A Type System for Computationally Secure Information Flow *

Peeter Laud and Varmo Vene
Tartu University
{peeter_l|varmo}@ut.ee

Abstract

The paper presents a novel type system for checking the security of information flow in programs containing operations of symmetric encryption. The type system is correct with respect to the complexity-theoretic security definitions of the encryption primitive.

Topics: semantics, cryptography.

1 Introduction and Related Work

Suppose that you have received a program that purports to help you to organize some kind of your personal data. The program has functionality to make that data available over a network, but only in encrypted form, so that only people designated by you may have access to it. How can you be sure that the program really does what it claims to do, and does not leak your data to someone not entitled to it?

Here we have an instance of the problem of *secure information flow* (SIF). The security here is *not absolute*, though. If the program acts as promised then it indeed leaks your personal data — someone that is able to break the encryption can recover it. At stake here is *computational* security — if we assume that the encryption cannot be broken with realistic resources, does that also mean that your data is safe against all realistic adversaries?

When blindly trusting the source of the program is not an option, we have to verify it somehow. One possibility for such a verification is typing the program with a type system that ensures that correctly typed programs have SIF. Using

*Supported by Estonian Science Foundation, grant #5279

static analysis for certification of SIF was pioneered by Denning and Denning [Den76, DD77]. The correctness of the analysis was not proven directly from the semantics of the program, though. Volpano et al. [VSI96] gave a definition of SIF without using any instrumentations. They also gave a type system which could check whether programs satisfy this definition. In subsequent papers [SV98, VS98] they have extended their approach to richer sets of programming constructions, and have also attempted to handle operations that provide non-information-theoretical security — namely one-way functions [VS00, Vol00]. The work in this paper can be seen as a significant extension to their approach. The interest in security types is motivated by the relative ease of incorporating them into existing programming languages [Mye99]. A good overview of research on language-based information flow security is given by Sabelfeld and Myers [SM03].

Type systems for handling cryptographic operations were first investigated by Abadi [Aba99]; a more recent paper is [AB03]. In his approach, the type of each piece of data expresses the intended uses of that data; the type of keys shows what kind of data it is intended to encrypt so as to make it safe to communicate over public channels. Our approach is somewhat different — our types reflect the source of the data; communicating data over public channels is safe if the sources are not sensitive. The main difference between their and our approaches is however the computational setting — their type systems are correct with respect to the Dolev-Yao model [DY83] while ours is correct with respect to complexity-theoretic definitions of security [Yao82].

Automatic or computer-assisted handling of cryptographic operations while remaining true to the complexity-theoretic security definitions has also attracted research but the emphasis or results of the approaches have been more or less different from the current paper. Universal composability [Can01, PW01] is a very general approach that strives to abstract away from the complexity-theoretic details of cryptographic primitives so that Dolev-Yao-style, but computationally justified arguments become possible. Tools for manipulating these abstractions have unfortunately failed to appear until now. Also, the current best abstraction of symmetric encryption [BP04] has some restrictions. Lincoln et al. [LMMS98, LMMS99] have given probabilistic semantics to spi-calculus [AG99], stated some protocols and proved them correct with respect to this semantics and the complexity-theoretic security definitions, but there is no tool support. Abadi et al. [AR00, AJ01] have given an automatic means to decide when certain reasonable computational interpretations of two messages in Dolev-Yao model are indistinguishable. We [Lau01, Lau03] have proposed program analyses for checking SIF in presence of encryption in programs.

In this paper we present a type system that can also gracefully handle symmetric encryptions as operations in the program, and, as the first of its kind, is correct with respect to the *complexity-theoretic* security definitions of the cryptographic primitive, instead of being based on the Dolev-Yao model.

This paper has the following structure. In Sec. 2 we state the preliminaries — we define our program language, its semantics, and the security of both information flow and encryption systems. Sec. 3 presents the type system and Sec. 4 states its correctness theorem and gives an overview of its proof. Sec. 5 concludes.

2 The Settings

We consider programs in a simple imperative language (the WHILE-language) whose expressions E and programs P are defined by the following grammar:

$$\begin{array}{rcl} \mathsf{P} & ::= & x := E \mid \mathit{skip} \mid \mathsf{P}_1; \mathsf{P}_2 \mid \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2 \mid \mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1 \\ E & ::= & o(x_1, \ldots, x_k) \end{array}$$

Here $x, x_1, \ldots, x_k, b$ are variables from the set $\mathbf{Var}$ and o is an operator from the set of operators $\mathbf{Op}$. The set $\mathbf{Op}$ has to contain two special operators — a nullary operator $\mathcal{G}en$ that denotes the generation of new encryption keys, and a binary operator $\mathcal{E}nc$ denoting the symmetric encryption. Our type system does not handle decryption specifically, therefore we do not mention it here.

Semantics. As our type system is proved correct mostly by showing the bisimilarity of certain programs, we use (small-step) structural operational semantics as the main method to describe what a program does. The encryption system that gives the semantics to the operators $\mathcal{G}en$ and $\mathcal{E}nc$ must be probabilistic, otherwise it cannot satisfy the requirements we are going to put on it. Hence the semantics of the programs must be probabilistic as well. Let $\mathcal{D}(X)$ denote the set of all probability distributions over the set X. For $x \in X$ let $\eta(x) \in \mathcal{D}(X)$ be the probability distribution that puts all its weight onto x. Let $x \leftarrow D$ denote that the random variable x is picked according to the probability distribution D. Let $\{\!|E : C|\!\}$ denote the distribution of E under the conditions C. A state of the program is a mapping from $\mathbf{Var}$ to the set of values $\mathbf{Val} = \{0,1\}^*$. A program configuration is a pair of a program (yet to be executed) and a state. A probabilistic program configuration is a pair of a program and a probability distribution over states. The semantics is a relation from program configurations to probabilistic program configurations and probability distributions over states. We assume that each k-ary operator $o \in \mathbf{Op}$ has been given a semantics $[\![o]\!] : \mathbf{Val}^k \to \mathcal{D}(\mathbf{Val})$. The semantics of programs is given in Fig. 1. Note that $\longrightarrow$ is a function.

For defining security we also have to state what the outcome of the program is. A *program run* is a sequence $C_0 \xrightarrow{p_1} C_1 \xrightarrow{p_2} \cdots \xrightarrow{p_n} C_n$ where $C_0, \ldots, C_{n-1}$ are program configurations and C_n is a program state. If $C_{i-1} = \langle \mathsf{P}_{i-1}, S_{i-1} \rangle \longrightarrow$

$$\langle x := o(x_1,\ldots,x_k), S\rangle \longrightarrow \{\!|S[x\mapsto v] \,:\, v \leftarrow [\![o]\!](S(x_1),\ldots,S(x_k))|\!\} \quad (1)$$

$$\langle skip, S\rangle \longrightarrow \eta(S) \quad (2)$$

$$\frac{\langle \mathsf{P}_1, S\rangle \longrightarrow D}{\langle \mathsf{P}_1;\mathsf{P}_2, S\rangle \longrightarrow \langle \mathsf{P}_2, D\rangle} \quad (3)$$

$$\frac{\langle \mathsf{P}_1, S\rangle \longrightarrow \langle \mathsf{P}_1', D\rangle}{\langle \mathsf{P}_1;\mathsf{P}_2, S\rangle \longrightarrow \langle \mathsf{P}_1';\mathsf{P}_2, D\rangle} \quad (4)$$

$$\frac{S(b) = \mathsf{true}}{\langle \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2, S\rangle \longrightarrow \langle \mathsf{P}_1, \eta(S)\rangle} \quad (5)$$

$$\frac{S(b) = \mathsf{false}}{\langle \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2, S\rangle \longrightarrow \langle \mathsf{P}_2, \eta(S)\rangle} \quad (6)$$

$$\frac{S(b) = \mathsf{true}}{\langle \mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1, S\rangle \longrightarrow \langle \mathsf{P}_1; \mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1, \eta(S)\rangle} \quad (7)$$

$$\frac{S(b) = \mathsf{false}}{\langle \mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1, S\rangle \longrightarrow \eta(S)} \quad (8)$$

Figure 1: The operational semantics of programs

$\langle \mathsf{P}_i, D_i\rangle$ then C_i must be equal to $\langle \mathsf{P}_i, S_i\rangle$ for some state S_i and $p_i = D_i(S_i)$ (if $i = n$ then we have just D_n instead of $\langle \mathsf{P}_i, D_i\rangle$). The *probability* of a run is the product of all p_i that it contains. Let $\mathbf{State}_\perp = \mathbf{State} \mathbin{\dot\cup} \{\perp\}$ where $\perp$ denotes nontermination. If $\langle \mathsf{P}, S\rangle$ is a configuration and $D \in \mathcal{D}(\mathbf{State}_\perp)$ then we write $\langle \mathsf{P}, S\rangle \Longrightarrow D$ if for all $S \in \mathbf{State}$, $D(S)$ equals the sum of the probabilities of all runs starting with $\langle \mathsf{P}, S\rangle$ and ending with S. The relation $\Longrightarrow$ defines the result of running a program on an initial state.

Encryption Systems. An *encryption system* 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 system is the same as Abadi and Rogaway [AR00] 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. Formally, for all probabilistic polynomial-time (PPT) algorithms $\mathcal{A}$ (with access to two oracles) the difference

$$\mathbf{P}[\mathcal{A}^{\mathcal{E}(1^n,k,\cdot),\mathcal{E}(1^n,k',\cdot)}(1^n) = 1 \mid k, k' \leftarrow \mathcal{G}(1^n)] - \mathbf{P}[\mathcal{A}^{\mathcal{E}(1^n,k,\mathbf{0}),\mathcal{E}(1^n,k,\mathbf{0})}(1^n) = 1 \mid k \leftarrow \mathcal{G}(1^n)] \quad (9)$$

must be a negligible function in n where $\mathbf{0}$ is a fixed bit-string. Here $\mathcal{E}(1^n, k, \cdot)$ is an oracle that encrypts its inputs with key k, and $\mathcal{E}(1^n, k, \mathbf{0})$ is an oracle that discards its inputs and returns encryptions of $\mathbf{0}$ (under key k) instead. A function is negligible if it is asymptotically smaller than the reciprocal of any polynomial.

We see that an encryption system does not just define a nullary and two binary algorithms, but it defines an entire family (indexed by $n \in \mathbb{N}$) of them. The semantics of programs therefore also has to be indexed by the security parameter n. Instead of a single relation $\longrightarrow$ we have a family $\{\xrightarrow[n]{}\}_{n\in\mathbb{N}}$. The semantics of a k-ary operation o is a family of probabilistic functions $[\![o]\!]_n : \mathbf{Val}^k \to \mathcal{D}(\mathbf{Val})$. We require that $[\![\mathcal{G}en]\!]_n = \mathcal{G}(1^n)$ and $[\![\mathcal{E}nc]\!]_n = \mathcal{E}(1^n, \cdot, \cdot)$ for some encryption system $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ satisfying (9).

Secure Information Flow. We have to fix what are the secret inputs and the public outputs of a program. For simplicity, let there be two fixed subsets $\mathbf{Var}_\mathrm{S}, \mathbf{Var}_\mathrm{P} \subseteq \mathbf{Var}$. The secret inputs are the initial values of the variables in $\mathbf{Var}_\mathrm{S}$, and the public outputs are the final values of the variables in $\mathbf{Var}_\mathrm{P}$.

Our definition of security is termination-insensitive. Sensitivity to termination is orthogonal to other issues and can be easily added as an afterthought [VS97]. Actually, non-termination cannot be detected at all, but running for a too long time can. In our setting, superpolynomial (in the security parameter) running time is definitely too long because encryption is secure only against polynomial-time adversaries. We say that a program P runs in *expected polynomial time* if there exists a polynomial q and a negligible function α, such that the sum of the probabilities of all program runs (for the semantics $\xrightarrow[n]{}$) of length at most $q(n)$ is at least $1 - \alpha(n)$.

The inputs of the program have to come from somewhere; when the program is run then its input state is picked from some probability distribution over program states. The nature of that distribution can have a profound effect on the security of the program. If the family of input distributions (indexed by the security parameter) D is not polynomial-time samplable (i.e. there exists no PPT algorithm $\mathcal{A}$ whose outputs on input 1^n are distributed as D_n) then some effects not achievable in polynomial time may happen during the program run and we no longer can be sure that the encryption is secure. Another consideration is, that we are interested whether *the program* leaks the secret inputs or not, so we want to exclude the cases where the secrets have already been leaked before running the program. We say that a family of distributions D over program states *isolates the secrets* if the values of the variables in $\mathbf{Var}_{\mathrm{S}}$ are *computationally independent* of the values of the rest of the variables. I.e. the families of probability distributions

$$\{\!|(S_n|_{\mathbf{Var}_{\mathrm{S}}}, S_n|_{\mathbf{Var}\setminus\mathbf{Var}_{\mathrm{S}}}) \ : \ S_n \leftarrow D_n|\!\} \tag{10}$$

and

$$\{\!|(S_n|_{\mathbf{Var}_{\mathrm{S}}}, S'_n|_{\mathbf{Var}\setminus\mathbf{Var}_{\mathrm{S}}}) \ : \ S_n, S'_n \leftarrow D_n|\!\} \tag{11}$$

have to be indistinguishable, i.e. no PPT algorithm that is given either a sample of (10) or (11) can tell with probability non-negligibly higher than $1/2$ which of these two distributions the sample was taken from.

A program P that runs in expected polynomial time has *computationally secure information flow* (CSIF) if the secret inputs and public outputs of the program are computationally independent. I.e. for all polynomial-time samplable families of probability distributions D that isolate the secrets, the families of probability distributions

$$\{\!|(S_n|_{\mathbf{Var}_{\mathrm{S}}}, S'_n|_{\mathbf{Var}_{\mathrm{P}}}) \ : \ S_n \leftarrow D_n, \langle\mathsf{P}, S_n\rangle \underset{n}{\Longrightarrow} D'_n, S'_n \leftarrow D'_n|\!\} \tag{12}$$

and

$$\{\!|(S_n|_{\mathbf{Var}_{\mathrm{S}}}, S'_n|_{\mathbf{Var}_{\mathrm{P}}}) \ : \ S_n, S''_n \leftarrow D_n, \langle\mathsf{P}, S''_n\rangle \underset{n}{\Longrightarrow} D'_n, S'_n \leftarrow D'_n|\!\} \tag{13}$$

have to be indistinguishable.

3 Type System

A *typing* assigns types to variables. The type of a variable indicates what kind of information is allowed to influence its value. The type also indicates whether the value of the variable is a valid encryption key. I.e. the type of a variable is a pair of an *information type* and a *usage type*. *Inference rules* allow to deduce the type of the program from the types of its variables. A typing is *valid* if the program has a type.

3.1 The Set of Types

The "basic" secrets that the program operates on are its secret inputs and also the encryption keys that the program operates on; we definitely have to keep track where they are flowing. We have to distinguish between different keys; if we did not then each key would potentially be able to decrypt any ciphertext. In our current approach we distinguish the keys statically — let $\mathcal{G}$ be a (finite) set whose elements we use to label the key generation statements $x := \mathcal{G}en$ in the program. We can distinguish two keys if they have been generated at statements with different labels. Let $\mathcal{T}_0 = \{\mathsf{h}\} \cup \mathcal{G}$; it is the set of basic secrets. Here h denotes the type of secret inputs.

Let $\mathcal{T}_1 = \{\{t\}_N \,|\, t \in \mathcal{T}_0, N \subseteq \mathcal{G}\}$. The type $\{t\}_N$ means that the information of type t has been encrypted with keys of the type N. To recover the information, one needs at least one key for each type that is contained in N. The set $\mathcal{T}_1$ is ordered — $\{t\}_N \leq \{t'\}_{N'}$ iff $t = t'$ and $N \supseteq N'$. We get less information out of something that is protected by more keys.

Let $\mathcal{T}_2 = \mathcal{P}(\mathcal{T}_1)$ (the power set). The elements of $\mathcal{T}_2$ denote the merging of information from several types in $\mathcal{T}_1$. The information types are basically elements of $\mathcal{T}_2$. The element $\emptyset \in \mathcal{T}_2$ denotes public data. The set $\mathcal{T}_2$ is ordered: $T \leq T'$ if $\forall \{t\}_N \in T\, \exists \{t'\}_{N'} \in T' : \{t\}_N \leq \{t'\}_{N'}$. The relation $\leq$ is a preorder (reflexive and transitive), so we identify T and T' whenever $T \leq T'$ and $T' \leq T$. In practice this amounts to the deletion of all non-maximal elements from $T \in \mathcal{T}_2$.

Besides the equivalence $\leq \cap \geq$ on $\mathcal{T}_2$ there is another one that corresponds to the usage of keys that are not protected by other keys. For example, if $T = \{\{\mathsf{h}\}_{\{1\}}, \{1\}_\emptyset\}$ then we the possible knowledge of a key generated at $1 \in \mathcal{G}$ allows us to recover h — T is equivalent to $T \cup \{\mathsf{h}\}$ (denote $T \equiv T \cup \{\mathsf{h}\}$). To formally define the relation $\equiv$ we introduce the sets T^N for $T \in \mathcal{T}_2$ and $N \subseteq \mathcal{G}$. The set T^N corresponds to the information that may be recovered if we have the information in T and possiblity to decrypt information with keys in N. They are defined as the least sets satisfying

$$\begin{array}{lrl} 1. & & T \subseteq T^\emptyset \\ 2. & N \subseteq N' \Rightarrow & T^N \subseteq T^{N'} \\ 3. & \{t\}_M \in T^N \Rightarrow & \{t\}_{M\setminus\{i\}} \in T^{N\cup\{i\}} \\ 4. & \{t\}_M \in T^{N\cup\{i\}} \wedge \{i\}_\emptyset \in T^N \Rightarrow & \{t\}_M \in T^N \\ 5. & \{i\}_\emptyset \in T^{N\cup\{i\}} \Rightarrow & \{i\}_\emptyset \in T^N \quad . \end{array} \tag{14}$$

Finally we define that $T \equiv T^\emptyset$. The items 1.-4. specify just the abilities of a Dolev-Yao attacker. The 5. item is used to break encryption cycles (see [AR00] for a more thorough discussion on them). This "Dolev-Yao attacker with the ability to break encryption cycles" first appeared in [Lau02].

There is a more direct way of computing $T^\emptyset$ than iterating the rules in (14). First we want to determine the set I of all key labels $i \in \mathcal{G}$ that occur as $\{i\}_\emptyset$ in $T^\emptyset$. We let $I \subseteq \mathcal{G}$ be the *largest* set satisfying

$$\big(\forall \{t\}_N \in T : t \neq i \vee N \not\subseteq I\big) \Rightarrow i \notin I \tag{15}$$

for all $i \in \mathcal{G}$. Such an I can be found by initializing I with $\mathcal{G}$ and then iterating (15).

Proposition 1. *Let $T \in \mathcal{T}_2$ and let I be defined as in (15). Then $T^\emptyset = \{\{t\}_{N\setminus I} \,|\, \{t\}_N \in T\}$ (up to the relation $\leq \cap \geq$).*

Proof. We start by showing that $I = \{i \,|\, \{i\}_\emptyset \in T^\emptyset\}$.
$\underline{I \subseteq \{i \,|\, \{i\}_\emptyset \in T^\emptyset\}}$. Let $I = \{i_1, \ldots, i_n\}$. From (15) it follows, that for any $i_j \in I$, there exists $\{i_j\}_N \in T$ where $N \subseteq I$. Hence, by using items 1.-3. of (14) we have that $\{i_j\}_\emptyset \in T^I$. But now, we can eliminate say key i_1 from superscripts by using the item 5.: $\{i_1\}_\emptyset \in T^I \Rightarrow \{i_1\}_\emptyset \in T^{I/\{i_1\}}$, and for all other keys using the item 4.: $\{i_j\}_\emptyset \in T^I \wedge \{i_1\}_\emptyset \in T^{I/\{i_1\}} \Rightarrow \{i_j\}_\emptyset \in T^{I/\{i_1\}}$. Analoguously, we can eliminate all other keys from superscripts, therefore getting that $\{i_j\}_\emptyset \in T^\emptyset$ for all j.
$\underline{\{i \,|\, \{i\}_\emptyset \in T^\emptyset\} \subseteq I}$. For all $\{i\}_\emptyset \in T^\emptyset$ it means that there is a type $\{i\}_{N_0} \in T$ and a sequence implications:

$$\{i\}_{N_0} \in T \Rightarrow \{i\}_{N_0} \in T^\emptyset \Rightarrow \{i\}_{N_1} \in T^{M_1} \Rightarrow \ldots \Rightarrow \{i\}_\emptyset \in T^\emptyset$$

where the first implication corresponds to the item 1. and the rest for items 2.-5. In particular, there is at least one implication corresponding to the item 4. for any key appearing in N_0. Now, consider a suffix of the sequence starting with the item 4.:

$$\{i\}_{N_k} \in T^{M_k} \wedge \{j\}_\emptyset \in T^{M_k\setminus\{j\}} \Rightarrow \{i\}_{N_k} \in T^{M_k\setminus\{j\}} \Rightarrow \ldots \Rightarrow \{i\}_\emptyset \in T^\emptyset$$

The tail of the sequence can be transformed to the following:

$$\{j\}_\emptyset \in T^{M_k\setminus\{j\}} \Rightarrow \ldots \Rightarrow \{j\}_\emptyset \in T^\emptyset$$

where any use of the item 3. is replaced by the corresponding use of item 2., uses of item 5. are replaced by the uses of item 4. with side-condition $\{i\}_\emptyset \in T^M$ (for suitable M), and all other implications use the items from the original sequence. Hence, if $j \in N_0$, then $\{j\}_\emptyset \in T^\emptyset$, and, therefore, all keys in the set $\{i \,|\, \{i\}_\emptyset \in T^\emptyset\}$ satisfy (15). But I is the largest such set, hence the required inclusion holds.

Now, for any $\{t\}_N \in T$, using repeatedly item 4. (and item 5., if necessary), we get that $\{t\}_{N\setminus I} \in T^\emptyset$. Hence $\{\{t\}_{N\setminus I} \,|\, \{t\}_N \in T\} \subseteq T^\emptyset$.

For the other way inclusion, let $\{t\}_N \in T^{\emptyset}$. Like in the proof that $\{i \mid \{i\}_{\emptyset} \in T^{\emptyset}\}$ is a subset of I, there exists sequence of implications, such that:

$$\{t\}_{N_0} \in T \Rightarrow \{t\}_{N_0} \in T^{\emptyset} \Rightarrow \{t\}_{N_1} \in T^{M_1} \Rightarrow \ldots \Rightarrow \{t\}_N \in T^{\emptyset}$$

And again, in the case of implications corresponding to the item 4. with the side condition $\{i\}_{\emptyset} \in T^M$, the rest of the sequence can be transformed to show that $\{i\}_{\emptyset} \in T^{\emptyset}$. Hence $i \in I$ and $N_0 \subseteq N \cup I$. □

In the following, when we talk about the elements of $\mathcal{T}_2$ then we mean the equivalence classes with respect to $(\leq \cap \geq) \sqcup \equiv$.

The set of *usage types* is $\mathcal{U} = \{\mathsf{Data}\} \cup \{\mathsf{Key}_N \mid N \subseteq \mathcal{G}\}$. If the usage type of variable is Data then its value is not usable as an encryption key. The value of a variable with the usage type Key_N is an encryption key generated at a key generation statement labeled with an element of N.

We can now state the actual sets of types for expressions, variables and programs. Define

$$\begin{array}{ll}\mathcal{T}_E & := \{\langle T, J, U\rangle \mid T, J \in \mathcal{T}_2, U \in \mathcal{U}, T \geq J\} \\ \mathcal{T}_V & := \{\langle T, U\rangle\, var \mid T \in \mathcal{T}_2, U \in \mathcal{U}\} \\ \mathcal{T}_C & := \{T\, cmd \mid T \in \mathcal{T}_2\} \ .\end{array}$$

For a program type $T\ cmd$ the type T is a lower bound on the information types of variables that are assigned to in this program. A variable type shows both the kinds of information that may be contained in that variable, and whether it may be used as an encryption key. The components T and U in an expression type $\langle T, J, U\rangle$ have the same meaning (for that expression). Additionally, J is an upper bound on the information that may control whether this expression is evaluated. A typing γ is a mapping from **Var** to $\mathcal{T}_V$. Compared to the type system of Volpano et al. [VSI96] the new details (besides the much richer set of information types) are the usage types for variables and expressions and the extra component in the expression types recording the information through implicit flow.

There is an order defined on $\mathcal{T}_E$:

$$\begin{array}{r} T \leq T' \wedge J \leq J' \Rightarrow \langle T, J, \mathsf{Data}\rangle \leq \langle T', J', \mathsf{Data}\rangle \\ T \leq T' \wedge J \leq J' \wedge N \subseteq N' \Rightarrow \langle T, J, \mathsf{Key}_N\rangle \leq \langle T', J', \mathsf{Key}_{N'}\rangle \\ J \leq J' \wedge T \cup \{\{i\}_{\emptyset} \mid i \in N\} \leq T' \Rightarrow \langle T, J, \mathsf{Key}_N\rangle \leq \langle T', J', \mathsf{Data}\rangle \ . \end{array}$$

The set $\mathcal{T}_C$ is ordered as well: $T\ cmd \leq T'\ cmd$ if $T \geq T'$. The set $\mathcal{T}_V$ is unordered. Let $\top$ be the greatest element of $\mathcal{T}_2$, i.e. $\top = \{\{t\}_{\emptyset} \mid t \in \mathcal{T}_0\}$. The rules for typing expressions and programs are given in Fig. 2. Note that the rule (18) is a "general" rule for typing expressions and it may also applied to encryptions and key generations.

$$\frac{\gamma \vdash e : \langle T', J', U' \rangle \quad \langle T', J', U' \rangle \leq \langle T, J, U \rangle}{\gamma \vdash e : \langle T, J, U \rangle} \tag{16}$$

$$\frac{\gamma \vdash \mathsf{P} : T'\ cmd \quad T'\ cmd \leq T\ cmd}{\gamma \vdash \mathsf{P} : T\ cmd} \tag{17}$$

$$\frac{\gamma \vdash e_i : \langle T, J, \mathsf{Data} \rangle}{\gamma \vdash o(e_1, \ldots, e_k) : \langle T, J, \mathsf{Data} \rangle} \tag{18}$$

$$\gamma \vdash \mathcal{G}en^i : \langle \emptyset, \emptyset, \mathsf{Key}_{\{i\}} \rangle \tag{19}$$

$$\frac{\gamma \vdash y : \langle T, J, \mathsf{Data} \rangle \quad \gamma \vdash k : \langle T, J, \mathsf{Key}_N \rangle}{\gamma \vdash \mathcal{E}nc(k, y) : \langle \{\, \{t\}_{M \cup \{i\}} |\, \{t\}_M \in T, i \in N\} \cup J, J, \mathsf{Data} \rangle} \tag{20}$$

$$\frac{\gamma(x) = \langle T, U \rangle\ var}{\gamma \vdash x : \langle T, \emptyset, U \rangle} \tag{21}$$

$$\frac{\gamma \vdash e : \langle T, J, \mathsf{Data} \rangle \quad \gamma(x) = \langle T, \mathsf{Data} \rangle\ var}{\gamma \vdash x := e : J\ cmd} \tag{22}$$

$$\frac{\gamma \vdash e : \langle T, J, \mathsf{Key}_N \rangle \quad \gamma(x) = \langle T, \mathsf{Key}_N \rangle\ var}{\gamma \vdash x := e : J\ cmd} \tag{23}$$

$$\gamma \vdash skip : \top\ cmd \tag{24}$$

$$\frac{\gamma \vdash \mathsf{P}_1 : T\ cmd \quad \gamma \vdash \mathsf{P}_2 : T\ cmd}{\gamma \vdash \mathsf{P}_1; \mathsf{P}_2 : T\ cmd} \tag{25}$$

$$\frac{\gamma \vdash e : \langle T, J, \mathsf{Data} \rangle \quad \gamma \vdash \mathsf{P}_1 : T\ cmd \quad \gamma \vdash \mathsf{P}_2 : T\ cmd}{\gamma \vdash if\ e\ then\ \mathsf{P}_1\ else\ \mathsf{P}_2 : T\ cmd} \tag{26}$$

$$\frac{\gamma \vdash e : \langle T, J, \mathsf{Data} \rangle \quad \gamma \vdash \mathsf{P} : T\ cmd}{\gamma \vdash while\ e\ do\ \mathsf{P} : T\ cmd} \tag{27}$$

Figure 2: Typing rules

3.2 Constraints Defined by Typing Rules

An alternative viewpoint on the rules in Fig. 2 is the following. Recall that we called a typing γ *valid* (for the program P) if $\gamma \vdash \mathsf{P} : T\ cmd$ is derivable for some $T \in \mathcal{T}_2$. The rules in Fig. 2 put certain constraints on γ. To further explain these rules, let us state these constraints explicitly. Each assignment statement $x := o(x_1, \ldots, x_k)$ in the program introduces a set of constraints. Let $b_1, \ldots, b_m$ be the variables controlling whether this assignment is executed (i.e. these are the conditional variables occurring in the if- and $while$-statements enclosing this assignment). Stating the constraints is simpler if we also introduce an order on $\mathcal{T}_V$ by defining

$$\begin{aligned} T \leq T' &\Rightarrow \langle T, \mathsf{Data}\rangle\ var \leq \langle T', \mathsf{Data}\rangle\ var \\ T \leq T' \wedge N \subseteq N' &\Rightarrow \langle T, \mathsf{Key}_N\rangle\ var \leq \langle T', \mathsf{Key}_{N'}\rangle\ var \\ T \cup \{\{i\}_\emptyset \,|\, i \in N\} \leq T' &\Rightarrow \langle T, \mathsf{Key}_N\rangle\ var \leq \langle T', \mathsf{Data}\rangle\ var \end{aligned}$$

and define $\gamma_{\mathsf{Data}}(x) \geq \gamma(x)$ to be the smallest type whose usage component equals Data.

- An assignment $x := o(x_1, \ldots, x_k)$ simply introduces the constraints $\gamma(x) \geq \gamma(x_i)$ and $\gamma(x) \geq \gamma_{\mathsf{Data}}(b_j)$ for all i and j. Also, the usage component of $\gamma(x)$ must be Data.

For the special kinds of assignments, there is a choice between the set of constraints we just stated and a set of constraints that depends of the statement.

- For $x := \mathcal{E}nc(k, y)$ the alternative set of constraints is the following. Let $\gamma(x) = \langle T_x, \mathsf{Data}\rangle\ var$, $\gamma(k) = \langle T_k, \mathsf{Key}_N\rangle\ var$, $\gamma_{\mathsf{Data}}(y) = \langle T_y, \mathsf{Data}\rangle\ var$, and $\gamma_{\mathsf{Data}}(b_j) = \langle B_j, \mathsf{Data}\rangle\ var$. The constraints $T_x \geq \{\, \{t\}_{M \cup \{i\}} |\, \{t\}_M \in T_y \cup T_k, i \in N\}$ and $T_x \geq B_j$ must then hold.

Such different handling of information flowing from k and y vs. the information flowing from b_j-s was the reason of introducing the second information-type component to the expression types. Notice that typing rule (20) is the only rule that handles two information-type components differently.

- For $x := \mathcal{G}en^i$, the alternative set of constraints is the following. Let $\gamma(x) = \langle T_x, \mathsf{Key}_N\rangle\ var$ and $\gamma_{\mathsf{Data}}(b_j) = \langle B_j, \mathsf{Data}\rangle\ var$. Then $i \in N$ and $T_x \geq B_j$.

- For $x := y$, where x and y are both keys, the alternative constraints are as follows. Let $\gamma(x) = \langle T_x, \mathsf{Key}_{N_x}\rangle\ var$, $\gamma(y) = \langle T_y, \mathsf{Key}_{N_y}\rangle\ var$ and $\gamma_{\mathsf{Data}}(b_j) = \langle B_j, \mathsf{Data}\rangle\ var$. Then $T_x \geq T_y$, $T_x \geq B_j$ and $N_x \supseteq N_y$.

Proposition 2. *A typing γ of a program P is valid iff it satisfies the constraints given above.*

Proof. Follows from the Lemmas 4 and 5 below. □

We prove the following lemmas to establish the correctness of Prop. 2. Let P be a program with the set of variables $\mathbf{Var}$ and let γ be a typing.

Lemma 3. *Let γ be a valid typing, $\gamma(x) = \langle T, U\rangle\ var$ and let $x := o(x_1, \ldots, x_k)$ be a statement of the program P. Let it be a part of the program P_1 or P_2 in the statement if b then P_1 else P_2 or while b do P_1. Then $T \geq \gamma_I(b)$.*

Proof. We first note that by the typing rules (25), (26) and (27) (and by the absence of other rules of the same kind) a program has a type according to γ only if each of its subprograms has. Let us prove the constraints for the implicit flow (from the guards of if- and $while$-statements to the variables whose assignment they control. The lemma justifies the constraints stating that the information type of a variable at the left hand side of an assignment must be equal or greater than the information type of an guard controlling the execution of that assignment.

Let $B = \gamma_I(b)$. By the order defined on $\mathcal{T}_C$ and rules (25), (26), and (27) the type of the statement $x := o(x_1, \ldots, x_k)$ must be $J\ cmd$ where $J \geq B$. This statement must be typed by either the rule (22) or (23). Both of these rules require that $\gamma(x) = \langle T, U\rangle\ var$ where $T \geq J$. The last inequality follows from the requirement that $\langle T, J, U\rangle$ must be an expression type. □

Lemma 4. *A valid typing γ satisfies the constraints given before Prop. 2.*

Proof. The constraints corresponding to the implicit flows of information are shown to hold by Lemma 3. Consider now explicit flows. Let $x := o(x_1, \ldots, x_k)$ be a statement of the program P. $x := o(x_1, \ldots, x_k)$ must be typable according to γ. If o is not a special operator then its typability must be derived from the rule (22). Let $\gamma \vdash x := o(x_1, \ldots, x_k) : J\ cmd$. Then $\gamma \vdash o(x_1, \ldots, x_k) : \langle T, J, \mathsf{Data}\rangle$ and $\gamma(x) = \langle T, \mathsf{Data}\rangle\ var$ for some $T \geq J$. From the typing rules (18) and (16) follows $\gamma \vdash x_i \langle T_i, J_i, U_i\rangle$ for some $\langle T_i, J_i, U_i\rangle \leq \langle T, J, \mathsf{Data}\rangle$ and from (21) follows $\gamma(x_i) = \langle T_i, U_i\rangle\ var$. Therefore $\langle T_i, U_i\rangle\ var \leq \langle T, \mathsf{Data}\rangle\ var$ and we have proven that the constraints hold for assignments where the operator is not a special one.

Let $x := \mathcal{E}nc(k, y)$ be a statement of P. Suppose that the constraints for the general case do not hold. In this case the statement $x := \mathcal{E}nc(k, y)$ must be typed according to the rule (22), giving to it a type $J\ cmd$ with $\gamma(x) = \langle T, \mathsf{Data}\rangle\ var$ and $\gamma \vdash \mathcal{E}nc(k, y) : \langle T, J, \mathsf{Data}\rangle$. The last claim must be inferred from the rule (20). Let $\gamma(k) = \langle T_k, \mathsf{Key}_N\rangle\ var$ and $\gamma_{\mathsf{Data}}(y) = \langle T_y, \mathsf{Data}\rangle\ var$. The constraints that we have to show follow from the rules (21), (16) and (20).

Similarly, if $x := \mathcal{G}en^i$ is a statement of P for which the general set of constraints does not hold then the special set follows from the rules (23) and (19). If $x := y$ is a statement of P where x and y are both keys and the general set

$$\dfrac{\dfrac{\dfrac{\dfrac{\gamma(x_i) = \langle T_i, U_i\rangle\ var}{\gamma \vdash x_i : \langle T_i, \emptyset, U_i\rangle}\,(21) \qquad \dfrac{\text{Constraints on }\gamma}{\langle T_i, \emptyset, U_i\rangle \leq \langle T_x, \emptyset, \mathsf{Data}\rangle}}{\gamma \vdash x_i : \langle T_x, \emptyset, \mathsf{Data}\rangle}\,(16)}{\gamma \vdash o(x_1, \ldots, x_k) : \langle T_x, \emptyset, \mathsf{Data}\rangle}\,(18) \qquad \langle T_x, \emptyset, \mathsf{Data}\rangle \leq \langle T_x, T_x, \mathsf{Data}\rangle}{\gamma \vdash o(x_1, \ldots, x_k) : \langle T_x, T_x, \mathsf{Data}\rangle}\,(16)$$

Figure 3: Inferring $\gamma \vdash o(x_1, \ldots, x_k) : \langle T_x, T_x, \mathsf{Data}\rangle$

$$\dfrac{\overline{\overline{\gamma \vdash \mathcal{G}en^i : \langle \emptyset, \emptyset, \mathsf{Key}_{\{i\}}\rangle}}\,(19) \qquad \dfrac{\text{Constraints on }\gamma}{\langle \emptyset, \emptyset, \mathsf{Key}_{\{i\}}\rangle \leq \langle T_x, T_x, U_x\rangle}}{\gamma \vdash \mathcal{G}en^i : \langle T_x, T_x, U_x\rangle}\,(16)$$

Figure 4: Inferring $\gamma \vdash \mathcal{G}en^i : \langle T_x, T_x, U_x\rangle$

of constraints does not hold then the special set follows from the rules (23) and (21). □

Lemma 5. *Let γ satisfy the constraints given before Prop. 2. Let P' be a subprogram of P (including P itself). Then $\gamma \vdash \mathsf{P}' : T\ cmd$ and T is equal or greater than $\gamma_I(b)$ for any guard b that controls whether P' is executed.*

Proof. Induction over the structure of P'. Let $b_1, \ldots, b_m$ be the guards controlling whether P' is executed; let $\gamma_{\mathsf{Data}}(b_i) = \langle B_i, \mathsf{Data}\rangle\ var$.

Suppose first that P' is a single assignment $x := e$ for some expression e. Let $\gamma(x) = \langle T_x, U_x\rangle\ var$. The constraints put on γ guarantee that $\gamma \vdash e : \langle T_x, T_x, U_x\rangle$ is inferrable. The derivation is given in Fig. 3 for general expressions and in Fig. 4 for key generations. For an expression y where y is a variable containing keys the derivation is straightforward. For encryptions the derivation is similar to general expressions. Then $\gamma \vdash x := e : T_x\ cmd$ is inferrable using either the rule (22) or (23). This, together with the constraints $T_x \geq B_j$ proves the lemma for the case where P' is a single assignment.

If P' is *skip* then $\gamma \vdash \mathsf{P}' : \top\ cmd$ is inferrable and $\top$ is no less than the information types of all guard variables controlling the execution of P'.

If P' is $\mathsf{P}'_1; \mathsf{P}'_2$ then by induction assumption $\gamma \vdash \mathsf{P}'_1 : T_1\ cmd$ and $\gamma \vdash \mathsf{P}'_2 : T_2\ cmd$ where T_1 and T_2 are equal or greater than the information types of all guard variables controlling the execution of P'_1 or P'_2. But the same guard variables also control the execution of P'. Their information types are not greater than $T_1 \wedge T_2$ and $\gamma \vdash \mathsf{P}' : T_1 \wedge T_2\ cmd$ is inferrable.

If P' is *if* b *then* P'_1 *else* P'_2 then by induction assumption $\gamma \vdash \mathsf{P}'_1 : T_1\ cmd$ and $\gamma \vdash \mathsf{P}'_2 : T_2\ cmd$ where T_1 and T_2 are equal or greater than the information types of all guard variables controlling the execution of P'_1 or P'_2. These guard variables also include b, therefore $\gamma_I(b) \leq T_1 \wedge T_2$. The rest of the guard variables is the same, therefore $B_j \leq T_1 \wedge T_2$ for $1 \leq j \leq m$. $\gamma \vdash \mathit{if}\ b\ \mathit{then}\ \mathsf{P}'_1\ \mathit{else}\ \mathsf{P}'_2 : T_1 \wedge T_2\ \mathit{cmd}$ is inferrable by rule (26). Same argument also works for while-statements. $\square$

4 Correctness of the Type System

For stating the correctness theorem we have to define which variables actually constitute the inputs of the program and which are merely used for storing the intermediate results or outputs. We say that $x \in \mathbf{Var}$ is an *input variable* if there is a path through the program where a read of x precedes the first write to x. If x is not an input variable then its initial value has no effect on the computation. Let $\mathbf{Var}_{\mathrm{I}} \subseteq \mathbf{Var}$ be the set of all input variables. The set $\mathbf{Var}_{\mathrm{I}}$ can be found using the same methods as for determining the potentially uninitialized variables in Java methods [LY99]. For a given γ let $\gamma_I(x) = T$ where $\langle T, U\rangle\ \mathit{var} = \gamma_{\mathsf{Data}}(x)$.

Theorem 6. *Let* P *be a program running in expected polynomial time with the set of variables* $\mathbf{Var}$. *Let* $\mathbf{Var}_{\mathrm{S}}$ *and* $\mathbf{Var}_{\mathrm{P}}$ *be fixed. If* P *has a valid typing* γ, *such that*

- $\gamma(x) \geq \langle\{\{\mathsf{h}\}_\emptyset\}, \mathsf{Data}\rangle\ \mathit{var}$ *for all* $x \in \mathbf{Var}_{\mathrm{S}}$,
- $\gamma(x) \geq \langle\emptyset, \mathsf{Data}\rangle\ \mathit{var}$ *for all* $x \in \mathbf{Var}_{\mathrm{I}} \cup \mathbf{Var}_{\mathrm{P}}$,
- $\bigvee_{x \in \mathbf{Var}_{\mathrm{P}}} \gamma_I(x) \not\geq \{\{\mathsf{h}\}_\emptyset\}$

then P *has secure information flow.*

We see that for applying this theorem the inputs and outputs of the program may not be keys. If we had allowed the inputs to be keys then the theorem would have had to demand that they really are valid keys. I.e. the values of corresponding variables must have been distributed indistinguishably to keys and two variables that are keys would have to be either equal or independent. In any case they would have to be independent of non-keys. We believe that the restriction that the theorem has in its current wording is not a major one. If one wants to consider keys as inputs as well, then one could just prepend the program with commands to generate those keys. If the public output of the program would have been a key then we can just assign it to a different (new) variable.

Theorem 6 directly follows from the following lemma that basically states that the public outputs of a program satisfying the premises of Thm. 6 can be computed without ever accessing the secret inputs. Using this lemma, Thm. 6 follows from some simple manipulation of probability distributions.

Lemma 7 (Simulation Lemma). *Let* P*,* $\mathbf{Var}$*,* $\mathbf{Var}_{\mathrm{S}}$*,* $\mathbf{Var}_{\mathrm{P}}$*,* $\mathbf{Var}_{\mathrm{I}}$ *satisfy the premises of Theorem 6. Then there exists a program* P' *running in expected polynomial time with the set of variables* $\mathbf{Var}'$ *and the set of input variables* $\mathbf{Var}_{\mathrm{I}}'$*, such that*

- $\mathbf{Var}_{\mathrm{S}} \cup \mathbf{Var}_{\mathrm{P}} \subseteq \mathbf{Var}'$, $\mathbf{Var}_{\mathrm{I}}' \subseteq \mathbf{Var}_{\mathrm{I}}$,
- P' *does not access the variables in* $\mathbf{Var}_{\mathrm{S}}$*,*
- *for every polynomial-time samplable family of probability distributions* D *over program states (we may assume that these states have just the variables in* $\mathbf{Var}_{\mathrm{I}} \cup \mathbf{Var}_{\mathrm{S}}$*) the families*

  $$\{\!|(S_n|_{\mathbf{Var}_{\mathrm{S}}}, S'_n|_{\mathbf{Var}_{\mathrm{P}}}) \,:\, S_n \leftarrow D_n, \langle \mathsf{P}, S_n\rangle \overset{}{\underset{n}{\Longrightarrow}} D'_n, S'_n \leftarrow D'_n|\!\} \tag{28}$$

  and

  $$\{\!|(S_n|\mathbf{Var}_{\mathrm{S}}, S'_n|_{\mathbf{Var}_{\mathrm{P}}}) \,:\, S_n \leftarrow D_n, \langle \mathsf{P}', S_n\rangle \overset{}{\underset{n}{\Longrightarrow}} D'_n, S'_n \leftarrow D'_n|\!\} \tag{29}$$

  are indistinguishable.

Proof of Theorem. 6. Let P, $\mathbf{Var}_{\mathrm{S}}$, $\mathbf{Var}_{\mathrm{P}}$, $\mathbf{Var}_{\mathrm{I}}$ be given as in the theorem. Let D be a polynomial-time samplable family of probability distributions over $\mathbf{State}$ that isolates the secrets. We have to show that (12) and (13) are indistinguishable.

According to Lemma 7 there exists a program P' not accessing the secret variables and otherwise behaving like P. Note that (12) and (28) are the same distribution. Therefore (12) and (29) are indistinguishable. The distribution (29) is just (12) for the program P'. For program P', (12) and (13) are obviously indistinguishable because D isolates the secrets and P' does not use the secrets. The distributions (13) for the program P' and (13) for the program P are again indistinguishable because of the indistinguishability of (28) and (29) — from a sample of (28) we can compute a sample of (13) for the program P by replacing the first component of the pair $(S_n|\mathbf{Var}_{\mathrm{S}}, S'_n|_{\mathbf{Var}_{\mathrm{P}}})$ with $S''_n|_{\mathbf{Var}_{\mathrm{S}}}$ where $S''_n \leftarrow D_n$ is freshly sampled. Similarly we can compute a sample of (13) for the program P' from a sample of (29). $\square$

The following sections 4.1—4.6 establish the Simulation Lemma. The program P' is constructed by transforming the program P. We give a number of basic transformations that should be the most obvious ones to consider, and then show how to put them together.

4.1 Equivalence of Programs

We are going to transform programs without (significantly) changing their functionality. We have to prove that the functionality remains the same. A common tool for these proofs is the notion of bisimulation which in our case has to be probabilistic. In the usual setting it is defined in the following way.

Let $(S, \rightarrow)$ be a *probabilistic transition system*, here S is a set of states and $\rightarrow \subseteq S \times \mathcal{D}(S)$ is the set of probabilistic transitions. For simplicity assume that $\rightarrow$ is a function, i.e. for each $s \in S$ there is exactly one $d \in \mathcal{D}(S)$ such that $s \rightarrow d$. We may also want to have final states in S, i.e. states s such that there is no $s \rightarrow d$; we model them by defining $s \rightarrow \nu(s)$. If $s \rightarrow d$ and d gives the probability p to the state s' then we also write $s \xrightarrow{p} s'$.

An *equivalence relation* $\sim$ is a *probabilistic bisimulation* on that transition system if for all $s, s' \in S$, $s \sim s'$ implies either that there are no transitions from s and s', or that $s \rightarrow d$, $s' \rightarrow d'$ and for each equivalence class $B \in S/\sim$, the distributions d and d' give the same weight to it (the weight of a subset is the sum of probabilities of all its elements). If we have a probabilistic bisimulation $\sim$ that relates the states s and s' then this means that the computations starting from states s and s' proceed in a lock-step fashion through the equivalence classes of $\sim$. Usually we are not interested in just any bisimulation $\sim$ (the total relation is a probabilistic bisimulation), but we want to have such a $\sim$ that identifies the unobservable differences and keeps significant differences apart.

Often (and particularly when we are not interested about timing) we do not want to take into account how much time a computation spends in the same equivalence class. We may want to show that computations beginning from some states s and s' proceed through the same equivalence classes but the number of steps made in the same equivalence class is not important for us. This may be the case when we want to replace a complex computation that only deals with secret data with $skip$. This notion is formalized by *weak* probabilistic bisimulations [BH97]. Let $\sim$ be an equivalence relation on S. Let $A, B \in S/\sim$ and $s \in A$. Let $\mathbf{P}(s, A, B)$ be the probability that a computation starting from s makes zero or more steps inside the set A and then makes a step to the set B. The relation $\sim$ is a *weak probabilistic bisimulation* if $\mathbf{P}(s, A, B) = \mathbf{P}(s', A, B)$ for all $A, B \in S/\sim$ and $s, s' \in A$.

The notion $\mathbf{P}(s, A, B)$ can be formalized as follows. We call a sequence $s_0 \xrightarrow{p_1} s_1 \xrightarrow{p_2} \cdots \xrightarrow{p_n} s_n$ a *run from* A *to* B if $s_0, \ldots, s_{n-1} \in A$ and $s_n \in B$. The probability of a run is the product of p_i-s on the arrows. $\mathbf{P}(s, A, B)$ is equal to the sum of probabilities of all runs from A to B that start with s.

When we are comparing two systems $(S, \rightarrow)$ and $(T, \Rightarrow)$ with starting states $s_0 \in S$ and $t_0 \in T$ then the bisimilarity of those systems means that we form the disjoint union of these systems $(S \mathbin{\dot\cup} T, \rightarrow \mathbin{\dot\cup} \Rightarrow)$ and look for a suitable (possibly weak) probabilistic bisimulation that relates s_0 and t_0.

4.2 Subject Reduction

Let P be a program. We say that a program P' is *reachable in one step* from P if there exist a state S and a distribution on states S' such that $\langle \mathsf{P}, S\rangle \longrightarrow \langle \mathsf{P}', D\rangle$. We say that P' is *reachable* from P if $\mathsf{P}' = \mathsf{P}$ or P' is reachable in one step from some program that is reachable from P.

Lemma 8. *Let* P *be a program with the set of variables* $\mathbf{Var}$ *and let* γ *be its valid typing. Let* P' *be reachable from* P*. Then* γ *is a valid typing of* P'.

Proof. It suffices to give the proof for the reachability in one step.

In this case, the proof is by induction over the structure of P. Consider the rule that establishes the reachability of P' from P in one step.

Rule (3). Then $\mathsf{P} = \mathsf{P}_1; \mathsf{P}_2$ and $\mathsf{P}' = \mathsf{P}_2$. The typability of P according to γ is inferred by the rule (25). Therefore γ is a valid typing for both P_1 and $\mathsf{P}_2 = \mathsf{P}'$.

Rule (4). Then $\mathsf{P} = \mathsf{P}_1; \mathsf{P}_2$, $\mathsf{P}' = \mathsf{P}'_1; \mathsf{P}_2$ and P'_1 is reachable in one step from P_1. Again, γ is a valid typing for both P_1 and P_2. By the induction assumption, γ is a valid typing for P'_1. The rule (25) now allows us to deduce that γ is valid typing for P'.

Rules (5) and (6). Then $\mathsf{P} = \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2$ and P' is either P_1 or P_2. The typability of P according to γ is inferred by the rule (26). Therefore γ is a valid typing for both P_1 and P_2.

Rule (7). Then $\mathsf{P} = \mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1$ and $\mathsf{P}' = \mathsf{P}_1; \mathsf{P}$. The typability of P according to γ is inferred by the rule (27). Therefore γ is a valid typing for P_1. The validness of γ for P' can now be inferred from the rule (25). □ □

4.3 Separating the Key Labels

We introduce a new kind of expressions — the $\mathcal{CE}nc$-expressions — to our programming language; these expressions allow us to ensure that each usage type occurring in the typing contains at most one key label. The expression has the following form:

$$\mathcal{CE}nc(k^{(t)} \| i_1, k_1, y_1 | i_2, k_2, y_2 | \dots | i_n, k_n, y_n), \tag{30}$$

where $i_1, \ldots, i_n \in \mathcal{G}$ and the rest are variables. The semantics of the expression compares the value $k^{(t)}$ with $i_1, \ldots, i_n$ (it is guaranteed to match one; say i_j) and returns $\mathcal{E}nc(k_j, y_j)$. Such expressions occur in the intermediate steps of transformation but not in the final program P'. The typing rule for (30) is

$$\frac{\gamma \vdash y_j : \langle T_j, J, \mathsf{Data}\rangle \quad \gamma \vdash k^{(t)} : \langle T, J, \mathsf{Data}\rangle \qquad T \leq T_j \quad \gamma \vdash k_j : \langle T_j, J, \mathsf{Key}_{\{i_j\}}\rangle}{\gamma \vdash \mathcal{CE}nc(k^{(t)} \| i'_1, k_1, y_1 | \ldots | i'_n, k_n, y_n) : \langle \{\{t\}_{N \cup \{i_j\}} \mid \{t\}_N \in T_j, 1 \leq j \leq n\}, J, \mathsf{Data}\rangle}$$

Let $\mathbf{Var}_K \subseteq \mathbf{Var}$ be the set of all variables k for which $\gamma(k) = \langle T, \mathsf{Key}_N\rangle\; var$ for some T and N; we denote that set N also by $\gamma_{\mathrm{KU}}(k)$. Let

$$\widehat{\mathbf{Var}} = \mathbf{Var} \backslash \mathbf{Var}_K \cup \{k^{(t)} \mid k \in \mathbf{Var}_K\} \cup \\ \{k^{(i)} \mid k \in \mathbf{Var}_K, \gamma(k) = \langle T, \mathsf{Key}_N\rangle\; var, i \in N\} \ .$$

We replace some assignments in P, giving us the program $\hat{\mathsf{P}}$ with the set of variables $\widehat{\mathbf{Var}}$. We formally define that $\hat{\mathsf{P}} = \mathfrak{R}(\mathsf{P})$ where $\mathfrak{R}$ is a program transformation defined in Fig. 5. We say that an encryption statement is *genuine* if the type of the variable used as a key is $\langle T, \mathsf{Key}_N\rangle\; var$ for some T and N.

The transformation "does not really change what the program does", we formalize this by giving a probabilistic bisimulation that relates the starting configurations of P and $\hat{\mathsf{P}}$ and requires the values of variables of $\mathbf{Var}_{\mathrm{P}}$ to be equal in related configurations. We have to form the disjoint union of the sets of configurations (and program states) that may be reached from the initial configurations of P and $\hat{\mathsf{P}}$. In the following we distinguish the configurations and final states stemming from P from the configurations and final states stemming from $\hat{\mathsf{P}}$ by putting a circumflex above the latter.

For program states S and $\hat{S}$ define

$$S \sim \hat{S} \Leftrightarrow \forall x \in \mathbf{Var} : S(x) = \begin{cases} \hat{S}(x), & x \notin \mathbf{Var}_K \\ \hat{S}(x^{(\hat{S}(x^{(t)}))}), & x \in \mathbf{Var}_K \ . \end{cases} \tag{40}$$

For $x \in \mathbf{Var}$ we also define the relation $\overset{x}{\sim}$ similarly to the relation $\sim$, only the quantification in (40) is over $\mathbf{Var}\backslash\{x\}$. We see that if $S \sim \hat{S}$ then $S|_{\mathbf{Var}_{\mathrm{P}}} = \hat{S}|_{\mathbf{Var}_{\mathrm{P}}}$ because $\mathbf{Var}_{\mathrm{P}} \cap \mathbf{Var}_K = \emptyset$.

Every program P has the form $\mathsf{P}_1; \mathsf{P}_2; \cdots; \mathsf{P}_k$ where P_1 is no longer a sequence of statements (other P_i may be, as we have not introduced any flattenings). Let us denote P_1 by $head(\mathsf{P})$ and $\mathsf{P}_2; \cdots; \mathsf{P}_k$ by $tail(\mathsf{P})$. Define $\mathfrak{R}(\mathsf{NIL}) = \mathsf{NIL}$.

Let $\sim$ be the smallest equivalence relation on program configurations and states reachable from initial configurations of P and $\hat{\mathsf{P}}$ that contains

- $\sim$ on states as defined by (40);

$$\mathfrak{R}(k := \mathcal{G}en^i) = [k^{(t)} := i; k^{(i)} := \mathcal{G}en^i] \ . \tag{31}$$

$$\mathfrak{R}(x := y) = [x^{(t)} := y^{(t)}; x^{(i_1)} := y^{(i_1)}; \cdots; x^{(i_n)} := y^{(i_n)}], \tag{32}$$

if $x, y \in \mathbf{Var}_K$ and $\gamma_{\mathrm{KU}}(y) = \{i_1, \ldots, i_n\}$.

$$\mathfrak{R}(x := o(x_1, \ldots, x_k)) = \left[\begin{array}{l} \mathit{if}\ x_l^{(t)} = i_1\ \mathit{then} \\ \quad \mathfrak{R}(x := o(x_1, \ldots, x_{l-1}, x_l^{(i_1)}, x_{l+1}, \ldots, x_k)) \\ \mathit{else\ if}\ x_l^{(t)} = i_2\ \mathit{then} \\ \quad \mathfrak{R}(x := o(x_1, \ldots, x_{l-1}, x_l^{(i_2)}, x_{l+1}, \ldots, x_k)) \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \mathit{else\ if}\ x_l^{(t)} = i_{n-1}\ \mathit{then} \\ \quad \mathfrak{R}(x := o(x_1, \ldots, x_{l-1}, x_l^{(i_{n-1})}, x_{l+1}, \ldots, x_k)) \\ \mathit{else} \\ \quad \mathfrak{R}(x := o(x_1, \ldots, x_{l-1}, x_l^{(i_n)}, x_{l+1}, \ldots, x_k)) \end{array}\right], \tag{33}$$

if l is the maximum number such that $x_l \in \mathbf{Var}_K$. If o is genuine encryption then we additionally demand $l \geq 2$ (i.e. x_l is not the encryption key). $\{i_1, \ldots, i_n\} = \gamma_{\mathrm{KU}}(x_l)$.

$$\mathfrak{R}(x := o(x_1, \ldots, x_k)) = [x := o(x_1, \ldots, x_k)], \tag{34}$$

if there is no such l and o is not genuine encryption.

$$\mathfrak{R}(x := \mathcal{E}nc(k, y)) = [x := \mathcal{CE}nc(k^{(t)} \| i_1, k^{(i_1)}, y | i_2, k^{(i_2)}, y | \ldots | i_n, k^{(i_n)}, y)], \tag{35}$$

where $\mathcal{E}nc$ is a genuine encryption and $\{i_1, \ldots, i_n\} = \gamma_{\mathrm{KU}}(k)$.

$$\mathfrak{R}(\mathit{skip}) = [\mathit{skip}] \ . \tag{36}$$

$$\mathfrak{R}(\mathsf{P}_1; \mathsf{P}_2) = [\mathfrak{R}(\mathsf{P}_1); \mathfrak{R}(\mathsf{P}_2)] \ . \tag{37}$$

$$\mathfrak{R}(\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2) = [\mathit{if}\ b\ \mathit{then}\ \mathfrak{R}(\mathsf{P}_1)\ \mathit{else}\ \mathfrak{R}(\mathsf{P}_2)] \ . \tag{38}$$

$$\mathfrak{R}(\mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1) = [\mathit{while}\ b\ \mathit{do}\ \mathfrak{R}(\mathsf{P}_1)] \ . \tag{39}$$

Figure 5: Transformation $\mathfrak{R}$ separating the key labels

- $\langle \mathsf{P}', S\rangle \sim \langle \mathfrak{R}(\mathsf{P}'), \hat{S}\rangle$ where P' is reachable from P and $S \sim S'$;
- if P' is reachable from P and $head(\mathsf{P}')$ is an assignment to x and x does not occur on the right hand side of $head(\mathsf{P}')$ then $\langle \mathsf{P}', S\rangle \sim \langle \mathsf{Q}; \mathfrak{R}(tail(\mathsf{P}')), \hat{S}\rangle$ where $S \overset{x}{\sim} \hat{S}$ and Q is a program from some configuration that is reachable from the configuration $\langle \mathfrak{R}(head(\mathsf{P}')), \hat{S}\rangle$. I.e. if $\mathfrak{R}(head(\mathsf{P}'))$ is defined with help of the rules (33) and (34) then the partial evaluation of $\mathfrak{R}(head(\mathsf{P}'))$ must be compatible with the values of the variables $x_l^{(t)}$.

We see that for each state S there exists a state $\hat{S}$ such that $S|_{\mathbf{Var_I}} = \hat{S}|_{\mathbf{Var_I}}$ and $\langle \mathsf{P}, S\rangle \sim \langle \hat{\mathsf{P}}, \hat{S}\rangle$.

Lemma 9. *The relation $\sim$, as defined above, is a weak probabilistic bisimulation.*

Proof. We first note the following general facts about the relation $\sim$ and about the proof we are going to give:

1. If C and C' are two configurations reachable from some initial configuration of P, and if the programs in C and C' are different, then $C \not\sim C'$.
2. It suffices to prove that if $C \sim \hat{C}$ according to the list given before the statement of the lemma (and not taking into account the extension of $\sim$ to an equivalence relation) then C and $\hat{C}$ proceed to the same equivalence classes of $\sim$ with the same probability.

The reason for the first claim is, that if $\langle \hat{\mathsf{P}}', \hat{S}\rangle$ is a configuration reachable from some initial configuration of $\hat{\mathsf{P}}$, then $\hat{\mathsf{P}}'$ has to either be $\mathfrak{R}(\mathsf{P}')$ for some P' or be achieved from some $\mathfrak{R}(\mathsf{P}')$ by running its initial fragment corresponding to the first statement of P' a number of steps. When we observe the definition of $\mathfrak{R}$ then we see that we can determine P' from $\hat{\mathsf{P}}'$.

The validity of the second claim is easy to see. If C and C' are any two configurations related by $\sim$ then there exist $C = C_0 \sim C_1 \sim \cdots \sim C_n = C'$ such that $C_i \sim C_{i+1}$ according to the list given before the statement of the lemma. We have proven that for each equivalence class B of $\sim$ the configurations C_i and C_{i+1} proceed to it with the same probability. Therefore also $C = C_0$ and $C' = C_n$ proceed to it with the same probability.

Consider now the cases enumerated above. If $S \sim \hat{S}$ then S and $\hat{S}$ are both final states for the transition system consisting of the program configurations reachable from P and $\hat{\mathsf{P}}$ and $\longrightarrow$. Therefore the probability that the execution reaches any other equivalence class from S or $\hat{S}$ is 0.

If $C = \langle \mathsf{P}', S$ where P' is reachable from P and $head(\mathsf{P}')$ is an assignment to x then in a similar configuration $\hat{C}$, the program $\hat{\mathsf{P}}'$ has the form $\mathsf{Q}; \mathfrak{R}(tail(\mathsf{P}'))$. Consider first the case where x also occurs at the right hand side of $head(\mathsf{P}')$. If

x is a key then $head(\mathsf{P}')$ must be $x := x$ which has the same effect as $skip$. The transformation $\mathfrak{R}$ turns it to a sequence of assignments, such that each of them has equal left hand side and right hand side. Therefore $\langle \mathsf{P}', S\rangle \xrightarrow{1} \langle tail(\mathsf{P}'), S\rangle$ and $\langle \mathfrak{R}(\mathsf{P}'), \hat{S}\rangle \xrightarrow{1} \cdots \xrightarrow{1} \langle \mathfrak{R}(tail(\mathsf{P}')), \hat{S}\rangle$. The resulting configurations are related by $\sim$.

If x occurs at the right hand side of the assignment $head(\mathsf{P}')$ and x is not a key then the program Q is in the middle of choosing the right values of operands. When it has finished choosing them (and has evaluated to a single assignment) then x will be updated with the same value in both P' and $\hat{\mathsf{P}}'$. This finishes the evaluation of both $head(\mathsf{P}')$ and $\mathfrak{R}(head(\mathsf{P}'))$.

The argument is the same if $head(\mathsf{P}')$ is an assignment to x and x does not occur at the right hand side of that assignment. The program states in two configurations may be related less, as the values of the variable x may be different, but this value does not influence the remaining computation. Not considering the value of x when deciding whether the configurations C and $\hat{C}$ are bisimilar is necessary when $\mathfrak{R}(head(\mathsf{P}'))$ was given by the rule (31) or (32). The relationship (40) is violated in the middle of the execution of the right hand sides of those rules.

If $head(\mathsf{P}')$ is not an assignment then the execution of P' and $\mathfrak{R}(\mathsf{P}')$ proceeds using the same rule in Fig. 1. The next step that these programs do does not change the state and results in some programs P'' and $\mathfrak{R}(\mathsf{P}'')$, i.e. the resulting configurations are bisimilar. □

Lemma 10. *The program $\hat{\mathsf{P}}$ runs in expected polynomial time.*

Proof. From the description of the transformation $\mathfrak{R}$ we see that for each step that the program P does during its execution, $\hat{\mathsf{P}}$ does at most $|\mathcal{G}| \cdot \max_{o \in \mathbf{Op}} \mathrm{arity}(o)$ steps. This quantity does not depend on n. As P runs in expected polynomial time, so does $\hat{\mathsf{P}}$. □

The program $\hat{\mathsf{P}}$ can be typed in the ways we have hinted. Define $\hat{\gamma} : \widehat{\mathbf{Var}} \rightarrow \mathcal{T}_V$ by

$$\begin{aligned}
\hat{\gamma}(x) &= \gamma(x) && \text{if } x \notin \mathbf{Var}_K \\
\hat{\gamma}(k^{(t)}) &= \langle T, \mathsf{Data}\rangle\ var && \text{if } \gamma(k) = \langle T, \mathsf{Key}_N\rangle\ var \\
\hat{\gamma}(k^{(i)}) &= \langle T, \mathsf{Key}_{\{i\}}\rangle\ var && \text{if } \gamma(k) = \langle T, \mathsf{Key}_N\rangle\ var\ .
\end{aligned}$$

We see that $\hat{\gamma}$ equals γ on secret and public variables.

Lemma 11. *The typing $\hat{\gamma}$ is valid for the program $\hat{\mathsf{P}}$.*

Proof. We show that the constraints mentioned in Prop. 2 hold. For an assignment $x := o(x_1, \ldots, x_k)$ where o is not one of the special cases we have $\hat{\gamma}(x) = \gamma(x)$.

The types of the variables at the right hand side of the assignment are not larger than the types of the variables at the right hand side of the corresponding assignment in the original program. There may be implicit flow to x from more variables than in the original program, but these extra variables are of the form $x_l^{(t)}$ for some $l \in \{1, \ldots, k\}$ and $\hat{\gamma}_I(x_l^{(t)}) \leq \hat{\gamma}_I(x_l^{(i)}) \leq \gamma_I(x_l) \leq \gamma(x)\hat{\gamma}(x)$.

For an assignment of keys $x^{(i)} := y^{(i)}$, where $\hat{\gamma}(x) = \langle T_x, \mathsf{Key}_{\{i\}}\rangle\ var$ and $\hat{\gamma}(y) = \langle T_y, \mathsf{Key}_{\{i\}}\rangle\ var$ we have $T_x \geq T_y$ because the original program had the assignment $x := y$. Similarly for the assignment of selectors $x^{(t)} := y^{(t)}$. Implicit flow to $x^{(i)}$ and $x^{(t)}$ is the same as in the original program. For the creation of a new key $k^{(t)} := i; k^{(i)} := \mathcal{G}en^i$ the argument is the same. There is only implicit flow to $k^{(t)}$ and $k^{(i)}$ and it was already accounted for the assignment $k := \mathcal{G}en^i$ in the original program.

For a $\mathcal{CE}nc$-expression we see that it gets the same type as the original $\mathcal{E}nc$-expression and the types of its arguments must be the same as the types of arguments of the original $\mathcal{E}nc$-expression. $\square$

In the rest of the correctness proof of the type system, we work with the program $\hat{\mathsf{P}}$ and typing $\hat{\gamma}$, denoting them by P and γ. In the rest we also assume without explicitly stating it that for each usage type Key_N occurring somewhere the cardinality of the set N is 1.

4.4 Removing a Basic Secret

Let $T \in \mathcal{T}_2$ be an information type. We say that some $i \in \{\mathsf{h}\} \cup \mathcal{G}$ *occurs in T as data* if $\{i\}_N \in T$ for some N. We say that $i \in \mathcal{G}$ *occurs in T as a key* if $\{t\}_{N\cup\{i\}} \in T$ for some t and N. We say that *j encrypts i in T* if $\{i\}_{N\cup\{j\}} \in T$ for some N. A key label i is *free* in T if occurs in T only as a key.

Let $j \in \{\mathsf{h}\} \cup \mathcal{G}$. Let $\mathbf{Var}_j$ be the set of all variables x, such that j occurs as data in $\gamma(x)$. Let $\overline{\mathbf{Var}_j} = \mathbf{Var}\backslash\mathbf{Var}_j$. Then the values of the variables in $\overline{\mathbf{Var}_j}$ can be computed without accessing the variables in $\mathbf{Var}_j$. This is achieved by running the program P but ignoring the statements that access the variables in $\mathbf{Var}_j$ in some way. We formalize this with help of the program transformation $\mathfrak{D}_j$ given in Fig. 6.

Let $\hat{\mathsf{P}} = \mathfrak{D}_j(\mathsf{P})$; in referring to the configurations of P and $\hat{\mathsf{P}}$ we use the same conventions as before.

Recall that we defined the relation $\Longrightarrow$ from program configurations to $\mathcal{D}(\mathbf{State}_\perp)$. The set $\mathbf{State}_\perp$ is partially ordered, $\perp$ is the smallest element and all other elements are incomparable. If X is a partially ordered set then we can define a partial order also on $\mathcal{D}(X)$. If $D_1, D_2 \in \mathcal{D}(X)$ then we define $D_1 \leq D_2$ if for all upper closed sets $Y \subseteq X$ (i.e. $x \in Y$ and $y \geq x$ implies $y \in Y$) $\sum_{x\in Y} D_1(x) \leq \sum_{x\in Y} D_2(x)$. I.e. D_2 has to put at least as much weight on

$$\mathfrak{D}_j(x := y) = \begin{cases} x := y, & x \notin \mathbf{Var}_j \\ skip, & \text{otherwise} \end{cases} \tag{41}$$

$$\mathfrak{D}_j(x := o(x_1, \ldots, x_k)) = \begin{cases} x := o(x_1, \ldots, x_k), & x \notin \mathbf{Var}_j \\ skip, & \text{otherwise} \end{cases} \tag{42}$$

$$\mathfrak{D}_j(skip) = skip \tag{43}$$

$$\mathfrak{D}_j(\mathsf{P}_1; \mathsf{P}_2) = \mathfrak{D}_j(\mathsf{P}_1); \mathfrak{D}_j(\mathsf{P}_2) \tag{44}$$

$$\mathfrak{D}_j(\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2) = \begin{cases} \mathit{if}\ b\ \mathit{then}\ \mathfrak{D}_j(\mathsf{P}_1)\ \mathit{else}\ \mathfrak{D}_j(\mathsf{P}_2), & b \notin \mathbf{Var}_j \\ skip, & b \in \mathbf{Var}_j \end{cases} \tag{45}$$

$$\mathfrak{D}_j(\mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1) = \begin{cases} \mathit{while}\ b\ \mathit{do}\ \mathfrak{D}_j(\mathsf{P}_1), & b \notin \mathbf{Var}_j \\ skip, & b \in \mathbf{Var}_j \end{cases} \tag{46}$$

Figure 6: Transformation $\mathfrak{D}_j$ deleting the access to secrets of type j

the larger elements of X as D_1. If $D_1, D_2 \in \mathcal{D}(\mathbf{State}_\perp)$ then $D_1 \leq D_2$ means $D_1(S) \leq D_2(S)$ for all $S \in \mathbf{State}$. If P_1 and P_2 are such that for all initial states S and all final states S' we have $D_1(S') \leq D_2(S')$, where $\langle \mathsf{P}_1, S \rangle \Longrightarrow D_1$ and $\langle \mathsf{P}_2, S \rangle \Longrightarrow D_2$ then P_1 approximates P_2 in the sense that if P_2 produces some output then P_1 either produces the same output or fails to terminate. If P_1 and P_2 both run in expected polynomial time then these two programs are indistinguishable.

For some $S_P : \overline{\mathbf{Var}_j} \to \mathbf{Val}$ and $D \in \mathbf{State}_\perp$ let $D[S_P]$ denote the sum $\sum_{S \in \mathbf{S}} D(S)$ where $\mathbf{S}$ is the set of all such program states that agree with S_P on $\overline{\mathbf{Var}_j}$.

Lemma 12. *Let $S \in \mathbf{State}$. Let $\langle \mathsf{P}, S \rangle \Longrightarrow D$ and $\langle \hat{\mathsf{P}}, S \rangle \Longrightarrow \hat{D}$. Let $S_P : \overline{\mathbf{Var}_j} \to \mathbf{Val}$. Then $D[S_P] \leq \hat{D}[S_P]$.*

Proof. By induction on the program structure. We prove a bit stronger statement, namely that if $\langle \mathsf{P}, S \rangle \Longrightarrow D$ and $\langle \hat{\mathsf{P}}, \hat{S} \rangle \Longrightarrow \hat{D}$ where $S|_{\overline{\mathbf{Var}_j}} = \hat{S}|_{\overline{\mathbf{Var}_j}}$ then $D[S_P] \leq \hat{D}[S_P]$.

If P is a single assignment then in $\mathfrak{D}_j(\mathsf{P})$ the value of x is either computed the same way as in P (if $x \notin \mathbf{Var}$ or not computed at all, depending on whether $x \in \mathbf{Var}_j$ or not. If $x \notin \mathbf{Var}_j$ and the value of x is computed then none of the arguments of o can be in $\mathbf{Var}_j$ by the properties of the type system. Therefore the values of the arguments of o agree in S and $\hat{S}$.

The case for *skip* is trivial. If P is $\mathsf{P}_1; \mathsf{P}_2$ then $\langle \mathsf{P}_1, S \rangle \Longrightarrow D_1$ and $\langle \mathfrak{D}_j(\mathsf{P}_1), \hat{S} \rangle \Longrightarrow \hat{D}_1$ where $D_1[S_P] \leq \hat{D}_1[S_P]$ for any $S_P : \overline{\mathbf{Var}_j} \to \mathbf{Val}$. If we run P_2 and $\mathfrak{D}_j(\mathsf{P}_2)$

from any states S' and $\hat{S}'$ that agree on $\overline{\mathbf{Var}_j}$ then we get the resulting distributions D and $\hat{D}$ where $D[S_P] \leq \hat{D}[S_P]$ for any S_P. The quantities $D[S_P]$ and $\hat{D}[S_P]$ may be somewhat smaller if P_2 and $\mathfrak{D}_j(\mathsf{P}_2)$ are run after P_1 and $\mathfrak{D}_k(\mathsf{P}_1)$, respectively, but the diminishing cannot be greater for $\hat{D}$ than for D because $\mathfrak{D}_j(\mathsf{P}_1)$ terminates no less often than P_1.

If the guard variable in an if- or while-statement is in $\overline{\mathbf{Var}_j}$ then the claim for the whole program directly follows from the claim(s) for the branches or loop body. The transformed program may terminate more often than the original because the branches or loop body may terminate more often, but otherwise the final state is the same for the variables in $\overline{\mathbf{Var}_j}$. If the guard variable is in $\mathbf{Var}_j$ then the left hand sides of the assignments in the branches or in the loop body can only be to variables in $\mathbf{Var}_j$ as well. This follows from the type system. Changing the whole program to skip thus has no effect on the values of variables in $\overline{\mathbf{Var}_j}$ (except that termination may occur more often). □

Lemma 13. *The program $\hat{\mathsf{P}}$ runs in expected polynomial time.*

Proof. From the description of the transformation $\mathfrak{D}_j$ we see that $\hat{\mathsf{P}}$ does at most one step for each step of P during the execution. As P runs in expected polynomial time, so does $\hat{\mathsf{P}}$. □

Lemma 14. *The program $\hat{\mathsf{P}}$ does not access the variables in $\mathbf{Var}_j$.*

Proof. An access is either a read or a write. The cases where a variable in $\mathbf{Var}_j$ is assigned to, or is read because it is the guard variable of some if- or while-statement are made explicit in Fig. 6. If a variable in $\mathbf{Var}_j$ occurs at the right hand side of an assignment then the left hand side must belong to $\mathbf{Var}_j$ as well, because of the properties of the type system. Therefore this assignment is replaced with skip. □

If the program P and the typing γ that are given to us are such, that h does not occur in $\bigvee_{x\in\mathbf{Var}_\mathsf{P}} \gamma_I(x)$ (as data) then $\mathfrak{D}_\mathsf{h}(\mathsf{P})$ is the program P' whose existence is claimed by the Simulation Lemma. For $j \in \mathcal{G}$ the transformation $\mathfrak{D}_j$ is used in the next section.

4.5 Replacing the Keys and Plaintexts in Encryptions

Before starting the replacing we prepend P with $k_\mathfrak{n} := \mathcal{G}en^\mathfrak{n}$ where $k_\mathfrak{n}$ is a new variable and $\mathfrak{n}$ a new key label. We also update γ with $\gamma(k_\mathfrak{n}) = \langle\emptyset, \mathsf{Key}_{\{\mathfrak{n}\}}\rangle\ \mathit{var}$.

The replacements in encryptions are done iteratively, one key label at a time. Assume now that $\mathcal{P}$ is a program that has been obtained as the result of some iteration and is now considered for the next iteration. Let γ be the corresponding typing. Then P and γ obey the following invariants:

A. $k_{\mathfrak{n}}$ is used (besides assigning a key to it at the beginning of the program) only for encrypting some fixed constant $\mathbf{0}$ (either in $\mathcal{E}nc$- or in $\mathcal{CE}nc$-expressions).

B. $\mathfrak{n}$ does not occur or is free in $\bigvee_{x\in\mathbf{Var}_\mathsf{P}} \gamma_I(x)$.

C. If $\{t\}_N \in \bigvee_{x\in\mathbf{Var}_\mathsf{P}} \gamma_I(x)$ for some t and N, where $\mathfrak{n} \in N$ then also $\{t\}_{N\setminus\{\mathfrak{n}\}\cup\{i\}} \in \bigvee_{x\in\mathbf{Var}_\mathsf{P}} \gamma_I(x)$ for some $i \notin N \cup \{\mathfrak{n}\}$.

Let us now state, what it means that a key is only used as an encryption key and the encryptions are therefore amenable to be changed to $\mathcal{E}nc(k^{\mathfrak{n}}, \mathbf{0})$. Let $\mathcal{O}$ be a stateful algorithm (oracle) that takes two bit-strings r and y as arguments. In the beginning, the state of $\mathcal{O}$ is empty. If $\mathcal{O}$ is queried with r and y and r has not occurred as the first argument before, then it generates a new key $\mathbf{k}_r$ and stores $(r, \mathbf{k}_r)$. Otherwise it retrieves the key $\mathbf{k}_r$. It then returns the encryption of y under $\mathbf{k}_r$. So the keys inside $\mathcal{O}$ can be only used as encryption keys.

Let us add one more construct to our programming language; actually we extend an existing construction. To the expression $\mathcal{CE}nc(k^{(t)} \| \cdots |i_l, k^{(l)}, y_l| \cdots)$ we introduce the possibility to underline certain triples $|i_l, k^{(l)}, y_l|$. Semantically, underlining means that if $k^{(t)}$ equals i_l then we do not compute the encryption of y_l under $k^{(l)}$, but invoke $\mathcal{O}$ with the arguments $k^{(l)}$ and y_l. We do not need to specify how to type the new construct. Also, let $\underline{\mathcal{E}nc}$ be a binary operator in the programming language whose semantics is to invoke $\mathcal{O}$.

Considering the algorithm $\mathcal{O}$ we may assume that $\mathcal{O}$ does not generate a new key each time it receives a new r, but the keys $\mathbf{k}_r$ for all possible r have been generated beforehand (independently of each other) and $\mathcal{O}$ just picks the one it needs. Let $j \in \mathcal{G}\setminus\{\mathfrak{n}\}$ be a key label that does not occur as data in any $\gamma(x)$ for $x \in \mathbf{Var}$. Let $\bar{\mathsf{P}} = \mathfrak{E}'_j(\mathsf{P})$ be the program obtained from P by underlining all triples $|j, k, y|$ and $|\mathfrak{n}, k, y|$ in $\mathcal{CE}nc$-expressions, and by replacing all expressions $\mathcal{E}nc(k_{\mathfrak{n}}, \mathbf{0})$ with $\underline{\mathcal{E}nc}(k_{\mathfrak{n}}, \mathbf{0})$.

We show that P and $\bar{\mathsf{P}}$ are bisimilar with respect to a probabilistic bisimulation that changes only the values of the keys with labels $\mathfrak{n}$ and j. To distinguish the configurations and states reachable from P from those reachable from $\bar{\mathsf{P}}$ we always put a horizontal bar over the latter. Let $\sim$ be the smallest equivalence relation containing the following:

- $S \sim \bar{S}$ if $S(x) = \mathbf{k}_{\bar{S}(x)}$ for all x with usage types $\mathsf{Key}_{\{j\}}$ and $\mathsf{Key}_{\{\mathfrak{n}\}}$, and $S(x) = \bar{S}(x)$ for all other variables.

- $\langle \mathsf{P}', S\rangle \sim \langle \mathfrak{E}'_j(\mathsf{P}'), \bar{S}\rangle$ if P' is a program reachable from P and $S \sim \bar{S}$.

Lemma 15. *If j and $\mathfrak{n}$ do not occur as data in any $\gamma(x)$ for $x \in \mathbf{Var}$ then $\sim$ is a probabilistic bisimulation.*

Proof. Consider the statement $head(\mathsf{P}')$. If it is not an assignment then the configurations $\langle \mathsf{P}', S\rangle$ and $\langle \mathfrak{E}'_j(\mathsf{P}'), \bar{S}\rangle$ will make the next step to configurations $\langle \mathsf{P}'', S\rangle$ and $\langle \mathfrak{E}'_j(\mathsf{P}''), \bar{S}\rangle$, respectively, for some program P'' and these two configurations are bisimilar.

Assume now that $head(\mathsf{P}')$ is an assignment. If it is $x := o(x_1, \ldots, x_k)$ for some non-special o (or for $o \equiv \mathcal{E}nc$ where we consider only the second argument) then j and $\mathfrak{n}$ may not occur in the usage types of $x_1, \ldots, x_k$ otherwise they would occur as data in x. Therefore S and $\bar{S}$ agree on $x_1, \ldots, x_k$. If $head(\mathsf{P}')$ is $x := y$ where the usage types of both x and y are $\mathsf{Key}_{\{j\}}$ then $S(y) = \mathbf{k}_{\bar{S}(y)}$ and therefore after the assignment $S(x) = \mathbf{k}_{\bar{S}(x)}$, as required by the definition of $\sim$. If $head(\mathsf{P}')$ is an $\mathcal{E}nc$- or $\mathcal{C}\mathcal{E}nc$-assignment then the actual value of the key that we are using is the same in configurations $\langle \mathsf{P}', S\rangle$ and $\langle \mathfrak{E}'_j(\mathsf{P}'), \bar{S}\rangle$. Therefore the value of the encryption will also be the same. For a statement $k^{(j)} := \mathcal{G}en^j$, the distributions $[\![\mathcal{G}en]\!](1^n)$ and $\mathbf{k}_{[\![\mathcal{G}en]\!](1^n)}$ are identical — both are correctly generated keys. $\square$

We have now isolated the uses of the keys labeled with j (and $\mathfrak{n}$). We can now do the replacement allowed by (9), we do this inside $\mathcal{O}$. Let $\mathcal{O}'$ be the following stateful oracle. In the beginning, $\mathcal{O}'$ generates a key $\mathbf{k}$. Whenever $\mathcal{O}'$ is queried with two arguments, it discards these arguments and returns a freshly created encryption of some constant bit-string $\mathbf{0}$ under the key $\mathbf{k}$.

Lemma 16. *The oracles $\mathcal{O}$ and $\mathcal{O}'$ are computationally indistinguishable, i.e. for all PPT algorithms $\mathcal{A}$ the difference of probabilities*

$$\mathbf{P}[\mathcal{A}^{\mathcal{O}(1^n,\cdot,\cdot)}(1^n) = 1] - \mathbf{P}[\mathcal{A}^{\mathcal{O}'(1^n,\cdot,\cdot)}(1^n) = 1] \tag{47}$$

is negligible in n.

Proof. By a standard hybrid argument. Assume the opposite, i.e. that there exists a PPT algorithm $\mathcal{A}$ for which the difference (47) is non-negligible. Let p_n denote the probability that $\mathcal{A}(1^n)$ returns 1 when given $\mathcal{O}$, and p'_n denote the probability that $\mathcal{A}(1^n)$ returns 1 when given $\mathcal{O}'$. The difference between p_n and p'_n is not negligible.

Let $\mathcal{O}_i$ be an oracle that takes two arguments r and y. It works as follows. In the beginning it generates a key $\mathbf{k}$ and sets the variable c to be equal to 0. The variable c will record how many different arguments r have been submitted to $\mathcal{O}_i$ so far. On input (r, y) the oracle $\mathcal{O}_i$ checks whether r has been an argument to it before. If not and if $c \geq i$ then it defines $\mathbf{k}_r := \mathbf{k}$ and $\mathbf{a}_r = 0$. If $c < i$ then it generates a new key $\mathbf{k}_r$, defines $\mathbf{a}_r = 1$ and increments c. In both cases the triple $(r, \mathbf{k}_r, \mathbf{a}_r)$ is recorded. If $\mathcal{O}_i$ has seen r before then it just retrieves that triple. It then returns an encryption of $\mathbf{a}_r \cdot y$ (i.e. either $\mathbf{0}$ or y) under the key $\mathbf{k}_r$. The oracles $\mathcal{O}_i$ are our hybrids — they are something with the functionality inbetween $\mathcal{O}$ and

$\mathcal{O}'$. Also, $\mathcal{O}_0 = \mathcal{O}'$. Let $p_n^{(i)}$ denote the probability that $\mathcal{A}(1^n)$ returns 1 when given the oracle $\mathcal{O}_i$; then $p_n^{(0)} = p'_n$.

Let $\mathcal{Q}^{(\cdot),(\cdot)}(1^n, i, \cdot, \cdot)$ be the following algorithm with two oracles. We denote its oracle arguments by f and g and its bit-string arguments by r and y. In it sets the variable c equal to 0. On input (r, y) it checks whether r has been an argument to it before. If not then

- If $c < i$ then a new key $\mathbf{k}_r$ is generated and $(r, \underline{\texttt{Key}}\ \mathbf{k}_r)$ is recorded.
- If $c = i$ then $(r, \underline{\texttt{Left}})$ is recorded.
- If $c \geq i + 1$ then $(r, \underline{\texttt{Right}})$ is recorded.

In all cases c is incremented. If r has been an argument to $\mathcal{Q}$ before then the corresponding pair is retrieved. A value x is then computed and returned, this value depends on the second component of the pair. If it is

- $\underline{\texttt{Key}}$ $\mathbf{k}$ then x is the encryption of y under the key $\mathbf{k}$;
- $\underline{\texttt{Left}}$ then x is the result of $f(y)$;
- $\underline{\texttt{Right}}$ then x is the result of $g(\mathbf{0})$.

Note that

$$\mathcal{Q}^{\mathcal{E}(1^n,k,\cdot),\mathcal{E}(1^n,k',\cdot)}(1^n, i, \cdot, cdot) = \mathcal{O}_{i+1}(1^n, \cdot, \cdot) \tag{48}$$

and

$$\mathcal{Q}^{\mathcal{E}(1^n,k,\mathbf{0}),\mathcal{E}(1^n,k,\mathbf{0})}(1^n, i, \cdot, cdot) = \mathcal{O}_i(1^n, \cdot, \cdot) \tag{49}$$

where k (and k') are (independent) encryption keys.

Let $\mathcal{B}^{(\cdot),(\cdot)}(1^n)$ now be the following algorithm, where we denote the oracle arguments of $\mathcal{B}$ be f and g. Let q be a polynomial that is upper bound on the running time of $\mathcal{A}$. Then $\mathcal{A}^{(\cdot)}(1^n)$ can make at most $q(n)$ different queries to its oracle and the results of running $\mathcal{A}$ with the oracle $\mathcal{O}$ or $\mathcal{O}_{q(n)}$ are identical, i.e. $p_n^{(q(n))} = p_n$. $\mathcal{B}$ first generates a random number i between 0 and $q(n) - 1$ and then runs $\mathcal{A}^{\mathcal{Q}^{f,g}(1^n,i,\cdot,\cdot)}(1^n)$.

The probability that $\mathcal{B}$ returns 1 when given the oracles $\mathcal{E}(1^n, k, \cdot)$ and $\mathcal{E}(1^n, k', \cdot)$ is $\frac{1}{q(n)} \sum_{i=1}^{q(n)} p_n^{(i)}$. The probability that $\mathcal{B}$ returns 1 when given the oracles $\mathcal{E}(1^n, k, \mathbf{0})$ and $\mathcal{E}(1^n, k, \mathbf{0})$ is $\frac{1}{q(n)} \sum_{i=0}^{q(n)-1} p_n^{(i)}$. Their difference is $\frac{1}{q(n)}(p_n^{(q(n))} - p_n^{(0)}) = \frac{1}{q(n)}(p_n - p'_n)$ which is not negligible. Therefore if we put $\mathcal{B}$ in place of $\mathcal{A}$ in (9) then the difference is not negligible, i.e. the encryption system is not secure. □

I.e. if we change the semantics of underlined triples $|\underline{i,k,y}|$ in $\mathcal{CE}nc$-expressions from invoking $\mathcal{O}$ to invoking $\mathcal{O}'$ then the values of variables do not change in any way that is distinguishable to PPT algorithms. Assume now the latter semantics of underlined triples. Let $\mathfrak{E}''_j$ be a program transformation that replaces all underlined triples in $\mathcal{CE}nc$-expressions with $|\mathfrak{n}, k_\mathfrak{n}, \mathbf{0}|$ and removes the underlining. If this creates a $\mathcal{CE}nc$-expression where the second and third components of all triples are $k_\mathfrak{n}$ and $\mathbf{0}$ respectively then the entire expression is replaced with $\mathcal{E}nc(k_\mathfrak{n}, \mathbf{0})$. Let $\hat{\mathsf{P}} = \mathcal{E}''(\bar{\mathsf{P}})$. We show that $\bar{\mathsf{P}}$ and $\hat{\mathsf{P}}$ are bisimilar with respect to a probabilistic bisimulation that only changes the values of the keys with labels $\mathfrak{n}$ and j. Let $\mathbf{k}$ be the key generated by $\mathcal{O}'$ in the beginning of the run and let $\sim$ be the smallest equivalence relation containing the following:

- $\bar{S} \sim \hat{S}$ if $\bar{S}(x) = \mathbf{k}$ for all x with usage type and $\mathsf{Key}_{\{\mathfrak{n}\}}$, and $\bar{S}(x) = \hat{S}(x)$ for all other variables with usage type other than $\mathsf{Key}_{\{j\}}$. There is no restriction on the value of the variables with usage type $\mathsf{Key}_{\{j\}}$.

- $\langle \mathsf{P}', \bar{S}\rangle \sim \langle \mathfrak{E}''_j(\mathsf{P}'), \hat{S}\rangle$ if P' is a program reachable from $\bar{\mathsf{P}}$ and $\bar{S} \sim \hat{S}$.

Lemma 17. *If j and $\mathfrak{n}$ do not occur as data in any $\gamma(x)$ for $x \in \mathbf{Var}$ then $\sim$ is a probabilistic bisimulation.*

Proof. Similar to lemma 15. The differences are in the handling of $\mathcal{CE}nc$- and $\mathcal{E}nc$-expressions, but they are not major. To a $\mathcal{CE}nc$-expression in P' may correspond an $\mathcal{E}nc$-expression in $\mathfrak{E}''_j(\mathsf{P}')$ but in this case all keys and plaintexts in the $\mathcal{CE}nc$-expression had the same value anyway. ☐

To summarize, if j and $\mathfrak{n}$ do not occur as data in any $\gamma(x)$ for $x \in \mathbf{Var}$ then the values of variables other than the keys labelled with j or $\mathfrak{n}$ are indistinguishable in programs P and $\hat{\mathsf{P}} = \mathfrak{E}''_j(\mathfrak{E}'_j(\mathsf{P}))$. The key $k_\mathfrak{n}$ is used in $\hat{\mathsf{P}}$ only for encrypting the constant $\mathbf{0}$, i.e. the invariant A holds. We define the typing $\hat{\gamma}$ as follows: Let $x \in \mathbf{Var}$ and $\gamma(x) = \langle T_x, U_x\rangle\ var$. We now define a set $\hat{T}_x \in \mathcal{T}_2$ in the following way. Let $\{t\}_N \in T_x$. It then causes the following elements of $\mathcal{T}_1$ to be in $\hat{T}_x$:

- If $j \notin N$ then $\{t\}_N \in \hat{T}_x$.

- If $j \in N$ and there exists some $i \in \mathcal{G}\backslash(N \cup \{\mathfrak{n}\})$, such that $\{t\}_{N\backslash\{j\}\cup\{i\}} \in T_x$ then $\{t\}_{N\backslash\{j\}\cup\{\mathfrak{n}\}} \in \hat{T}_x$.

- If neither of the above cases holds then no element is added to $\hat{T}_x$.

We let $\hat{\gamma}(x) = \langle \hat{T}_x, U_x\rangle\ var$. From the description of $\hat{\gamma}$ it is obvious that it satisfies the invariants B and C.

Lemma 18. *$\hat{\gamma}$ is a valid typing of $\hat{\mathsf{P}}$.*

Proof. In the program we basically replaced encryptions with keys labelled by j with the encryptions with the key labelled by $\mathfrak{n}$. So, as long as we just replace j by $\mathfrak{n}$ in all information types the typing remains valid. But sometimes we just delete an element $\{t\}_{N\cup\{j\}}$ from the type of x.

Let x be such a variable and assume that it was wrong to leave $\{t\}_{N\cup\{\mathfrak{n}\}}$ out of the type of x, i.e. $\hat{\gamma}$ is not a valid typing because of that. Then the information from some $\mathcal{CE}nc$-expression must flow to x and $k_{\mathfrak{n}}$ must occur in that $\mathcal{CE}nc$-expression. Indeed, $k_{\mathfrak{n}}$ may occur only in $\mathcal{E}nc$- and $\mathcal{CE}nc$-expressions and in $\mathcal{E}nc$-expressions it may encrypt only the constant $\mathbf{0}$ which gives no elements of $\mathcal{T}_1$ to the types. In $\mathcal{CE}nc$-expressions the occurrence of $k_{\mathfrak{n}}$ also gives the type of the selector variable $k^{(t)}$, encrypted with $\mathfrak{n}$. Thus the selector variable of that $\mathcal{CE}nc$-expression must have had $\{t\}_N$ occurring in its information type. Because it is a $\mathcal{CE}nc$-expression, keys with some other label i must also occur in it. But then $\{t\}_{N\cup\{i\}}$ also occurs in the type of x, it also occurred in the original type of x and we should not have just deleted $\{t\}_{N\cup\{j\}}$ from the type of x. □

Let $\mathfrak{E}_j$ denote the composition of transformations $\mathfrak{E}''_j \circ \mathfrak{E}'_j$.

4.6 Putting the Transformations Together

The building of the program P' that does not access secret variables from the program P, such that the distributions (28) and (29) are indistinguishable (as required by the Simulation lemma), consists of invoking the transformations described so far in a certain order. We invoke them in the following way.

— $\boxed{1}$ Invoke the transformation $\mathfrak{R}$. After that transformation each usage type is either Data or contains at most one key label.

— $\boxed{2}$ Prepend $k_{\mathfrak{n}} := \mathcal{G}en^{\mathfrak{n}}$ to the program, add $\mathfrak{n}$ to $\mathcal{G}$ and $\{k_{\mathfrak{n}} \mapsto \langle\emptyset, \mathsf{Key}_{\{\mathfrak{n}\}}\rangle\ var$ to γ.

— $\boxed{3}$ Consider the information type $T_P = \bigvee_{x\in\mathbf{Var}_\mathsf{P}} \gamma_I(x)$. If h occurs in it then go to $\boxed{4}$ else go to $\boxed{7}$.

— $\boxed{4}$ h occurs in T_P, hence some key label must occur as a key in this type (some key must certainly encrypt h). By invariant C, if we delete all elements of the form $\{t\}_{N\cup\{\mathfrak{n}\}}$ from T_P, giving us T'_P, some key labels still occur there as keys. By [Lau02, Thm. 1] some key label j is free in T'_P. It is also free in T_P.

— $\boxed{5}$ Invoke the transformation $\mathfrak{D}_j$. After its invocation j does not occur as data in any $\gamma(x)$.

— $\boxed{6}$ The premises of Lemmas 15 and 17 are satisfied. Invoke the transformation $\mathfrak{E}_j$. After that transformation j no longer occurs in T_P as a key — the number of key labels occurring in T_P as a key has decreased. Go to $\boxed{3}$.

— $\boxed{7}$ Invoke the transformation $\mathfrak{D}_{\mathsf{h}}$. This deletes all accesses to secret variables from the program.

The steps presented here either do not change the distribution (29) (where P is the original program and P' the current program) at all or change it indistinguishably. The number of steps is bounded by a constant (it is $O(|\mathcal{G}|)$ which does not depend on n) therefore the initial and final distributions are indistinguishable.

5 Conclusions and future work

We have presented a type system for computationally secure information flow that should be simple enough to be integrated into existing programming languages and used by software engineers.

The presented type system could definitely be developed further. It could be used for programs containing procedures; the existing data flow analyses [Lau03] cannot cope with them. Extending the programming language with procedures probably requires some form of key label polymorphism.

It is also important to get rid of the constraint that two keys generated at the same program point cannot be distinguished. Here some form of key relabeling during the program run could be useful.

References

[AB03] Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.

[Aba99] Martín Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[AG99] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, January 1999.

[AJ01] Martín Abadi and Jan Jürjens. Formal Eavesdropping and Its Computational Interpretation. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, volume 2215 of *LNCS*, pages 82–94, Sendai, Japan, September 2001. Springer-Verlag.

[AR00] Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 3–22, Sendai, Japan, August 2000. Springer-Verlag.

[BH97] Christel Baier and Holger Hermanns. Weak Bisimulation for Fully Probabilistic Processes. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference*, volume 1254 of *LNCS*, pages 119–130, Haifa, Israel, June 1997. Springer-Verlag.

[BP04] Michael Backes and Birgit Pfitzmann. Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library. In *17th IEEE Computer Security Foundations Workshop*, Pacific Grove, CA, June 2004.

[Can01] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, October 2001. IEEE Computer Society Press.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

[Lau01] Peeter Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *LNCS*, pages 77–91, Genova, Italy, April 2001. Springer-Verlag.

[Lau02] Peeter Laud. Encryption Cycles and Two Views of Cryptography. In Simone Fischer-Hübner and Erland Jonsson, editors, *NordSec 2002, Proceedings of the 7th Nordic Workshop on Secure IT Systems*, number 31 in Karlstad University Studies, pages 85–100, Karlstad, Sweden, November 2002.

[Lau03] Peeter Laud. Handling Encryption in Analyses for Secure Information Flow. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, pages 159–173, Warsaw, Poland, April 2003. Springer-Verlag.

[LMMS98] Patrick Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. A Probabilistic Poly-Time Framework for Protocol Analysis. In *CCS '98, Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 112–121, San Francisco, California, November 1998. ACM Press.

[LMMS99] Patrick Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. Probabilistic Polynomial-Time Equivalence and Security Analysis. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 776–793, Toulouse, France, September 1999. Springer-Verlag.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[Mye99] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press.

[PW01] Birgit Pfitzmann and Michael Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *2001 IEEE Symposium on Security and Privacy*, pages 184–200, Oakland, California, May 2001. IEEE Computer Society Press.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SV98] Geoffrey Smith and Dennis M. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998. ACM Press.

[Vol00] Dennis M. Volpano. Secure Introduction of One-way Functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 246–254, Cambridge, England, July 2000. IEEE Computer Society Press.

[VS97] Dennis M. Volpano and Geoffrey Smith. Eliminating Covert Flows with Minimum Typings. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 156–169, Rockport, Massachusetts, June 1997. IEEE Computer Society.

[VS98] Dennis M. Volpano and Geoffrey Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 34–43, Rockport, Massachusetts, June 1998. IEEE Computer Society.

[VS00] Dennis M. Volpano and Geoffrey Smith. Verifying Secrets and Relative Secrecy. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–276, Boston, Massachusetts, January 2000. ACM Press.

[VSI96] Dennis M. Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[Yao82] Andrew C. Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, Chicago, Illinois, November 1982. IEEE Computer Society Press.