances in Languages and Compilers for Parallel Processing, , MIT Press, Cambridge, MA, pages 445–467, 1991.
27. Ilja Tšahhirov and Peeter Laud. Digital Signature in Automatic Analyses for Confidentiality against Active Adversaries. Nordsec 2005, 10th Nordic Workshop on Secure IT Systems, Tartu, Estonia, October 20-21, 2005, pp 29-41.
28. Andrew C. Yao. Theory and Applications of Trapdoor Functions (extended abstract). 23rd Annual Symposium on Foundations of Computer Science, pages 80–91, November 1982.

A Appendix: Dependency Graph Semantics

Let Σ be the set of bit strings the computations are performed on. Let $\Sigma_\perp$ be the $\Sigma \cup \{\perp\}$. Let $\leq_\Sigma$ be the order on the $\Sigma_\perp$, defined as $\forall x \in \Sigma_\perp . \perp \leq_\Sigma x$, and everything else is incomparable. Let $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$, ordered by $\mathit{false} \leq_\mathbb{B} \mathit{true}$. Let Lab be the set of labels.

The configuration is the state of the computation on a graph. It assigns the values to all dependencies (including the adversary's view), adversarial inputs, and random coin tosses. Formally, it has the type:

$$Configuration : ((\mathit{Lab}_\bullet \to \Sigma_\perp \cup \mathbb{B}) \times (\mathit{Lab}_\leftarrow \to \Sigma_\perp \cup \mathbb{B}) \times (\mathit{Lab}_* \to \Sigma))^\top$$

For some $\langle \rho, \phi, \psi \rangle \in Configuration$, $\rho[l]$ denotes the output value of the operation with label l, $\phi[l']$ – the value set by adversary at operation with label l', and $\psi[l'']$ – the value of the random coins generated by $rs^{l''}$. The value $\psi[\mathit{secret}]$ is the secret message being exchanged. The motivation behind putting all the randomness used to one place is to be able to "replay" the run of the protocol in "lock-step" with another, earlier completed run. Special configuration $\top$ denotes that something inconsistent has happened during the computations.

Each operation with label l introduces a function from the dependencies used as arguments to the operation to the result of the operation. This function has type:

$$f^l : Configuration \to (\Sigma_\perp \cup \mathbb{B})^\top$$

For example, the operation $\mathit{keypair}^l(l_1, l_2)$, generating the key pair using random coins generated at l_2, and l_1 as the control dependency introduces the function:

$$f^l(\langle \rho, \phi, \psi \rangle) = \begin{cases} \underline{\mathit{keypair}}(\rho[l_2]) & \text{if } \rho[l_1] = \mathit{true} \wedge \rho[l_2] \neq \bot \\ \bot & \text{otherwise} \end{cases} \tag{1}$$

where $\underline{\mathit{keypair}}$ is the cryptographic function computing the public-secret key pair from the random coins. Semantics for the rest of the operations are defined in a similar way. The operations using the random source (*rs*, *secret*) or adversarial input (*req*, *receive*), use (in addition to ρ) the configuration components ϕ (for accessing the adversarial input) or ψ (for getting the random source). The function for $\mathit{mux}_n^l(l^c, l_1^c, l_1^v, \ldots, l_n^c, l_n^v)$ is defined to return the value the selector of which is *true*, and $\bot$ or $\top$ if less or more than one selector is *true*, respectively.

It can be verified that all the functions for evaluating operations' result are defined to be monotone. The graph evaluation function has the type

$$\mathit{Eval} : \mathit{Configuration} \rightarrow \mathit{Configuration} \tag{2}$$

and is defined as

$$\mathit{Eval}(\langle \rho, \phi, \psi \rangle) = \begin{cases} \top \text{ if } \exists l \in \mathit{Lab}. f^l(\langle \rho, \phi, \psi \rangle) = \top \\ \langle \rho', \phi, \psi \rangle \text{ where } \forall l \in \mathit{Lab}. \rho'[l] = f^l(\langle \rho, \phi, \psi \rangle) \text{ otherwise} \end{cases}$$
$$\mathit{Eval}(\top) = \top$$

The graph evaluation function is monotone, continuous and expanding ($\forall C. \mathit{Eval}(C) \geq C$).

The *execution* of a dependency graph, in parallel with the adversary $\mathcal{A}$, proceeds as follows:

1. ρ is set to map every dependency to $\bot$. ψ is initialized with the (uniformly generated) random coins used in the execution. ϕ (containing information on which of the protocol outputs are to be evaluated and the values to be fed to the graph from the network) is set to map every *req* operation to *false* and every *receive* operation to $\bot$.
2. The adversary produces a mapping $\phi' : \mathit{Lab}_{\leftarrow} \rightarrow \Sigma_{\bot}$ satisfying $\phi \leq \phi'$. The computational cost of outputting ϕ' is defined to be the number of labels l where $\phi(l) \neq \phi'(l)$.
3. The graph is evaluated — let $\langle \rho', \phi', \psi \rangle$ be the least fixed point of *Eval* that is greater or equal to $\langle \rho, \phi', \psi \rangle$. The existence of such fixed point follows from the properties of *Eval*.
4. Let $\mathit{Lab}_{\rightarrow} \subseteq \mathit{Lab}_{\bullet}$ be the labels of all *send*-operations. The adversary is given the values of ρ on all points of $\mathit{Lab}_{\rightarrow}$.
5. The adversary decides whether the sequence should be repeated from the step 2 (putting $\phi := \phi'$, $\rho := \rho'$) or terminated.