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 static analysis for certification of SIF was pioneered by Denning and Denning [8, 9]. The correctness of the analysis was not proven directly from the semantics of the program, though. Volpano et al. [23] 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 [21, 25] 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 [26, 22]. 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 [18]. A good overview of research on language-based information flow security is given by Sabelfeld and Myers [19].

* Partially supported by Estonian Science Foundation, grant #6095

Type systems for handling cryptographic operations were first investigated by Abadi [1]; a more recent paper is [2]. 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 [10] while ours is correct with respect to complexity-theoretic definitions of security [27].

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 [7, 20] 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 [6] has some restrictions. Lincoln et al. [15, 16] have given probabilistic semantics to spi-calculus [3], 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. [5, 4] have given an automatic means to decide when certain reasonable computational interpretations of two messages in Dolev-Yao model are indistinguishable. We [11, 13] 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. The 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:

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

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, where $\mathbf{Val} = \mathsf{true} \,\dot\cup\, \mathsf{false}$ is a fixed partition. Note that $\longrightarrow$ is a function.

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

$$\langle \mathit{skip}, S\rangle \longrightarrow \eta(S) \tag{2}$$

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

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

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

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

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

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

Fig. 1. The operational semantics of programs

For defining security we also have to state what the outcome of the program is. A *program run* is a sequence $C_0 \stackrel{p_1}{\to} C_1 \stackrel{p_2}{\to} \cdots \stackrel{p_n}{\to} C_n$ where $C_0,\ldots,C_{n-1}$ are program configurations and C_n is a program state. If $C_{i-1} = \langle \mathsf{P}_{i-1}, S_{i-1}\rangle \longrightarrow \langle \mathsf{P}_i, D_i\rangle$ 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 $\mathbf{State}_\perp = \mathbf{State} \,\dot\cup\, \{\perp\}$ 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 [5] 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 [24]. 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

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

and

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

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.

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 types for 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 possibility to decrypt information with keys in N. They are defined as the least sets satisfying

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

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 [5] for a more thorough discussion on them). This "Dolev-Yao attacker with the ability to break encryption cycles" first appeared in [12].

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 $\leq \cap \geq$).*

See the full version [14] for a proof. 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 $\mathcal{U} = \{\mathsf{Data}\} \cup \{\mathsf{Key}_N \,|\, N \subseteq \mathcal{G}\}$. 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

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

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 $\mathbf{Var}$ to $\mathcal{T}_V$. Compared to the type system of Volpano et al. [23] 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 $\mathcal{T}_E$:

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

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. $\top = \{\{t\}_\emptyset \,|\, t \in \mathcal{T}_0\}$. 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.

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.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Typing rules

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, we have the following alternative set of constraints. 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 $N_x \supseteq N_y$.

These constraints can be used to automatically infer typings for programs. The next proposition is proved in the full version of this paper [14].

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

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 [17]. For a given γ let $\gamma_I(x) = T$ where $\langle T, U\rangle\ var = \gamma_{\mathsf{Data}}(x)$.

Theorem 1. *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 $\gamma(x) \geq \langle\{\{\mathsf{h}\}_\emptyset\}, \mathsf{Data}\rangle\ var$ for all $x \in \mathbf{Var}_\mathrm{S}$, $\gamma(x) \geq \langle\emptyset, \mathsf{Data}\rangle\ var$ for all $x \in \mathbf{Var}_\mathrm{I} \cup \mathbf{Var}_\mathrm{P}$, and $\bigvee_{x\in\mathbf{Var}_\mathrm{P}} \gamma_I(x) \not\geq \{\{\mathsf{h}\}_\emptyset\}$ 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 1 follows by some simple manipulation of probability distributions [14] from the following lemma basically stating that the public outputs of a program satisfying the premises of Thm. 1 can be computed without ever accessing the secret inputs.

Lemma 1 (Simulation Lemma). *Let P, $\mathbf{Var}$, $\mathbf{Var}_\mathrm{S}$, $\mathbf{Var}_\mathrm{P}$, $\mathbf{Var}_\mathrm{I}$ satisfy the premises of Theorem 1. 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 $\mathbf{Var}_\mathrm{S}$, and for every polynomial-time samplable family of probability distributions D over program states the families $\{\!|(S_n|_{\mathbf{Var}_\mathrm{S}}, S'_n|_{\mathbf{Var}_\mathrm{P}}) : S_n \leftarrow D_n, \langle\mathsf{P}, S_n\rangle \overset{n}{\Longrightarrow} D'_n, S'_n \leftarrow D'_n|\!\}$ and $\{\!|(S_n|\mathbf{Var}_\mathrm{S}, S'_n|_{\mathbf{Var}_\mathrm{P}}) : S_n \leftarrow D_n, \langle\mathsf{P}', S_n\rangle \overset{n}{\Longrightarrow} D'_n, S'_n \leftarrow D'_n|\!\}$ are indistinguishable.*

Let us give a short description of the proof of the Simulation Lemma (full proof can be found in [14]). The main tools for constructing the program P' and showing that its public outputs are indistinguishable from those of P are *probabilistic bisimulations* and the definition of secure encryption (9). The definition states that sometimes the encryption $\mathcal{E}nc(k, y)$ may be replaced with $\mathcal{E}nc(k_\mathrm{n}, \mathbf{0})$ where $\mathbf{0}$ is a constant and k_n is a fixed key (generated somewhere in the beginning of the program).

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. If h does not occur in $\bigvee_{x\in\mathbf{Var}_\mathrm{P}} \gamma_I(x)$ then the program P' mentioned in the Simulation Lemma can be constructed by just deleting all statements that access variables of types where h occurs. These statements are assignments to variables whose information type is at least $\{\mathsf{h}\}_\mathcal{G}$ and *if*- and *while*-statements whose guards have types with the same property. We can show that P and P' are bisimilar with respect to a bisimulation that requires the equality of public variables.

If h occurs in $\bigvee_{x\in\mathbf{Var}_\mathrm{P}} \gamma_I(x)$ then we repeatedly use the indistinguishability (9) to construct programs that use certain keys to encrypt only public data (actually, the constant $\mathbf{0}$), not secrets. The behavior of these programs is indistinguishable from the original program if we only look at public variables. The typing γ is also a valid typing for these programs, but they also have more permissive typings. A valid typing γ'' of the last constructed program P'' is such, that h does not occur in $\bigvee_{x\in\mathbf{Var}_\mathrm{P}} \gamma''_I(x)$. From the program P'' one can construct P' as before.

The replacement of encryptions may only be done if the encryption key is only used in ways that is possible in (9) — it may only be used for encryption. To better account for the flow of different keys, we first separate the keys with different labels. For this we introduce to our programming language a new expression

$$\mathcal{CE}nc(k^{(t)}\|i_1, k_1, y_1|i_2, k_2, y_2|\ldots|i_n, k_n, y_n), \tag{28}$$

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 (28) is

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

In the set of variables $\mathbf{Var}$ we replace each k where $\gamma(k) = \langle T, \mathsf{Key}_{\{i_1,\ldots,i_n\}}\rangle\ var$ by the variables $k^{(t)}$ and $k^{(i_1)}, \ldots, k^{(i_n)}$. The variable $k^{(t)}$ gets the type $\langle T, \mathsf{Data}\rangle\ var$; its value chooses which one of the variables $k^{(i_j)}$ is to be used. The type of $k^{(i_j)}$ will be $\langle T, \mathsf{Key}_{\{i_j\}}\rangle\ var$. Let γ_0 be the new typing. In the program we have to change key generation statements (we assign the key to the right $k^{(i_j)}$ and i_j to $k^{(t)}$), the assignments of a key to a key (we copy $k^{(t)}$ and all variables $k^{(i_j)}$), the uses of a key as an encryption key (we use the expression (28)) and the uses of a key in other ways (we use nested *if*-statements to check for different values of $k^{(t)}$ and use the right $k^{(i_j)}$ inside). The resulting program P_0 is bisimilar to P, it types according to γ_0 and the values of all variables of P can be recovered from the values of variables of P_0.

We introduce a new variable $k_\mathfrak{n}$ to the set of variables of P_0 and prepend P_0 with the statement $k_\mathfrak{n} := \mathcal{G}en^\mathfrak{n}$; here $\mathfrak{n}$ is a new key label. The key $k_\mathfrak{n}$ will be used by the transformed programs instead of keys that have been processed. In short, $k_\mathfrak{n}$ will play the role of k at the right hand side of (9).

We will now describe one iteration of replacing the encryptions with encryptions under the key $k_\mathfrak{n}$. Let P_o be the current program and γ_o the current typing; initially

$\mathsf{P}_o = \mathsf{P}_0$ and $\gamma_o = \gamma_0$. Consider again the type $T_P = \bigvee_{x \in \mathbf{Var}_\mathrm{P}} \gamma_{oI}(x)$. The type T_P satisfies certain invariants — they are satisfied for P_0 and remain satisfied during the iterations. First — the key label $\mathfrak{n}$ either does not occur in T_P (this is the case for P_0) or is free in it. Second — if $\mathfrak{n}$ encrypts some $i \in \{\mathsf{h}\} \cup \mathcal{G}$ in T_P then some other key label encrypts i in T_P as well. Therefore, if any key labels occur in T_P as a key (if no key labels occur as a key then h does not occur in T_P; then we are done, see above) then some $i \neq \mathfrak{n}$ is free in T_P. This follows from the fact that each $T \in \mathcal{T}_2$ (in normal form) contains a free key label, if any key labels occur in T as a key. If we delete from T_P all elements $\{t\}_N$ where $\mathfrak{n} \in N$ then we get another element of $\mathcal{T}_2$ where some key labels still occur as a key.

We delete from P_o all statements that access variables of types where i occurs as data; this deletion is identical to the deletion of h above. The deletion does not change the values of public variables because their types are not larger than T_P and i does not occur in T_P as data. In the resulting program keys generated at statements $x := \mathcal{G}en^i$ are only used for encryption. We then replace triples $|i, k, y|$ in $\mathcal{CE}nc$-expressions, where $\gamma_o(k) = \langle T, \mathsf{Key}_{\{i\}} \rangle\ var$, with $|i, k_\mathfrak{n}, \mathbf{0}|$. After that, if the triples of a $\mathcal{CE}nc$-expression all have $k_\mathfrak{n}$ and $\mathbf{0}$ as their second and third components, we replace the entire expression with $\mathcal{E}nc(k_\mathfrak{n}, \mathbf{0})$. In this way we get rid of encrypted *secrets* — the type of $\mathcal{E}nc(k_\mathfrak{n}, \mathbf{0})$ is $\langle \emptyset, \emptyset, \mathsf{Data} \rangle$. The resulting program $\mathsf{P}_{o'}$ is the input to the next iteration.

The typing γ_o changes as well. All variables in whose types i occurred as data will be deleted. For the rest of the variables x we get $\gamma_{o'}(x)$ from $\gamma_o(x) = \langle T_x, U_x \rangle\ var$ in the following way. The usage type U_x remains the same. The information type will contain all such $\{t\}_N \in T_x$ where $i \notin N$. If $\{t\}_{N \cup \{i\}} \in T_x$ (here $i \notin N$) then the information type according to $\gamma_{o'}$ may or may not contain $\{t\}_{N \cup \{\mathfrak{n}\}}$. We choose the least $\gamma_{o'}$ satisfying these conditions that is a valid typing of $\mathsf{P}_{o'}$.

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 [13] 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

1. Abadi, M.: Secrecy by Typing in Security Protocols. Journal of the ACM **46** (1999) 749–786
2. Abadi, M., Blanchet, B.: Secrecy types for asymmetric communication. Theoretical Computer Science **298** (2003) 387–415

3. Abadi, M., Gordon, A.: A Calculus for Cryptographic Protocols: The Spi Calculus. Information and Computation **148** (1999) 1–70
4. Abadi, M., Jürjens, J.: Formal Eavesdropping and Its Computational Interpretation. In proc. of TACS 2001 (LNCS 2215), pages 82–94
5. Abadi, M., Rogaway, P.: Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In proc. of the International Conference IFIP TCS 2000 (LNCS 1872), pages 3–22
6. Backes, M., Pfitzmann, B.: Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library. In proc. of CSFW 2004, pages 204–218
7. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In proc. of FOCS '01, pages 136–145
8. Denning, D.: A Lattice Model of Secure Information Flow. Communications of the ACM **19** (1976) 236–243
9. Denning, D., Denning, P.: Certification of Programs for Secure Information Flow. Communications of the ACM **20** (1977) 504–513
10. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **IT-29** (1983) 198–208
11. Laud, P.: Semantics and Program Analysis of Computationally Secure Information Flow. In proc of. ESOP 2001 (LNCS 2028), pages 77–91
12. Laud, P.: Encryption Cycles and Two Views of Cryptography. In proc. of Nordsec 2002, pages 85–100
13. Laud, P.: Handling Encryption in Analyses for Secure Information Flow. In proc. of ESOP 2003 (LNCS 2618), pages 159–173
14. Laud, P., Vene, V.: A Type System for Computationally Secure Information Flow. Tech. Report IT-LU-O-043-050307, Cybernetica AS, March 7th 2005.
15. Lincoln, P., Mitchell, J., Mitchell, M., Scedrov, A.: A Probabilistic Poly-Time Framework for Protocol Analysis. In proc. of ACM CCS '98, pages 112–121
16. Lincoln, P., Mitchell, J., Mitchell, M., Scedrov, A.: Probabilistic Polynomial-Time Equivalence and Security Analysis. In proc. of the World Congress on Formal Methods in the Development of Computing Systems '99 (LNCS 1708), pages 776–793
17. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1999)
18. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In proc. of POPL '99, pages 228–241
19. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE Journal on Selected Areas in Communications **21** (2003) 5–19
20. Pfitzmann, B., Waidner, M.: A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In proc. of IEEE S&P 2001, pages 184–200
21. Smith, G., Volpano, D.: Secure Information Flow in a Multi-threaded Imperative Language. In proc. of POPL '98, pages 355–364
22. Volpano, D.: Secure Introduction of One-way Functions. In proc. of CSFW '00, pages 246–254
23. Volpano, D., Smith, G., Irvine, C.: A Sound Type System for Secure Flow Analysis. Journal of Computer Security **4** (1996) 167–187
24. Volpano, D.M., Smith, G.: Eliminating Covert Flows with Minimum Typings. In proc. of CSFW '97, pages 156–169
25. Volpano, D., Smith, G.: Probabilistic Noninterference in a Concurrent Language. In proc. of CSFW '98, pages 34–43
26. Volpano, D., Smith, G.: Verifying Secrets and Relative Secrecy. In: proc. of POPL 2000, pages 268–276
27. Yao, A.: Theory and applications of trapdoor functions (extended abstract). In proc. of FOCS '82, pages 80–91