Handling Encryption in an Analysis for Secure Information Flow

Peeter Laud*

Tartu University and Cybernetica AS
peeter@cyber.ee

Abstract. This paper presents a program analysis for secure information flow. The analysis works on a simple imperative programming language containing a cryptographic primitive—encryption—as a possible operation. The analysis captures the intuitive qualities of the (lack of) information flow from a plaintext to its corresponding ciphertext. The analysis is proved correct with respect to a complexity-theoretical definition of the security of information flow. In contrast to the previous results, the analysis does not put any restrictions on the structure of the program, especially on the ways of how the program uses the encryption keys.

1 Introduction

Executing a program causes information about its inputs to flow to its outputs. If the inputs and outputs of a program are partitioned into public and secret ones then it is important to be sure that the program has *secure information flow* — no information about secret inputs flows to public outputs in a way that an adversary could make use of it.

If one wants to prove correct an analysis checking programs for secure information flow, one has to formalize when the public outputs of the program contain or do not contain information about secret inputs that is useful for an adversary. The usual formalization is *noninterference* [9] which states that the public outputs must not contain *any* information about the private inputs. In its variant for probabilistic systems [10], noninterference means that the probability of private inputs being equal to some value must be equal to the conditional probability of private inputs having that value, under the condition that the public outputs have a certain value.

Consider a program that takes two inputs — an encryption key k and a message M — and outputs $\mathcal{E}nc(k, M)$ — the encryption of M under key k. Obviously, $\mathcal{E}nc(k, M)$ contains information about M as it is in principle possible to find M from $\mathcal{E}nc(k, M)$. Hence, if M is secret and $\mathcal{E}nc(k, M)$ is public then the noninterference property does not hold. On the other hand, the security of $\mathcal{E}nc$ requires that an adversary, whose resources (computation time and space) have certain bounds, cannot derive anything about M from $\mathcal{E}nc(k, M)$. If we consider only such bounded adversaries then we could deem this program secure. The bounds on working time (and space) of the adversaries

* Supported by Estonian Science Foundation grant #5279. Most of this work was done while the author was at the University of Saarland, Germany.

are lax enough (probabilistic polynomial time) for all realistic adversaries to satisfy them. This example shows that for taking into account the effects of cryptographic primitives, we need a bit weaker definition of noninterference. We give that definition in this paper.

Having given that weaker definition, we obviously want to know which programs satisfy it and which ones do not. We use static program analysis to determine this. This analysis is the other contribution of this paper. The analysis also takes into account the intuitive qualities of the encryption operation.

The structure of this paper is the following. In Sec. 2 we describe some related work. Particularly, we describe our own earlier results [11] and explain what they were lacking. In Sec. 3 we explain what the security of an encryption scheme means. We also explain what the sameness means in cryptography. In Sec. 4 we introduce our programming language and give the definition of secure information flow. The information flow is deemed secure if certain two probability distributions, containing the inputs and outputs of the program, are "the same". In Sec. 5 we describe the structures that the analysis works on; these structures are abstractions of probability distributions over program states. Sec. 6 presents the analysis itself and also gives a small example of it in action. Sec. 7 says some words about the correctness proof. Finally, Sec. 8 concludes.

2 Related Work

Using program analysis for certification of secure information flow was pioneered by Denning and Denning [7, 8]. They annotated the program statements with the information flow between the variables caused by that statement, and analyzed this flow. Volpano et al. [24] gave a definition of secure information flow and accompanying analysis without using any instrumentations.

Leino and Joshi [13] define a program to be secure if, no matter what its secret inputs are, the public outputs always look the same for the same non-secret inputs. "Looking the same" is not specified further, different security definitions can be obtained by plugging in different formalizations. The security definition that we are using can be seen as an instance of theirs.

Recently, some work has been done to define weaker notions of secure information flow which allow analyzing programs containing cryptographic primitives without losing the precision one intuitively assigns to these primitives. Volpano and Smith [23, 22] have presented analyses of programs containing one-way functions as primitive operations. Unfortunately, one is quite restricted in using the one-way function if one wants to take advantage of the weakened security definition.

Another approach has been our own [11], analyzing programs containing encryption as a primitive operation, and having its own set of restrictions. The restriction was in the usage of encryption keys — their only allowed usage was as an encryption key. They were not allowed to occur in any other situations, for example in other expressions. Particularly, the encryption keys were not allowed to be plaintexts in encryptions. Such usage would have created dependencies (between values of different variables) that our abstraction could not keep track of.

There was also another restriction on programs. The equality and inequality of different variables storing encryption keys had to be known statically at each program point. Making the equality of keys depend on the inputs of the program again introduced dependencies that our abstraction could not keep track of.

There have also been attempts to precisely formalize and analyse cryptographic protocols. Instead of the usual assumption that cryptographic primitives are perfectly secure—they are modeled as functions for which only a very restricted set of formulas holds ([5, 1] are among the most prominent examples), one attempts to take into account that the cryptographic primitives may be implemented by any algorithm satisfying some complex complexity-theoretical definition; dealing with those definitions is an issue that both the analyses for secure information flow and the analyses of cryptographic protocols must handle. Mitchell et al. [14–16] extended the spi-calculus [1] with (polynomial) bounds on message lengths and execution time, and developed a probabilistic semantics for this extension. This has allowed them to prove the protocols correct with respect to polynomially bounded adversaries, where the cryptographic primitives that the protocols employ are real ones. These proofs are entirely hand-crafted, though; there are no mechanical means (like program analysis) to derive them. Pfitzmann et al. [18–20, 4] have given a framework to faithfully abstract the cryptographic primitives, such that the proofs about protocols using these abstractions would also hold if the abstractions are replaced with the actual primitives. Abadi and Rogaway [3] have shown that the formal construction of messages from simpler ones by tupling and encryption is computationally justified—if two formal messages look the same (where "looking the same" is defined over the formal structure; it makes the contents of the encrypted submessages irrelevant), and if the encryption primitive satisfies certain requirements, then no polynomially bounded adversary can distinguish the actual representations of these messages as bit-strings. This work was later extended by Abadi and Jürjens [2]. They considered program traces instead of expressions.

3 Cryptography and Secure Encryption

Encryption plays a big part in our contribution, so let us formally explain what it is and what its security means. In the course of this explanation we also cover the notion of *indistinguishability* — the computational equivalent of sameness.

An *encryption scheme* is a triple of algorithms $(\mathcal{G}, \mathcal{E}, \mathcal{D})$. They all must have running times polynomial to the length of their arguments. The algorithm $\mathcal{G}$ is the *key-generation algorithm*. It is invoked to create new encryption keys. The algorithm $\mathcal{G}$ takes one argument — the *security parameter* $n \in \mathbb{N}$ (represented in unary, because of the comment about the running times of algorithms) which determines the security of the system — more concretely, it determines the length of the keys. Larger security parameter means longer keys. The *encryption algorithm* takes as its arguments the security parameter, a key returned by $\mathcal{G}(1^n)$ (actually, we could assume that the security parameter is contained in that key but this is the usual presentation), and a plaintext — a bit-string. It returns the corresponding ciphertext. The arguments and the return value of the *decryption algorithm* are similar, only the places of plaintext and ciphertext are reversed. The key generation algorithm is obviously probabilistic, the decryption algorithm is deterministic. The encryption algorithm may either be deterministic or probabilistic but for satisfying the security requirements stated below it has to be probabilistic. It is required that the decryption of an encryption of a bit-string is equal to that bit-string.

The security requirement we put on the encryption scheme is the same as Abadi and Rogaway [3] used. We want the encryption to conceal the identity of both plaintexts and encryption keys and we want it also to hide the length of the plaintexts. The precise definition follows.

Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) algorithm, let $n \in \mathbb{N}$ and $b \in \{0,1\}$. Consider the following experiment $\mathbf{Exp}^{\mathcal{A}}(n)$:

1. Generate a random bit $b \in \{0,1\}$ by tossing a fair coin.
2. Define two black boxes $\mathcal{O}_1$ and $\mathcal{O}_2$. A black box is something that can be queried with bit-strings, for each query the black box returns an answer — another bit-string. There are no other ways to find out the implementation details of a black box. The contents of the boxes $\mathcal{O}_1$ and $\mathcal{O}_2$ depends on the value of b:
   - If $b = 0$, then generate two keys k, k' by invoking $\mathcal{G}(1^n)$ twice. Let $\mathcal{O}_1$ be a box that, on input $x \in \{0,1\}^*$, invokes $\mathcal{E}(1^n, k, x)$ and outputs its return value. Similarly, let $\mathcal{O}_2$ be a box that encrypts its inputs with the key k'.
   - If $b = 1$, then let $\mathbf{0} \in \{0,1\}^*$ be a fixed (and known to all) bit-string. Generate a key k by invoking $\mathcal{G}(1^n)$. Let $\mathcal{O}_1$ be a box that, on input $x \in \{0,1\}^*$, invokes $\mathcal{E}(1^n, k, \mathbf{0})$ and outputs its return value. Let $\mathcal{O}_2$ be identical to $\mathcal{O}_1$.
3. Invoke the algorithm $\mathcal{A}$, giving it 1^n as an argument, and also giving it (oracle) access to the black boxes $\mathcal{O}_1$ and $\mathcal{O}_2$. Let b^* be its output.
4. If $b = b^*$ then output true, else output false.

Consider the quantity $\mathbf{Adv}^{\mathcal{A}}(n) = 2 \cdot \Pr[\mathbf{Exp}^{\mathcal{A}}(n) = \mathsf{true}] - 1$. Here the probability is taken over the choice of b, as well as over the random choices of $\mathcal{G}$ (while generating the key(s)), $\mathcal{E}$ (while invoking the oracles) and $\mathcal{A}$. The quantity $\mathbf{Adv}^{\mathcal{A}}$ (called the *advantage* of $\mathcal{A}$; it shows how much better $\mathcal{A}$ is in guessing b, compared to simple coin-tossing) is a function from $\mathbb{N}$ to $\mathbb{R}$. We say that the encryption scheme is *type-0 secure*[1], if $\mathbf{Adv}^{\mathcal{A}}$ is *negligible* for all PPT algorithms $\mathcal{A}$. A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if its absolute value is asymptotically smaller than the reciprocal of any positive polynomial.

It is possible to construct type-0 secure encryption schemes (under standard assumptions). See [3] for details.

The security definition was an instance of demanding the *indistinguishability* of certain families (indexed by $n \in \mathbb{N}$) of probability distributions. In our case, we demanded the indistinguishability of the following families of distributions over pairs of black boxes:

$$\{\!|(\mathcal{E}(1^n, k, \cdot), \mathcal{E}(1^n, k', \cdot)) \,:\, k, k' \leftarrow \mathcal{G}(1^n)|\!\}_{n \in \mathbb{N}}$$

and

$$\{\!|(\mathcal{E}(1^n, k, \mathbf{0}), \mathcal{E}(1^n, k, \mathbf{0})) \,:\, k \leftarrow \mathcal{G}(1^n)|\!\}_{n \in \mathbb{N}} \;.$$

Here $x \leftarrow D$ denotes that the variable x is distributed according to the probability distribution D. The brackets $\{\!| \cdot |\!\}$ are used to construct new probability distributions.

[1] alternative name of this property is: repetition-concealing, which-key concealing and message-length concealing

For defining the indistinguishability of two families of probability distributions D and D' we have to change the wording of the description of the experiment $\mathbf{Exp}^{\mathcal{A}}(n)$ a bit. Namely, let the 2nd and 3rd point be the following:

2. Generate a quantity x, according to one of the probability distributions D_n or D'_n. If $b = 0$ then use D_n, else use D'_n.
3. Invoke the algorithm $\mathcal{A}$, giving it 1^n as an argument, and also giving it access to x.

The meaning of the phrase "access to x" depends on the type of x. If x is a bit-string then it is simply given as an argument to $\mathcal{A}$. If x is a black box then $\mathcal{A}$ is given oracle access to it. If x is a tuple then $\mathcal{A}$ is given access to all components of the tuple. Again, we consider the advantage of $\mathcal{A}$ and demand its negligibility for all PPT algorithms. We let $D \approx D'$ denote that D and D' are indistinguishable.

Our definition of secure information flow is given through the notion of *[computational] independence*, which is defined as follows. Let D be a family of probability distributions over some set of tuples. We assume that all tuples in that set have same arity and also same names of components. In the rest of this paper, the program state is represented as a tuple, its components are the values of the variables. If f is a tuple and X is a set of component names, then we let $f(X)$ denote the sub-tuple of f, consisting of only the components with names in X. Let X and Y be two sets of component names. We say that X and Y are independent (or that X is independent from Y) in the family of distributions D, if

$$\{\!|(f(X), f(Y)) \,:\, f \leftarrow D|\!\}_{n\in\mathbb{N}} \approx \{\!|(f(X), f'(Y)) \,:\, f, f' \leftarrow D|\!\}_{n\in\mathbb{N}} \ . \qquad (1)$$

4 Syntax, Semantics and Security Definition

The programs whose information flow we are studying are written in the following simple imperative programming language (the WHILE-language):

$$\mathsf{P} ::= x := o(x_1, \ldots, x_k) \mid \mathit{skip} \mid \mathsf{P}_1; \mathsf{P}_2 \mid \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2 \mid \mathit{while}\ b\ \mathit{do}\ \mathsf{P}' \ .$$

Here $x, x_1, \ldots, x_k, b$ are variables from the set $\mathbf{Var}$ and o is an *operator* from the set $\mathbf{Op}$. Each operator has a fixed arity. We assume that there are two special operators in the set $\mathbf{Op}$ — a binary operator $\mathcal{E}nc$ that denotes encryption, and a nullary operator $\mathcal{G}en$ that denotes the generation of new keys. Our analysis handles these two operators in a more optimistic way than others. Decryption is not handled differently from other operators, therefore it will not be mentioned any more.

Our security definition is given in terms of the inputs and outputs of the program, therefore it is natural to use denotational semantics. The denotational semantics $[\![\mathsf{P}]\!]$ of the program P maps the initial state of the program to the final state, i.e. its type is $\mathbf{State} \rightarrow \mathbf{State}_\perp$. For imperative programs, the state is a function mapping the variables to their values (or alternatively, a tuple of values, indexed by variables) from the set $\mathbf{Val}$. The extra element $\perp$ denotes non-termination. Note that the denotational semantics hides some aspects that may be observable in the real world — for example the running time of the program, the power consumption of the computer executing the program, the electromagnetic radiation emitted by that computer etc. Our security definition cannot take these aspects into account.

We mentioned that the encryption algorithm has to be probabilistic. If this is the case, then the semantics of programs also has to accommodate probabilism. The range of $[\![\mathsf{P}]\!]$ therefore has to be $\mathcal{D}(\mathbf{State}_\perp)$. Here $\mathcal{D}(X)$ denotes the set of all probability distributions over the set X. Another detail that the semantics has to incorporate is the security parameter. For this we let $[\![\mathsf{P}]\!]$ to be not just a single function from $\mathbf{State}$ to $\mathcal{D}(\mathbf{State}_\perp)$, but an entire family of functions (of the same type), indexed by $n \in \mathbb{N}$.

The rest of the definition of $[\![P]\!]_n$ is quite standard (see for example [17, Sec. 4.1]). First, we need semantics $[\![o]\!]_n$ for each operator $o \in \mathbf{Op}$. For a k-ary operator o, the semantics $[\![o]\!]_n$ is a function from $\mathbf{Val}^k$ to $\mathbf{Val}$. We demand that there exists a type-0 secure encryption scheme $(\mathcal{G}, \mathcal{E}, \mathcal{D})$, such that $[\![\mathcal{G}en]\!]_n = \mathcal{G}(1^n)$ and $[\![\mathcal{E}nc]\!]_n = \mathcal{E}(1^n, \cdot, \cdot)$. Now the semantics $[\![P]\!]_n$ is defined exactly as in Fig. 4.1 in [17] and we are not going to elaborate it here any more.

The model of security that we have in mind here is the following: There is a certain set of *private* variables $\mathbf{Var}_\mathrm{S} \subseteq \mathbf{Var}$ whose initial values we want to keep secret. After the program P has run, the values of the variables in a certain set $\mathbf{Var}_\mathsf{P} \subseteq \mathbf{Var}$ become *public*. The attacker tries to find out something about the initial values of secret variables. It can read the final values of public variables.

The possible inputs of the program P are somehow distributed. For the security parameter $n \in \mathbb{N}$, let their distribution be $D_n \in \mathcal{D}(\mathbf{State})$. The (structure of the) family of distributions D is assumed to be public knowledge.

We define security only for programs that run in (expected) polynomial time. We claim that this decision causes us no loss of generality. Namely, before the attacker obtains the final values of public variables, it is expected to wait for the program to finish its execution. If the program runs for too long time and the attacker keeps waiting then it cannot find out anything about the initial values of secret variables. Alternatively, at a certain moment the attacker may decide that the program is taking too long time to run and should be considered to be effectively nonterminating; the final state should be considered to be $\perp$. We "compose" the original program and the attacker's decision-making process about the running time of the program. The result is a program that runs in polynomial time. We could define the original program to be secure iff the composed program is. This composition amounts to running a clock parallel to the program (here "parallel to" means "interleaved with") and terminating after having run for a long enough time.

If the program runs in polynomial time then we no longer have to take the possible non-termination into account. Therefore the semantics of the program $[\![P]\!]$ transforms the initial family of probability distributions D to a final family of probability distributions $D' = [[\![P]\!]_n(D_n)]_{n\in\mathbb{N}}$ (we have somewhat abused the notation here). Let us make one more assumption — that the program does not change the values of the private variables $\mathbf{Var}_\mathrm{S}$. This assumption obviously is not a significant one — we can always add new variables to the program and use them instead of the ones in $\mathbf{Var}_\mathrm{S}$. We now say that the program P (with inputs distributed according to D) has *secure information flow* if $\mathbf{Var}_\mathrm{S}$ and $\mathbf{Var}_\mathrm{P}$ are [computationally] independent in the family of distributions D'.

5 Abstract Domain

The domain of the analysis is an abstraction of the set $\mathcal{D}(\mathbf{State})^{\mathbb{N}}$ — the set of families of probability distributions over $\mathbf{State}$. The analysis then maps the abstraction of the initial family of distributions D to an abstraction of the final family of distributions D'. Note that we said “an abstraction”, not “the abstraction” — the analysis is allowed to err to the safe side. The question of secure information flow is obviously incomputable therefore an always precise analysis cannot exist.

Let us introduce some notation first. Let $S \in \mathbf{State}$ and $x \in \mathbf{Var}$. Then $S(x)$ denotes the value of the variable x in the program state S. Additionally, we let $S([x]_{\mathcal{E}})$ denote a black box that encrypts its inputs, using $S(x)$ as the key. If we let $\widetilde{\mathbf{Var}}$ denote the set $\mathbf{Var} \cup \{[x]_{\mathcal{E}} : x \in \mathbf{Var}\}$ then we can assume that S is a tuple whose components are named with the elements of $\widetilde{\mathbf{Var}}$ — the state S contains all the values of program variables as well as all black boxes encrypting with these values.

The abstraction $A = \alpha(D)$ of a family of distributions D is a pair $(A_{\mathsf{indep}}, A_{\mathsf{key}})$ where $A_{\mathsf{indep}} \subseteq \mathcal{P}(\widetilde{\mathbf{Var}}) \times \mathcal{P}(\widetilde{\mathbf{Var}})$ (here $\mathcal{P}(X)$ denotes the power set of the set X) and $A_{\mathsf{key}} \subseteq \mathbf{Var}$ (so $A \in \mathcal{F}(\mathbf{Var}) = \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}}) \times \mathcal{P}(\widetilde{\mathbf{Var}})) \times \mathcal{P}(\mathbf{Var})$). Here the set A_{indep} contains all such pairs $(X, Y) \in \widetilde{\mathbf{Var}} \times \widetilde{\mathbf{Var}}$ where X and Y are independent in the family of distributions D. The set A_{key} contains all such variables $x \in \mathbf{Var}$ where the black box $S_n([x]_{\mathcal{E}})$, where S_n is distributed according to D_n, is indistinguishable from a “real” encrypting black box $\mathcal{E}(1^n, k, \cdot)$ where k is distributed according to $\mathcal{G}(1^n)$. “Erring to the safe side” while abstracting D means leaving out some elements from these two sets.

The introduction of the encrypting black boxes $[x]_{\mathcal{E}}$ allows us to track different “kinds of dependence”. As an example, let x, k and l be variables and let D be such a family of distributions, that the value of k is distributed as an encryption key, the value of x is some ciphertext that has been created by encrypting something with the key k, and the value of l is obtained from the value of k through a simple (and reversible) arithmetic operation. Then neither $\{l\}$ nor $\{x\}$ are independent from $\{k\}$ in the family of distributions D (for detecting whether the value of x and the value of k come from the same state or from different states, try to decrypt the value of x with the value of k and consider, whether the result is a sensible plaintext). However, the dependence of l and k is of quite different quality than the dependence of x and k. Someone that knows (the value of) l can decrypt ciphertexts encrypted with k. Someone that knows only x surely cannot do that. Independence from $[k]_{\mathcal{E}}$ distinguishes l and x. The sets $\{x\}$ and $\{[k]_{\mathcal{E}}\}$ are independent in the family of distributions D. The sets $\{l\}$ and $\{[k]_{\mathcal{E}}\}$ are not.

The analysis takes the program text and an abstraction A of the initial family of distributions D. The description of this family of distributions must be found from the context where the program is used. This description should be precise enough, such that a reasonable abstraction A can be deduced from it. Describing D and finding A is, however, not the topic of this paper. In most of earlier papers, an implicit assumption has been made that all variables are independent of each other.

6 Analysis

The analysis $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ is defined inductively over the structure of program P. Here $\mathbf{Var}$ denotes the set of variables currently in consideration. We have introduced it because for computing the analysis of certain programs we may need to analyse their subprograms with respect to more variables. For analyzing the program *if* b *then* P_1 *else* P_2 we need to introduce an extra variable while analyzing P_1 and P_2. This extra variable is used to keep track of the dependencies of the initial value of the variable b.

Let P be an assignment $x := o(x_1, \ldots, x_k)$ and let $A \in \mathcal{F}(\mathbf{Var})$ be an abstract value. In this case $A' = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](A)$ is defined in the following way:

- The sets A'_{indep} and A'_{key} contain all the elements to satisfy the rules on Fig. 1.
- A'_{indep} is symmetric: if $(X, Y) \in A'_{\mathsf{indep}}$ then also $(Y, X) \in A'_{\mathsf{indep}}$.
- A'_{indep} is monotone: if $(X, Y) \in A'_{\mathsf{indep}}$ and $X' \subseteq X$ and $Y' \subseteq Y$ then also $(X', Y') \in A'_{\mathsf{indep}}$.
- A'_{indep} and A'_{key} are the smallest sets satisfying the above conditions.

Let us explain this definition a bit. The requirements of symmetry and monotonicity have been added to decrease the number of different cases that the rules must cover. It is obvious (from the definition of independence) that it is safe to state that A'_{indep} is symmetric. It is almost as obvious that monotonicity is also a safe requirement — if X' and Y' are not independent, i.e. there exists an algorithm that can distinguish the two distributions in (1) for X' and Y', then the same algorithm can also distinguish these two distributions for X and Y.

As next we will explain what is the basis of the rules in Fig. 1. The rule (2) says that if the program P does not change the values of certain variables (namely those in sets X and Y) then their independence before the execution of the program implies their independence after the execution. The rule (3), as well as its variants say that if a certain set of variables (the set Y) is independent from another one then everything that can be computed from the values of these variables is still independent of that other set.

The rule (4) makes use of the type-0 security of the encryption scheme. Consider, what do we need for the independence of X and $Y \cup \{x\}$ in the final family of distributions.

First, by monotonicity X and Y must be independent in the final family of distributions, and by the rule (2) also in the initial family. Obviously, the variable k must be an encryption key. And if we want the value of x to appear like a random bit-string, then everything else in our possession (the values of variables and encrypting black boxes in X and Y) must not help us in decrypting it. We also have to add the value of y to the things that do not help us in decrypting x because the security definition of the encryption scheme does not cover the case where something that is related to the encryption key is encrypted with it (Abadi and Rogaway [3, Sec. 4.2] explain this case in more detail). As we have explained in Sec. 5, this non-relatedness corresponds to the last antecedent in the rule (4).

The rule (5) states that a "real" encrypting black box (which $[x]_{\mathcal{E}}$ in this case is) is independent of itself. This is a simple consequence of the security definition of the encryption scheme. The rules (6) and (6') state that if some value is distributed as an

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := o(x_1,\ldots,x_k)\\ (X,Y)\in A_{\mathsf{indep}}\\ x,[x]_{\mathcal{E}}\notin X\cup Y\end{array}}{(X,Y)\in A'_{\mathsf{indep}}} \tag{2}$$

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := o(x_1,\ldots,x_k)\\ (X,Y)\in A_{\mathsf{indep}}\\ x,[x]_{\mathcal{E}}\notin X\cup Y\\ x_1,\ldots,x_k\in Y\end{array}}{(X,Y\cup\{x,[x]_{\mathcal{E}}\})\in A'_{\mathsf{indep}}} \tag{3}$$

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := \mathcal{E}nc(k,y)\\ (X,Y)\in A_{\mathsf{indep}}\\ x,[x]_{\mathcal{E}}\notin X\cup Y\\ y,[k]_{\mathcal{E}}\in Y\end{array}}{(X,Y\cup\{x,[x]_{\mathcal{E}}\})\in A'_{\mathsf{indep}}} \tag{3'}$$

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := y\\ (X,Y)\in A_{\mathsf{indep}}\\ x,[x]_{\mathcal{E}}\notin X\cup Y\\ X' := X\cup\langle\!\langle y\in X\ \mathbf{?}\ \{x\}\ \mathbf{:}\ \emptyset\rangle\!\rangle\cup\langle\!\langle [y]_{\mathcal{E}}\in X\ \mathbf{?}\ \{[x]_{\mathcal{E}}\}\ \mathbf{:}\ \emptyset\rangle\!\rangle\\ Y' := Y\cup\langle\!\langle y\in Y\ \mathbf{?}\ \{x\}\ \mathbf{:}\ \emptyset\rangle\!\rangle\cup\langle\!\langle [y]_{\mathcal{E}}\in Y\ \mathbf{?}\ \{[x]_{\mathcal{E}}\}\ \mathbf{:}\ \emptyset\rangle\!\rangle\end{array}}{(X',Y')\in A'_{\mathsf{indep}}} \tag{3''}$$

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := \mathcal{E}nc(k,y)\\ (X,Y)\in A_{\mathsf{indep}}\\ x,[x]_{\mathcal{E}}\notin X\cup Y\\ k\in A_{\mathsf{key}}\\ (\{[k]_{\mathcal{E}}\},X\cup Y\cup\{y\})\in A_{\mathsf{indep}}\end{array}}{(X,Y\cup\{x,[x]_{\mathcal{E}}\})\in A'_{\mathsf{indep}}} \tag{4}$$

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := \mathcal{G}en()\\ (X,Y)\in A_{\mathsf{indep}}\\ x\notin X\cup Y\end{array}}{(X\cup\{[x]_{\mathcal{E}}\},Y\cup\{[x]_{\mathcal{E}}\})\in A'_{\mathsf{indep}}} \tag{5}$$

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := o(\ldots)\\ k\in A_{\mathsf{key}}\setminus\{x\}\end{array}}{k\in A'_{\mathsf{key}}} \tag{6}$$

$$\frac{\begin{array}{c}\mathsf{P}\text{ is }x := y\\ y\in A_{\mathsf{key}}\end{array}}{x\in A'_{\mathsf{key}}} \tag{6'}$$

$$\frac{\mathsf{P}\text{ is }x := \mathcal{G}en()}{x\in A'_{\mathsf{key}}} \tag{7}$$

Fig. 1. The analysis $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ for assignments

encryption key before the execution then the same *value* is still distributed as an encryption key afterwards. Last, the rule (7) states that the operation $\mathcal{G}en$ generates encryption keys.

Let us go on with the definition of $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$. The analysis $\mathbb{A}^{(\mathbf{Var})}[\![skip]\!]$ is the identity function over $\mathcal{F}(\mathbf{Var})$. Also, $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}_1;\mathsf{P}_2]\!]$ is the composition of $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}_2]\!]$ and $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}_1]\!]$. Consider now the program *if* b *then* P_1 *else* P_2. Let $\mathbf{Var}_{\mathrm{asgn}} \subseteq \mathbf{Var}$ be the set of variables that are assigned to in at least one of the programs P_1 and P_2. Let N be a variable that is not an element of $\mathbf{Var}$ and let $\mathbf{Var}' = \mathbf{Var} \uplus \{N\}$. Given an abstract value $A \in \mathcal{F}(\mathbf{Var})$, representing the abstraction of the initial family of distributions, compute

$$A^{(1)} = \mathbb{A}^{(\mathbf{Var}')}[\![N := b; \mathsf{P}_1]\!](A)$$
$$A^{(2)} = \mathbb{A}^{(\mathbf{Var}')}[\![N := b; \mathsf{P}_2]\!](A) \ .$$

So, we have used the extra variable N to "save" the initial value of b. In the analyses of P_1 and P_2, the variable N appears where the initial value of b would have appeared.

The next step is to take the meet of the analyses of P_1 and P_2. The order on $\mathcal{F}(\mathbf{Var})$ is defined so that larger values are more precise and smaller values more conservative, therefore the meet of two values is the most precise value that is at least as conservative as any of them. Let $A'' \in \mathcal{F}(\mathbf{Var}')$ be such, that $A''_{\mathsf{indep}} = A^{(1)}_{\mathsf{indep}} \cap A^{(2)}_{\mathsf{indep}}$ and $A''_{\mathsf{key}} = A^{(1)}_{\mathsf{key}} \cap A^{(2)}_{\mathsf{key}}$. As the last step, we have to record the flow of information from N to the variables in $\mathbf{Var}_{\mathrm{asgn}}$. The analysis result $A' = \mathbb{A}^{(\mathbf{Var})}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2]\!]$ is defined as follows:

- The sets A'_{indep} and A'_{key} contain all the elements to satisfy the rules on Fig. 2. Here $\widetilde{\mathbf{Var}_{\mathrm{asgn}}}$ denotes the set $\mathbf{Var}_{\mathrm{asgn}} \cup \{[x]_{\mathcal{E}} \ : \ x \in \mathbf{Var}_{\mathrm{asgn}}\}$.
- A'_{indep} is symmetric and monotone.
- A'_{indep} and A'_{key} are the smallest sets satisfying the above conditions.

Some explanation is in order for the rules in Fig. 2, too. In the rule (8), the only entities that may have been modified in one of the branches are the encrypting black boxes $[x_1]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}$. They are distributed in the same way at the ends of both branches — no matter what the branch was, they are "real" encrypting black boxes. As they also are independent of everything else (this is stated by the first group of antecedents and by the antecedent just above it), their values cannot give away which of the branches was taken. The second group of antecedents states that for having $[x_i]_{\mathcal{E}}$ independent of itself after the *if*-statement, it also has to be independent of itself at the end of both branches.

The rule (9) says that if something is independent of N (the initial value of the guard variable b) and Y at the end of both branches then it is also independent of Y after the *if*-statement. This follows from the possibility to find the values of the variables and black boxes in Y after the *if*-statement, if we know their values at the end of both branches and we also know which branch was taken. The additional black boxes $[x_1]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}$ that are in the other side of the pair of variables and encrypting black boxes, have to satisfy similar conditions as in the previous rule.

The rule (10) is the same as the rule (6). But if k may have been changed in the branches then its distribution as a key at the end of both branches does not necessarily

$$
\frac{\begin{array}{c}
(X, Y) \in A''_{\mathsf{indep}} \\
x_1, \ldots, x_l, x_{l+1}, \ldots, x_m \in \mathbf{Var}_{\mathrm{asgn}} \\
(\{[x_1]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}, X \cup Y \cup \{N\}) \in A''_{\mathsf{indep}} \\
\left[\begin{array}{c}
(\{[x_1]_{\mathcal{E}}\}, \{[x_2]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}} \\
(\{[x_2]_{\mathcal{E}}\}, \{[x_3]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}} \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
(\{[x_{m-1}]_{\mathcal{E}}\}, \{[x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}}
\end{array}\right] \\
\left[\begin{array}{c}
(\{[x_{l+1}]_{\mathcal{E}}\}, \{[x_{l+1}]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}} \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
(\{[x_m]_{\mathcal{E}}\}, \{[x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}}
\end{array}\right] \\
x_1, \ldots, x_m \in A''_{\mathsf{key}} \\
(X \cup Y) \cap (\mathbf{Var}_{\mathrm{asgn}} \cup \{N\}) = \emptyset
\end{array}}{(X \cup \{[x_1]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}, Y \cup \{[x_{l+1}]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}) \in A'_{\mathsf{indep}}} \tag{8}
$$

$$
\frac{\begin{array}{c}
x_1, \ldots, x_l, x_{l+1}, \ldots, x_m, y_1, \ldots, y_r, y_{r+1}, \ldots, y_s \in \mathbf{Var}_{\mathrm{asgn}} \\
\big(X \cup \{[x_1]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}, \\
Y \cup \{N, [x_{l+1}]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}, y_1, [y_1]_{\mathcal{E}}, \ldots, y_r, [y_r]_{\mathcal{E}}, [y_{r+1}]_{\mathcal{E}}, \ldots, [y_s]_{\mathcal{E}}\}\big) \\
\in A''_{\mathsf{indep}} \\
(X, \{[x_1]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}} \\
\left[\begin{array}{c}
(\{[x_1]_{\mathcal{E}}\}, \{[x_2]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}} \\
(\{[x_2]_{\mathcal{E}}\}, \{[x_3]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}} \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
(\{[x_{m-1}]_{\mathcal{E}}\}, \{[x_m]_{\mathcal{E}}\}) \in A''_{\mathsf{indep}}
\end{array}\right] \\
x_1, \ldots, x_m \in A''_{\mathsf{key}} \\
(X \cup Y) \cap (\mathbf{Var}_{\mathrm{asgn}} \cup \{N\}) = \emptyset
\end{array}}{\begin{array}{c}
\big(X \cup \{[x_1]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}, \\
Y \cup \{[x_{l+1}]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}, y_1, [y_1]_{\mathcal{E}}, \ldots, y_r, [y_r]_{\mathcal{E}}, [y_{r+1}]_{\mathcal{E}}, \ldots, [y_s]_{\mathcal{E}}\}\big) \\
\in A'_{\mathsf{indep}}
\end{array}} \tag{9}
$$

$$
\frac{\begin{array}{c}
k \in A''_{\mathsf{key}} \\
k \notin \mathbf{Var}_{\mathrm{asgn}}
\end{array}}{k \in A'_{\mathsf{key}}} \tag{10}
$$

$$
\frac{\begin{array}{c}
k \in \mathbf{Var}_{\mathrm{asgn}} \\
k \in A''_{\mathsf{key}} \\
(\{[k]_{\mathcal{E}}\}, \{N\}) \in A''_{\mathsf{indep}}
\end{array}}{k \in A'_{\mathsf{key}}} \tag{11}
$$

Fig. 2. Merging the branches together

guarantee its distribution as a key at the end of the *if*-statement. Namely, the value of k may have influenced, which of the branches was taken. But if the value of k has not influenced the chosen branch (i.e. k is independent of the initial value of the guard variable) then its distribution as a key at the end of both branches is sufficient for its distribution as a key at the end of the *if*-statement.

Consider now the program $while\ b\ do$ P and let $A \in \mathcal{F}(\mathbf{Var})$. Let $A^{(0)} = A$ and

$$A^{(i)} = \mathbb{A}^{(\mathbf{Var})}[\![if\ b\ then\ \mathsf{P}\ else\ skip]\!](A^{(i-1)}) \ .$$

Finally, the analysis value $\mathbb{A}^{(\mathbf{Var})}[\![while\ b\ do\ \mathsf{P}]\!](A)$ is defined as the meet (i.e. componentwise intersection) of all $A^{(i)}$. It is computable because the sequence of abstract values $A^{(0)}, A^{(1)}, A^{(2)}, \ldots$ stabilizes at some point. Stabilization is caused by the finiteness of the lattice $\mathcal{F}(\mathbf{Var})$ and by the monotonicity of the analysis.

Let us give an example of the analysis in action. Consider the following program.

1: $k_1 := \mathcal{G}en()$
2: $if\ b\ then\ k_2 := k_1\ else\ k_2 := \mathcal{G}en()$
3: $x_1 := \mathcal{E}nc(k_1, y_1)$
4: $x_2 := \mathcal{E}nc(k_2, y_2)$

With the help of the presented analysis, we can derive that $\{b\}$ is independent of $\{x_1, x_2\}$ at the end of the program (without making any assumptions about the initial distribution of values of variables). This program is a sequence of four statements (the second of which is an *if*-statement), let $A^{(0)}$ be the abstraction of the initial family of distributions and let $A^{(i)}$, where $1 \le i \le 4$ be the abstract value computed by the analysis after the i-th statement. We have

(A). $(\emptyset, \{b, y_1, y_2\}) \in A^{(0)}_{\mathsf{indep}}$, because $\emptyset$ is independent of everything else.
(B). $(\emptyset, \{b, y_1, y_2\}) \in A^{(i)}_{\mathsf{indep}}$, where $i \in \{1, \ldots, 4\}$, from (A) and rule (2).

We are not any more going to mention the use of rules (2), (6) and (10) below. Basically, if some pair of sets of variables or some variable belongs to a component of $A^{(i)}$, and if none of these variables are changed in the statements $i+1, \ldots, j$, then the same pair of sets of variables or the same variable also belongs to the same component of $A^{(j)}$.

(C). $(\{[k_1]_{\mathcal{E}}\}, \{b, y_1, y_2, [k_1]_{\mathcal{E}}\}) \in A^{(1)}_{\mathsf{indep}}$ by (A) and rule (5).
(D). $k_1 \in A^{(1)}_{\mathsf{key}}$ by rule (7).

As next we have to analyse the *if*-statement. Let N be a new variable and let $\mathbf{Var}' = \mathbf{Var} \cup \{N\}$. According to the description of the analysis, we have to compute

$$B^{(0)} = \mathbb{A}^{(\mathbf{Var}')}[\![N := b]\!](A^{(1)})$$
$$B^{\mathsf{true}} = \mathbb{A}^{(\mathbf{Var}')}[\![k_2 := k_1]\!](B^{(0)}) \qquad B^{\mathsf{false}} = \mathbb{A}^{(\mathbf{Var}')}[\![k_2 := \mathcal{G}en()]\!](B^{(0)})$$
$$A'' = B^{\mathsf{true}} \wedge B^{\mathsf{false}}$$

and $A^{(2)}$ from A'' by using the rules in Fig. 2. We have

(E). $(\{[k_1]_\mathcal{E}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}) \in B^{(0)}_{\mathsf{indep}}$ by (C) and rule (3).
(F). $(\{[k_2]_\mathcal{E}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}) \in B^{\mathsf{true}}_{\mathsf{indep}}$ by (E) and rule (3").
(G). $k_2 \in B^{\mathsf{true}}_{\mathsf{key}}$ by (D) and rule (6').
(H). $(\{[k_2]_\mathcal{E}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}) \in B^{\mathsf{false}}_{\mathsf{indep}}$ by (C), rule (5) and monotonicity.
(I). $k_2 \in B^{\mathsf{false}}_{\mathsf{key}}$ by rule (7).
(J). $(\{[k_2]_\mathcal{E}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}) \in A''_{\mathsf{indep}}$ by (F) and (H).
(K). $k_2 \in A''_{\mathsf{key}}$ by (G) and (I).
(L). $(\{[k_2]_\mathcal{E}\}, \{b, y_2\}) \in A^{(2)}_{\mathsf{indep}}$ by (A), (J), (K) and rule (8).
(M). $(\{[k_2]_\mathcal{E}, b, y_1, y_2\}, \{[k_1]_\mathcal{E}\}) \in A^{(2)}_{\mathsf{indep}}$ by (C), (J), (K) and rule (8).
(N). $k_2 \in A^{(2)}_{\mathsf{key}}$ by (K), (J) and rule (11).
(O). $(\{b\}, \{x_1\}) \in A^{(3)}_{\mathsf{indep}}$ by (A), (D), (C) and rule (4).
(P). $(\{[k_2]_\mathcal{E}\}, \{b, x_1, y_2\}) \in A^{(3)}_{\mathsf{indep}}$ by (L), (D), (M) and rule (4).
(Q). $(\{b\}, \{x_1, x_2\}) \in A^{(4)}_{\mathsf{indep}}$ by (O), (N), (P) and rule (4).

The domain of the analysis — $\mathcal{F}(\mathbf{Var})$ — is quite large, therefore we may ask, whether the analysis can be implemented in a way that does not cause prohibitive running times. It turns out that it can indeed be implemented in such a way. The set $\mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}}) \times \mathcal{P}(\widetilde{\mathbf{Var}}))$ is isomorphic to the set of formulas of propositional calculus, where the set of variables is $\widetilde{\mathbf{Var}} \uplus \widetilde{\mathbf{Var}}$. Indeed,

$$\mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}}) \times \mathcal{P}(\widetilde{\mathbf{Var}})) \cong \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}} \uplus \widetilde{\mathbf{Var}})) \cong \{0,1\}^{\widetilde{\mathbf{Var}} \uplus \widetilde{\mathbf{Var}}} \to \{0,1\} \ .$$

These formulas can be implemented as binary decision diagrams (BDD). The analysis will then transform one BDD to another one. The rules on Fig. 1 and Fig. 2 are such, that these transformations can be efficiently implemented on BDD-s. We believe from our experimentation with the implementation that the size of the abstract domain will not be the cause of long running times — small programs like above example can be analyzed in split-second on a modern computer.

7 About the Proof of Correctness

The correctness of the analysis means that if the analysis says that two sets of variables are independent of each other at the end of the execution of the program, then it really is so.

Proving the rules in Fig. 1 and Fig. 2 correct is simple, it takes some elementary cryptography. However, these proofs alone are not enough for the correctness of the entire analysis. They are enough for the correctness of everything but the analysis of loops.

For using the standard results [6] about the approximation of fixed points (the semantics of loops is defined through a fix-point operation, the same holds for the analysis of loops), we need the continuity of the abstraction function α from $\mathcal{D}(\mathbf{State}_\perp)^\mathbb{N}$ to $\mathcal{F}(\mathbf{Var})$. However, as we show in [12, Sec. 3.3], there are no non-trivial (abstraction) functions α from $\mathcal{D}(\mathbf{State}_\perp)^\mathbb{N}$ with the following properties:

1. α is continuous;
2. if $D, \tilde{D} \in \mathcal{D}(\mathbf{State}_\perp)^\mathbb{N}$ are such that $D \approx \tilde{D}$, then $\alpha(D) = \alpha(\tilde{D})$.

The second requirement should come as something obvious, we want to abstract away everything that does not affect polynomial-time computations.

In [12, Sec. 3.3] we show that for all $D, \tilde{D} \in \mathcal{D}(\mathbf{State}_\perp)^\mathbb{N}$ there exist $D^{(i)}, \tilde{D}^{(i)} \in \mathcal{D}(\mathbf{State}_\perp)^\mathbb{N}$ (here $i \in \mathbb{N}$), such that the least upper bound of the family $\{D^{(i)}\}_{i\in\mathbb{N}}$ is D, the least upper bound of the family $\{\tilde{D}^{(i)}\}_{i\in\mathbb{N}}$ is $\tilde{D}$, and $D^{(i)} \approx \tilde{D}^{(i)}$ holds for each $i \in \mathbb{N}$. Therefore $\alpha(D^{(i)}) = \alpha(\tilde{D}^{(i)})$ by the second condition on α and $\alpha(D) = \alpha(\tilde{D})$ by the first.

Our abstraction function α cannot therefore be continuous. We have devised an *ad hoc* proof of the correctness of the analysis. If D is the initial family of distributions, $A = \alpha(D)$ and $A' = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](A)$, then to show that $(X, Y) \in A'_{\mathsf{indep}}$ implies the independence of X and Y in the final distribution D', we first fix the security parameter n. Then we construct two "slices" of the program P, whose output distributions are the left and right distribution in (1), respectively. We then introduce a number of possible steps for transforming these slices. We show that

1. The first slice can be transformed to second in a number of steps polynomial in n. This sequence of steps can be efficiently constructed.
2. Each step has only a negligible effect on the output distribution.

This construction and transformation are described in [12, Chapter 4].

8 Conclusions

We have devised an analysis for secure information flow for programs containing encryptions. We believe that we have found the right abstractions this time, as the analysis puts no restrictions at all on the program structure. We do not even have the restriction that Abadi and Rogaway [3] had — we also allow *encryption cycles* — cases where encryption keys are encrypted with other keys, and where the relation "is encrypted with" is circular. This relation is defined by the program structure, therefore it is even a bit surprising that an analysis that does not keep track of the program structure is able to gracefully handle encryption cycles.

The main future direction for extending this work should be the inclusion of authentication primitives (signatures, MACs, etc.) and active adversaries. It may be hard to extend the full analysis, if we do not have convenient means for approximating fixed points, but we may try to devise the analysis for some kind of language that does not contain a looping construct. There exist simple intuitive formalisms (without a looping construct) for expressing cryptographic protocols, for example strand spaces [21].

Another extension would be the handling of other primitives for ensuring confidentiality. It should be quite easy to add public-key encryption to our language and analysis. In the analysis, the public keys would behave similarly to the encrypting black boxes — one can encrypt with them, but not decrypt.

References

1. M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.
2. M. Abadi and J. Jürjens. Formal Eavesdropping and Its Computational Interpretation. In proc. of *TACS 2001* (LNCS 2215), pages 82–94.
3. M. Abadi and P. Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In proc. of *International Conference IFIP TCS 2000* (LNCS 1872), pages 3–22.
4. M. Backes. *Cryptographically Sound Analysis of Security Protocols*. PhD thesis, Universität des Saarlandes, 2002.
5. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, Feb. 1990.
6. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science* 277(1-2):47–103, Apr. 2002.
7. D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.
8. D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.
9. J. Goguen and J. Meseguer. Security Policies and Security Models. In proc. of *IEEE S&P 1982*, pages 11–20.
10. J. Gray III. Probabilistic Noninterference. In proc. of *IEEE S&P 1990*, pages 170–179.
11. P. Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In proc. of *ESOP 2001* (LNCS 2028), pages 77–91.
12. P. Laud. *Computationally Secure Information Flow*. PhD thesis, Universität des Saarlandes, 2002.
13. K. Leino and R. Joshi. A Semantic Approach to Secure Information Flow. In proc. of *Matehematics of Program Construction '98* (LNCS 1422), pages 254–271.
14. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A Probabilistic Poly-Time Framework for Protocol Analysis. In proc. of *ACM CCS '98*, pages 112–121.
15. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic Polynomial-Time Equivalence and Security Analysis. In proc. of the *World Congress on Formal Methods in the Development of Computing Systems '99* (LNCS 1708), pages 776–793.
16. J. Mitchell. Probabilistic Polynomial-Time Process Calculus and Security Protocol Analysis. In proc. of *ESOP 2001* (LNCS 2028), pages 23–29.
17. H. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
18. B. Pfitzmann, M. Schunter, and M. Waidner. Cryptographic Security of Reactive Systems. In proc. of *Workshop on Secure Architectures and Information Flow* (ENTCS 32), 2000.
19. B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In proc. of *ACM CCS 2000*, pages 245–254.
20. B. Pfitzmann and M. Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In proc. of *IEEE S&P 2001*, pages 184–200.
21. F. Thayer, J. Herzog, and J. Guttman. Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.
22. D. Volpano. Secure Introduction of One-way Functions. In proc. of *CSFW '00*, pages 246–254.
23. D. Volpano and G. Smith. Verifying Secrets and Relative Secrecy. In proc. of *POPL 2000*, pages 268–276.
24. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.