Semantics and Program Analysis of Computationally Secure Information Flow

Peeter Laud[⋆]

FR Informatik
Universität des Saarlandes
laud@cs.uni-sb.de

Abstract. This paper presents a definition of secure information flow. It is not based on noninterference, but on computational indistinguishability of the secret inputs, when the public outputs are observed. This definition allows cryptographic primitives to be handled. This paper also presents a Denning-style information-flow analysis for programs that use encryption as a primitive operation. The proof of the correctness of the analysis is sketched.

1 Introduction

When is a program safe to run? One aspect of safety is confidentiality, which arises when the inputs and outputs of the program are partitioned into several different security classes. The typical classification of data is into public and confidential (of course, more complex classifications are also possible); for example, data that is received from or sent to a network may be treated as public, and data that is local to the site executing the program, as confidential. Our goal is to verify that an attacker who can observe the public outputs of the program cannot learn anything about the confidential inputs. In this case we say, that the program has *secure information flow.*

What does it mean that an attacker can or cannot learn anything? There exists quite a large body of literature studying different instances of secure information flow, e.g., [3,4,5,7,10,11,12,13,14,15,16]. With the notable exception of [15,16], security is defined through noninterference, i.e., it is required that the public outputs of the program do not contain any information (in the information-theoretic sense) about the confidential inputs. This corresponds to an all-powerful attacker who, in his quest to obtain confidential information, has no bounds on the resources (time and space), that it can use. Furthermore, in these definitions an "attacker" is represented by an arbitrary function, which does not even have to be a computable function; the attacker is permitted essentially arbitrary power.

The approach of defining the concept of "secure information flow" by means of an all-powerful adversary does not allow for cryptographic primitives to be treated easily, as they are usually only computationally secure, but not informa-

[⋆] Supported by Deutsche Forschungsgemeinschaft, under the Graduate Studies Program #623, "Quality Guarantees for Computer Systems" tion-theoretically. Also, realistic adversaries are resource-bounded; hence this should be reflected in our definitions, especially if we can show that more programs are secure by this means.

The contributions of this paper are as follows:

– A definition of secure information flow that corresponds to an adversary working in probabilistic polynomial time (in the following abridged as PPT; where "polynomial" means polynomial in a suitable security parameter[1]). Our definition of secure information flow is stated in terms of program's inputs and outputs only, in this sense it is similar to that of Leino and Joshi [10].
– A program analysis that allows us to certify that the flow of information in the program is secure in the aforementioned sense. The programming language contains a binary operator "encrypt message x under the key k"[2]. The analysis reflects that finding the message from the cryptotext is infeasible without knowing the key.

2 Related Work

The use of program analysis to determine information flow was pioneered by Denning [4,5]. She instrumented the semantics of programs with annotations that expressed which program variables were dependent on which other program variables. The definition of secure information flow was given in terms of these instrumentations [5]. Also, she gave an accompanying program analysis.

Volpano et al. [14] gave a definition of secure information flow without using any instrumentations. They define a program to be secure if there exists a simulation of the program that operates only on the public variables and delivers the same public outputs. They also give a type system for certifying programs for secure information flow.

Leino and Joshi [10] give a definition of secure information flow that makes use of programs' inputs and outputs only. They define the program to be secure if the values of its secret inputs cannot be observed from its public outputs. Sabelfeld and Sands [13] and Abadi et al. [1] have given generalisations of this idea. However, none of these papers describes an accompanying mechanical verification tool.

Recently, Volpano and Smith [15,16] have weakened the security definition a bit (it is no longer noninterference) to handle cryptographic primitives, namely one-way functions. However, there are two ways in which their definition might be considered unsatisfactory:

– It is too restrictive in allowing the use of constructs that violate the noninterference condition.
– The notion of security is too lax (in [16] Volpano allows the adversary to find out *partial* information about the confidential input value).

[1] by encryption, the security parameter is related to (and often defined to be equal to) the key length

[2] also contains a nullary operator "generate a new key"

Work has also been done to define and handle the integrity properties of data (e.g. [7,11]). These questions are outside the scope of this paper.

Another work, which is not about secure information flow, but which has influenced the current paper, is that of Abadi and Rogaway [2]. They provide a computational justification of one part of the formal approach to cryptography — namely to the construction of messages from simpler ones by tupling and encryption. The construction of messages could be viewed as a straight-line program; actually, if one only considers straight-line programs, then their results subsume ours. Because of our treatment of control flow, we have seemingly more restrictions on programs than they have.

3 Syntax and Semantics

The programming language that we consider is a simple imperative programming language (the WHILE-language). Given a set **Var** of variables and a set **Op** of arithmetic, relational, boolean etc. operators, the syntax of the programs is given by the following grammar:

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

where $b, x, x_1, \ldots, x_k$ range over **Var**, o ranges over **Op** and $\mathsf{P}_1, \mathsf{P}_2$ range over programs. We assume that there are two distinguished elements in the set **Op** — a binary operator $\mathcal{E}nc$ ($\mathcal{E}nc(k, x)$ is the encryption of the message x under the key k) and a nullary operator $\mathcal{G}en$ (generating a new key).

We also make use of flowcharts as the program representation. Each node of the flowchart contains either an assignment statement or a test. Additionally, we add an extra *start*- and an extra *end*-node to the flowchart. It should be clear, how a program P is converted to a flowchart.

The semantics that we give for the programming language, is denotational in style. If **State** is the set of all possible internal states of the program, then the denotational semantics usually has the type $\mathbf{State} \rightarrow \mathbf{State}_\perp$, i.e. it maps the initial state of the program to its final state. $\mathbf{State}_\perp := \mathbf{State} \uplus \{\perp\}$, where $\perp$ denotes non-termination. Moreover, one usually defines $\mathbf{State} := \mathbf{Var} \rightarrow \mathbf{Val}$, where **Val** is the set of all possible values that the variables can take. The set **Val** is usually not specified any further.

Furthermore, for defining the semantics of programs one requires that for each operator $o \in \mathbf{Op}$ of arity k its semantics $[\![o]\!] : \mathbf{Val}^k \rightarrow \mathbf{Val}$ has already been defined. We want that $[\![\mathcal{E}nc]\!]$ and $[\![\mathcal{G}en]\!]$ satisfy certain cryptographic definitions, namely the following:

Definition 1 (from [2]). *An encryption scheme* (*Gen*, *Enc*) [3] *is* which-key and repetition concealing, *if for every PPT algorithm with two oracles* $\mathcal{A}^{(\cdot),(\cdot)}$, *the following difference of probabilities is negligible in* n*:*

[3] This tuple should also have the third component — the decryption algorithm, but we do not need it for that definition

$$\Pr\big[k, k' \leftarrow \mathit{Gen}(\mathbf{1}^n)\ :\ \mathcal{A}^{\mathit{Enc}(\mathbf{1}^n,k,\cdot),\mathit{Enc}(\mathbf{1}^n,k',\cdot)}(\mathbf{1}^n) = 1\big] - \\ \Pr\big[k \leftarrow \mathit{Gen}(\mathbf{1}^n)\ :\ \mathcal{A}^{\mathit{Enc}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|}),\mathit{Enc}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})}(\mathbf{1}^n) = 1\big]\ .$$

Here $\ell(n) \in \mathbb{Z}[n]$ is a suitable *length polynomial*. An encryption system is thus which-key and repetition concealing, if no (polynomially bounded) adversary can distinguish the following two situations: There are two black boxes.

1. Given a bitstring as the input, the boxes encrypt it and return the result. They use different encryption keys.
2. The black boxes throw away their input and return the encryption of a fixed bit-string $\mathbf{0}^{\ell(n)}$. Both boxes use the same encryption key.

From this definition we see, that the structure of the semantics has to be more complicated than just having the type $\mathbf{State} \rightarrow \mathbf{State}_\perp$. The issues are:

– According to Def. 1, *Enc* and *Gen* are not functions, but are families of functions, indexed by the *security parameter* $n \in \mathbb{N}$. Hence the semantics also has to be a family of functions, mapping a program's inputs to its outputs. For each n we have to define a set $\mathbf{State}_n$ and the n-th component of semantics would map each element of $\mathbf{State}_n$ to $\mathbf{State}_{n\perp}$.
– The algorithms *Enc* and *Gen* operate over bit-strings[4]. Therefore we specify $\mathbf{State}_n := \mathbf{Var} \rightarrow \mathbf{Val}_n$ and $\mathbf{Val}_n := \{0,1\}^{\ell(n)}$.
– Clearly no family of *deterministic* functions satisfies Def. 1; the families of functions *Gen* and *Enc* have to be *probabilistic*. Thus the semantics of programs has to be probabilistic, too. Its n-th component maps each element of $\mathbf{State}_n$ to a probability distribution over $\mathbf{State}_{n\perp}$. We denote the set of probability distributions over a set $\mathcal{X}$ by $\mathcal{D}(\mathcal{X})$.

There is one more issue. We are not interested in program runs that take too much time. The encryption scheme is defined to be secure only against polynomially bounded adversaries. Thus we are interested in only "polynomially long" program runs (i.e. the set of interesting runs may be different for different values of the security parameter). We support this discrimination by defining a previously described family of functions mapping inputs to outputs *for each path in the flowchart* from the *start*- to the *end*-node. Also, the (intuitive) meaning of $\perp$ changes. Instead of denoting non-termination it now means "control flow does not follow that path".

To sum it all up, the semantics $[\![\mathsf{P}]\!]_{\mathrm{Path}}$ of a program P has the type $\mathbf{Path} \rightarrow \prod_{n\in\mathbb{N}}(\mathbf{State}_n \rightarrow \mathcal{D}(\mathbf{State}_{n\perp}))$, where $\mathbf{Path}$ denotes all paths in the flowchart of P from *start* to *end*. Also

– for each operator $o \in \mathbf{Op}$ of arity k the semantics of o is a family of (possibly probabilistic) functions $[\![o]\!] : \prod_{n\in\mathbb{N}}(\mathbf{Val}_n^k \rightarrow \mathcal{D}(\mathbf{Val}_n))$;
– $([\![\mathcal{G}en]\!], [\![\mathcal{E}nc]\!])$ is a which-key and repetition concealing encryption scheme.

[4] this is not so explicit in the above definition, but it is the standard in complexity-theoretic definitions of cryptographic primitives

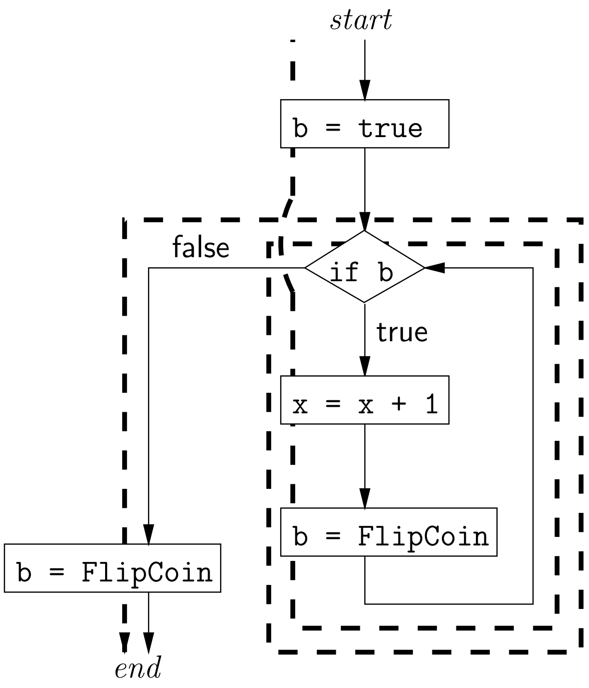

Fig. 1. Example for the semantics of paths

To explain the semantics of the program at a certain path, look at the program P in Fig. 1. Suppose that the semantics of the operator `FlipCoin` is such, that it returns either true or false with equal probability. Let the marked path be P. For each $x_{\text{init}}, b_{\text{init}} \in \mathbf{Val}_n$ we have

$$[\![\mathsf{P}]\!]_{\text{Path}}(P)\left(\left\{\begin{array}{l} x \mapsto x_{\text{init}} \\ b \mapsto b_{\text{init}} \end{array}\right\}\right) = \left\{\begin{array}{ll} \left\{\begin{array}{l} x \mapsto x_{\text{init}} + 2 \\ b \mapsto \mathsf{true} \end{array}\right\} & \mapsto \frac{1}{8} \\ \left\{\begin{array}{l} x \mapsto x_{\text{init}} + 2 \\ b \mapsto \mathsf{false} \end{array}\right\} & \mapsto \frac{1}{8} \\ \bot & \mapsto \frac{3}{4} \end{array}\right\}.$$

It should be clear from this example, how $[\![\mathsf{P}]\!]_{\text{Path}}$ is calculated. We omit the further specifications from this paper.

4 Confidentiality Definition

The confidentiality definition expresses that something holds for all (long enough) polynomially long computations. This needs an exact definition of a polynomially long computation.

Definition 2. *A function* $\mathcal{S} : \mathbb{N} \to \mathcal{P}(\mathbf{Path})$ *is a* polynomially long path-set, *iff there exists a polynomial* $q \in \mathbb{Z}[n]$, *such that for all* n, *the length of the elements (paths) of* $\mathcal{S}(n)$ *is not greater than* $q(n)$.

We denote the set of all polynomially long path-sets by $\mathcal{PLP}$. The set $\mathcal{PLP}$ is ordered by pointwise inclusion.

Given a program P and a polynomially long path-set $\mathcal{S}$, we can define, how P behaves on $\mathcal{S}$.

$$[\![\mathsf{P}]\!] : \mathcal{PLP} \to \prod_{n\in\mathbb{N}} (\mathbf{State}_n \to \mathcal{D}(\mathbf{State}_{n\perp}))$$

$$[\![\mathsf{P}]\!](\mathcal{S})(S_{n,s}) = \left\{ \begin{array}{l} S_n \mapsto \sum\limits_{P\in\mathcal{S}(n)} [\![\mathsf{P}]\!]_{\mathrm{Path}}(P)(S_{n,s})(S_n) \\ \perp \ \mapsto 1 - \sum\limits_{S_n\in\mathbf{State}_n} [\![\mathsf{P}]\!](\mathcal{S})(S_{n,s})(S_n) \end{array} \right\},$$

where $n \in \mathbb{N}$, $S_{n,s}, S_n \in \mathbf{State}_n$. Thus the behaviour of P on a path-set is the "sum" of its behaviours on the elements of this path-set. The probability, that a final state S_n is reached over $\mathcal{S}$ is the sum of probabilities that this state is reached over the elements of $\mathcal{S}(n)$. $\perp$ means, that the control flow does not follow any path in $\mathcal{S}(n)$.

We are going to define, what it means, that certain outputs of the program do reveal something about a part (the confidential part) of the input to the program. What kind of object is this "part of the input"? In most general case it is just a family of functions $c = \{c_n\}_{n\in\mathbb{N}}$, $c_n : \mathbf{State}_n \to \{0,1\}^*$. For example, if the set of variables has been partitioned into public and confidential variables, then $c_n(S_n)$ would return a tuple consisting of the values of all confidential variables in the state S_n. As we are going to define *computational* security, we require, that c is polynomial-time computable, i.e. there exists an algorithm $\mathcal{C}$ that works in polynomial time and $\mathcal{C}(\mathbf{1}^n, S_n) = c_n(S_n)$.

Definition 3. *A part $c = \{c_n\}_{n\in\mathbb{N}}$ of the input, given to program* P *is* recoverable *from the final values of the variables in $Y \subseteq \mathbf{Var}$, given that the input to* P *is distributed accordingly to $D_s = \{D_{n,s}\}_{n\in\mathbb{N}}$, $D_{n,s} \in \mathcal{D}(\mathbf{State}_n)$, iff there exists a polynomial-time computable predicate $B = \{B_n\}_{n\in\mathbb{N}}$, $B_n : \{0,1\}^* \to \{0,1\}$ and a polynomially long path-set $\mathcal{S}_0 \in \mathcal{PLP}$, such that for all polynomially long path-sets $\mathcal{S} \in \mathcal{PLP}$, where $\mathcal{S} \geq \mathcal{S}_0$, there exists a PPT algorithm $\mathcal{A}$, such that for all PPT algorithms $\mathcal{B}$ the following difference is not negligible in n:*

$$\Pr\big[S_{n,s} \leftarrow D_{n,s}, S_n \leftarrow [\![\mathsf{P}]\!](\mathcal{S})(S_{n,s}) \ : \ \mathcal{A}(\mathbf{1}^n, S_n|_Y) = B_n(c_n(S_{n,s}))\big] - \\ \Pr\big[S_{n,s} \leftarrow D_{n,s} \ : \ \mathcal{B}(\mathbf{1}^n) = B_n(c_n(S_{n,s}))\big] \ .$$

Let us explain some points of this definition a bit:

- We demand that after the program has run for long enough (but still polynomial!) time ($\exists\mathcal{S}_0\,\forall\mathcal{S} \geq \mathcal{S}_0$), its behaviour "stabilises" in the sense that nothing more will become or cease to be recoverable.
- We demand that after this stabilisation, a property that is similar to the notion of semantic security (more exactly, to its negation) of encryption systems must hold for $[\![\mathsf{P}]\!](\mathcal{S})$, with regard to c and Y. We require, that the final values of the variables in Y tell us something about c, that we did not know before. See [6, Sec. 5] (and also references therein) for the discussion of semantic security.

- The probability distribution of the inputs to the program has been made explicit in our definition. It has not usually been the case in earlier works (for example [4,11,14]), where one has *implicitly* assumed, that the input variables are independent of each other.
- The definition of recoverability is termination-sensitive, as the value S_n that is picked accordingly to $[\![\mathsf{P}]\!](\mathcal{S})(S_{n,s})$ may also be $\perp$ (in this case define the contraction of $\perp$ to Y to be $\perp$, too). However, it is not sensitive to timing.

We say, that the program P is secure for the initial distribution D_s, if the secret part of its input is not recoverable from the final values of the set of the public variables of P.

5 Program Analysis

We suppose that we have a fixed program P, the secret part of the input c that is calculated by the algorithm $\mathcal{C}$ in polynomial time, and the initial probability distribution D_s. Before we present our analysis, let us state some further constraints to the programs and to the details of the semantics, that we have not stated before, because they had been unnecessary to give the definition of recoverability. For some of these constraints it is intuitively clear that they must be obeyed, if we talk about computational security. Some other constraints are really the shortcomings of our quite simple analysis (comparable in its power to the one in [5]) and removal of them should be the subject of further work.

- For each operator $o \in \mathbf{Op}$, the semantics of o must be computable in polynomial time, i.e. there must exist a polynomial-time algorithm $\mathcal{O}$, such that $[\![o]\!]_n(\cdot) = \mathcal{O}(\mathbf{1}^n, \cdot)$ for all n. Otherwise we could have an atomic operation "crack the key" in our programming language . . . . Thus the necessity of this constraint should be obvious.
- The probability distribution D_s must be polynomial-time constructible, i.e. there must be a PPT algorithm $\mathcal{D}$, such that the random variables $D_{n,s}$ and $\mathcal{D}(\mathbf{1}^n)$ have identical distribution for each n. Without this requirement it might be possible to use the initial distribution as an oracle of some sort and thus answer questions that a PPT algorithm should be unable to answer.
- Keys and non-keys must not be mixed, i.e., each variable in **Var** should have a type of either "key" or "data" and it is not allowed to substitute one for another. The output of $\mathcal{G}en$ and the first input (the key) of $\mathcal{E}nc$ have the type "key", everything else has the type "data". This has several consequences:
  - One may not use data as keys. This is an obvious constraint, because it just means that the used keys must be good ones, i.e. created by $\mathcal{G}en$. An extra constraint is that the initial values of the variables of type "key" must be good keys, too.
  - One may not use keys as data. This is a shortcoming of our analysis, that for example [2] (almost) does not have. We believe that using their techniques it is possible to get rid of that constraint.
  - There is no decryption operator. Also a shortcoming. We believe that by building definition-use chains between encryption and decryption operators it is possible to keep track, what could be calculated from the encrypted data, and to remove the shortcoming that way.

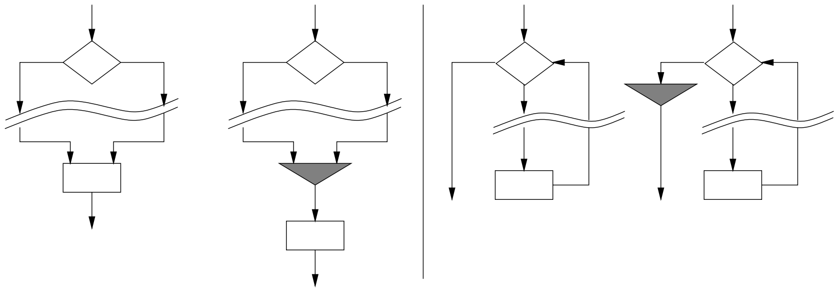

Fig. 2. Auxiliary nodes to separate control dependency regions

- At each program point, it must be statically known, which keys are equal and which are not. This is probably the biggest shortcoming of our analysis. We defer its discussion to Sec. 7. For each node N of the flowchart and each variable k with type "key", we let $\mathsf{E}(N, k) \subseteq \mathbf{Var}$ denote the set of keys that are equal to k at the node N. If the program satisfies this requirement, then one can compute E easily by using the methods of alias analysis, see, for example [9, Sec. 4.2.1 and 4.2.2]. Note that this requirement also generates a constraint on D_s, similar to the case of requiring good keys.
- The program P must run in expected polynomial time. Although our security definition is termination sensitive, our analysis is not.

To handle the implicit flows more comfortably, we add dummy nodes (we call them *merge*-nodes) to the flowchart in those places where the control dependency regions end. Figure 2 explains adding *merge*-nodes to branches and while-loops. In this way the starts and ends of the control dependency regions are marked in the flowchart (starts with *if*, ends with *merge*) and we do not have to treat the edges going from one control dependency region to another differently from those that start and end inside the same region.

The type of the analysis $\mathcal{W}$ is $\mathit{Node} \rightarrow \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}}))$, where Node is the set of the nodes in the flowchart and $\widetilde{\mathbf{Var}} = \mathbf{Var} \cup \mathit{Node}_{\mathit{if}}$, where $\mathit{Node}_{\mathit{if}} \subset \mathit{Node}$ is the set of all *if*-nodes. Here $\mathcal{P}_{\mathrm{U}}(\mathcal{X})$, where $\mathcal{X}$ is a partially ordered set, denotes the set of all such subsets of $\mathcal{X}$ that are upper closed. We will also make use of an operator $\mathsf{cl}_{\mathrm{U}}(\cdot)$ which, when applied to a subset of a partially ordered set, returns its upper closure.

$Y \in \mathcal{W}(N)$, where $N \in \mathit{Node}$ and $Y \subseteq \widetilde{\mathbf{Var}}$ means that the analysis has determined that at node N the secret part of the input of the program may be recoverable from the values of the variables in $Y \cap \mathbf{Var}$, if one also takes into account that at the nodes in $Y \cap \mathit{Node}_{\mathit{if}}$ the branch leading to the node N was chosen. Actually the *if*-nodes in Y are used to track the implicit flow from the boolean variables that guard them. A better explanation can be given after presenting the transfer functions for *if*- and *merge*-nodes. The analysis may erroneously report that the secret part of the input is recoverable from Y, when in reality it isn't. However, the analysis may not err to the other side.

We continue with presenting the transfer functions $[\![N]\!]_{\mathrm{abstr}}$, where $N \in \mathit{Node}$. The type of these functions is obviously $\mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}})) \to \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}}))$. We start by giving two auxiliary functions (of the same type). Let $x \in \widetilde{\mathbf{Var}}$, $X, Y \subseteq \widetilde{\mathbf{Var}}$, $\mathcal{X} \in \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}}))$.

$$\begin{aligned}\mathsf{kill}\big[x\big](\mathcal{X}) &= \mathsf{cl}_{\mathrm{U}}(\{Z \mid Z \in \mathcal{X}, x \notin Z\})\\ \mathsf{flow}\big[X \Rightarrow Y\big](\mathcal{X}) &= \mathsf{cl}_{\mathrm{U}}(\mathcal{X} \cup \big\{(Z \backslash X) \cup \{v\} \mid Z \in \mathcal{X}, v \in Y\big\})\end{aligned}$$

The meaning of the function $\mathsf{kill}\big[x\big]$ should be obvious. The function $\mathsf{flow}\big[X \Rightarrow Y\big]$ describes the flow of information from the variables in X to the variables in Y. It says, that if an adversary can find something from the values of the variables in the set Z, then after the information has flown, the properties of the values of the variables in $Z \cap X$, that the adversary makes use of, may also be derivable from the value of some variable in Y.

- The node N is labeled with $x := o(x_1, \ldots, x_k)$, where $o \neq \mathcal{E}nc$. Assume w.l.o.g. that x is different from $x_1, \ldots, x_k$.

$$[\![N]\!]_{\mathrm{abstr}} = \mathsf{flow}\big[\{x_1, \ldots, x_k\} \Rightarrow \{x\}\big] \circ \mathsf{kill}\big[x\big]$$

- The node N is labeled with $x := \mathcal{E}nc(k, y)$. We again assume that $x \neq y$. Let $K \subseteq \mathbf{Var}$, such that all variables in K have the type "key". Define one more auxiliary function

$$\mathsf{flow}\big[X \overset{K}{\Rightarrow} Y\big](\mathcal{X}) = \mathsf{cl}_{\mathrm{U}}(\mathcal{X} \cup \big\{(Z \backslash X) \cup \{v, k\} \mid Z \in \mathcal{X}, v \in Y, k \in K\big\}),$$

which describes the flow of information from X to Y, during which it becomes encrypted with some key from the set K.

$$[\![N]\!]_{\mathrm{abstr}} = \mathsf{flow}\big[\{y\} \overset{\mathsf{E}(N,k)}{\Rightarrow} \{x\}\big] \circ \mathsf{kill}\big[x\big]$$

- The node N is labeled with *if* b.

$$[\![N]\!]_{\mathrm{abstr}} = \mathsf{flow}\big[\{b\} \Rightarrow \{N\}\big]$$

- The node N is a *merge*-node. Let the corresponding *if*-node be N_{if}. Let $\mathbf{Var}_{\mathrm{asgn}} \subseteq \mathbf{Var}$ be the set of variables that are assigned to somewhere between N_{if} and N.

$$[\![N]\!]_{\mathrm{abstr}} = \mathsf{kill}\big[N_{\mathrm{if}}\big] \circ \mathsf{flow}\big[\{N_{\mathrm{if}}\} \Rightarrow \mathbf{Var}_{\mathrm{asgn}}\big]$$

  Here we see how the implicit flow from a boolean variable to those variables, assigning to which it controls, is taken care of. ($b \Rightarrow N_{\mathrm{if}} \Rightarrow \mathbf{Var}_{\mathrm{asgn}}$)
- The transfer function for the *end*-node is the identity function.
- The node N is the *start*-node. What should be our initial analysis information? It must describe the initial probability distribution D_s. For example, if the secret part c of the input is just the values of the variables from the set $\mathbf{Var}_{\mathrm{conf}}$ and according to D_s, all input variables are independent of each other, the initial analysis information must be at least

$\mathsf{cl}_{\mathrm{U}}(\{\{x\} \mid x \in \mathbf{Var}_{\mathrm{conf}}\})$. In general case (and formally), if $Y_1, \ldots, Y_l \subseteq \mathbf{Var}$ and there exist a polynomial-time computable predicate $B = \{B_n\}_{n \in \mathbb{N}}$ and a PPT algorithm with an oracle $\mathcal{A}^{(\cdot)}$, such that for each PPT algorithm $\mathcal{B}$ there exists an i, $1 \leq i \leq l$, such that

$$\Pr\big[S_{n,s} \leftarrow D_{n,s} \;:\; \mathcal{A}^{\mathcal{B}(\mathbf{1}^n)}(\mathbf{1}^n, i, S_{n,s}|_{Y_i}) = B(c_n(S_{n,s}))\big] - \Pr\big[S_{n,s} \leftarrow D_{n,s} \;:\; \mathcal{B}(\mathbf{1}^n) = B(c_n(S_{n,s}))\big]$$

is not negligible in n, then at least one of the sets $Y_1, \ldots, Y_l$ must be an element of the initial analysis information.

The set $\mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}}))$ is obviously ordered by inclusion and the least upper bound is the set union. This completes the specification of the data flow analysis.

6 Proof of Correctness

Theorem 1. *Let* P *be a program in the* WHILE*-language, let* $\mathbf{Var}$ *be its set of variables and* N_{end} *be the node labeled with end in its flowchart. Let* c *be the secret part of the input to* P*, let* D_s *be the probability distribution of the input, let* $Y \subseteq \mathbf{Var}$*. Let the requirements in the beginning of Sec. 5 be satisfied and let* $\mathcal{W}$ *be the program analysis for* P*,* c *and* D_s*. If* c *is recoverable from the final values of the variables in* Y*, given that the input to* P *is distributed accordingly to* D_s*, then* $Y \in \mathcal{W}(N_{end})$*.*

The theorem thus says that if one can find out something about the secret input when observing certain outputs, then the analysis reports that these outputs give away secret information.

We define some auxiliary notions to present a technical lemma. Let $\overline{\mathcal{W}} : \mathcal{P}(\mathbf{Var}) \to \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\mathbf{Var}))$ be such, that $\overline{\mathcal{W}}(X)$, where $X \subseteq \mathbf{Var}$, equals $\mathcal{W}(N_{end})$ if we take $\mathcal{W}(N_{start})$ to be equal to $\mathsf{cl}_{\mathrm{U}}(\{X\})$ and calculate $\mathcal{W}(N_{end})$ by the analysis given in the previous section. Thus $\overline{\mathcal{W}}(X)$ is the set of sets of variables that are "caused to be" in the analysis result for N_{end} by the presence of X in the initial analysis information. For $Y \subseteq \mathbf{Var}$ let $\mathcal{M}(Y) := \{X \subseteq \mathbf{Var} \mid Y \in \overline{\mathcal{W}}(X)\}$ and let $\overline{\mathcal{M}}(Y)$ be the set of all maximal elements of $\mathcal{M}(Y)$.

Lemma 1. *Let* P *be a program. Let its flowchart be annotated with the assumptions about the equality of keys (i.e. let* E *be fixed). Let* $q \in \mathbb{Z}[n]$ *and let* $\mathcal{S} \in \mathcal{PLP}$ *be such, that a path* P *belongs to* $\mathcal{S}(n)$ *iff* $|P| \leq q(n)$*.*

There exist PPT algorithms $\{\mathcal{A}_X\}_{X \subseteq \mathbf{Var}}$*, such that*

- *the input of* $\mathcal{A}_X(\mathbf{1}^n, \cdot)$ *is a function of type* $X \to \mathbf{Val}_n$*;*
- *the output of* $\mathcal{A}_X(\mathbf{1}^n, \cdot)$ *is either* $\perp$ *or a set* $Z \subseteq \mathbf{Var}$ *and a function* $f : Z \to \mathbf{Val}_n$*.*

For all initial probability distributions $D_s = \{D_{n,s}\}_{n\in\mathbb{N}}$, which satisfies the annotations of the program and for which the expected running time of the program is bounded by the polynomial q, for all subsets $Y \subseteq \mathbf{Var}$ and all PPT algorithms $\mathcal{D}$ the following difference is negligible in n:

$$\Big(\sum_{X\in\overline{\mathcal{M}}(Y)} \Pr\big[S_{n,s} \leftarrow D_{n,s}, (Z,f) \leftarrow \mathcal{A}_X(\mathbf{1}^n, S_{n,s}|_X) \;:\; Y \subseteq Z \wedge \mathcal{D}(\mathbf{1}^n, S_{n,s}, f|_Y) = 1\big]\Big) - \Pr\big[S_{n,s} \leftarrow D_{n,s}, S_n \leftarrow [\![\mathsf{P}]\!](\mathcal{S})(S_{n,s}) \;:\; \mathcal{D}(\mathbf{1}^n, S_{n,s}, S_n|_Y) = 1\big] \;.$$

The lemma is proved by induction over the syntax of the While-language.

The above lemma states an indistinguishability result ([6, Sec. 5] discusses the relationship between semantic security and indistinguishability). Intuitively, the lemma claims that if we know only some of the program's input variables, then we can still simulate the run of the program, by computing only the values of the variables that only depend on the values of known input variables. The final set of variables, whose values the algorithm knows at the end of the program, may depend on the chosen branches at *if*-nodes. Also, the lemma claims that the sets Z that the algorithms $\mathcal{A}_X$ output, are in average not too small — for each $Y \subseteq \mathbf{Var}$ there are sets $X \subseteq \mathbf{Var}$, for which $\mathcal{A}_X$ outputs $Z \supseteq Y$ with significant probability. Moreover, we can recreate the distribution $(S_{n,s}, S_n|_Y)$, where $S_{n,s}$ is the initial and S_n the corresponding final state of the program, with the help of the algorithms $\mathcal{A}_X$ (at least for a computationally bounded observer).

$\mathcal{A}_X$ works by executing the program (by following its flowchart). If the next statement is $x = o(x_1, \dots, x_k)$ and the algorithm currently knows $x_1, \dots, x_k$, then it will also know x, otherwise it will forget x, except for encryptions, where the algorithm randomly generates unknown inputs. At statement *if* b, if $\mathcal{A}_X$ currently knows b, then it will enter the right branch, otherwise it will jump directly to the corresponding *merge* and forget the variables in $\mathbf{Var}_{\mathrm{asgn}}$.

Proof (of the theorem). Suppose that the secret part c is recoverable from the final values of Y. According to the definition of recoverability, there exists an algorithm $\mathcal{A}$ that does that. The lemma provides us with the algorithms $\{\mathcal{A}_X\}_{X\subseteq\mathbf{Var}}$. For each $X \subseteq \mathbf{Var}$ we construct an algorithm $\mathcal{A}'^{(\cdot)}_X$ as follows: on input $(\mathbf{1}^n, f_0)$ we first run $\mathcal{A}_X$ on the same input, which gives us a set Z and a function $f : Z \rightarrow \mathbf{Val}_n$. We then check whether $Y \subseteq Z$ and

– if true, return $\mathcal{A}(\mathbf{1}^n, f|_Y)$;
– if false, invoke the oracle and return whatever it returns.

Let $\{Y_1, \dots, Y_l\}$ be the set $\overline{\mathcal{M}}(Y)$. Let $\mathcal{A}'^{(\cdot)}$ be an algorithm, that on input $(\mathbf{1}^n, i, f_0)$, where $1 \leq i \leq l$ runs the algorithm $\mathcal{A}'^{(\cdot)}_{Y_i}$ on $(\mathbf{1}^n, f_0)$. Let $\mathcal{B}$ be any PPT algorithm. We now claim that the following difference is negligible in n:

$$\Big(\sum_{i=1}^{l} \Pr\big[S_{n,s} \leftarrow D_{n,s} \;:\; \mathcal{A}'^{(\cdot)}(\mathbf{1}^n, i, S_{n,s}|_{Y_i}) = B(c_n(S_{n,s}))\big]\Big) -$$
$$\Big(\Pr\big[S_{n,s} \leftarrow D_{n,s}, S_n \leftarrow [\![\mathsf{P}]\!](\mathcal{S})(S_{n,s}) \;:\; \mathcal{A}(\mathbf{1}^n, S_n|_Y) = B_n(c_n(S_{n,s}))\big] +$$
$$(l-1)\Pr\big[S_{n,s} \leftarrow D_{n,s} \;:\; \mathcal{B}(\mathbf{1}^n) = B_n(c_n(S_{n,s}))\big]\Big) \ .$$

Thus there exists an i that $\Pr\big[S_{n,s} \leftarrow D_{n,s} : \mathcal{A}^{\mathcal{B}(\mathbf{1}^n)}(\mathbf{1}^n, i, S_{n,s}|_{Y_i}) = B(c_n(S_{n,s}))\big]$ is greater or equal or only negligibly less than

$$\frac{1}{l}\Pr\big[S_{n,s} \leftarrow D_{n,s}, S_n \leftarrow [\![\mathsf{P}]\!](\mathcal{S})(S_{n,s}) \;:\; \mathcal{A}(\mathbf{1}^n, S_n|_Y) = B_n(c_n(S_{n,s}))\big] +$$
$$\frac{l-1}{l}\Pr\big[S_{n,s} \leftarrow D_{n,s} \;:\; \mathcal{B}(\mathbf{1}^n) = B_n(c_n(S_{n,s}))\big]$$

but this is significantly greater than $\Pr\big[S_{n,s} \leftarrow D_{n,s} : \mathcal{B}(\mathbf{1}^n) = B(c_n(S_{n,s}))\big]$. Thus there exists a j, such that Y_j is in the initial analysis information. But $Y \in \overline{\mathcal{W}}(Y_j)$ and thus $Y \in \mathcal{W}(N_{end})$.

The claim follows from

$$\sum_{i=1}^{l} \Pr\big[S_{n,s} \leftarrow D_{n,s} \;:\; \mathcal{A}'^{(\cdot)}(\mathbf{1}^n, i, S_{n,s}|_{Y_i}) = B(c_n(S_{n,s}))\big] =$$
$$\sum_{X \in \overline{\mathcal{M}}(Y)} \Pr\big[S_{n,s} \leftarrow D_{n,s}, (Z,f) \leftarrow \mathcal{A}_X(\mathbf{1}^n, S_{n,s}|_X) \;:$$
$$Y \subseteq Z \wedge \mathcal{A}(\mathbf{1}^n, f|_Y) = B(c_n(S_{n,s}))\big] +$$
$$\sum_{X \in \overline{\mathcal{M}}(Y)} \Pr\big[S_{n,s} \leftarrow D_{n,s}, (Z,f) \leftarrow \mathcal{A}_X(\mathbf{1}^n, S_{n,s}|_X) : Y \not\subseteq Z \wedge \mathcal{B}(\mathbf{1}^n) = B(c_n(S_{n,s}))\big].$$

The lemma gives that the first summand is only negligibly different from

$$\Pr\big[S_{n,s} \leftarrow D_{n,s}, S_n \leftarrow [\![\mathsf{P}]\!](\mathcal{S})(S_{n,s}) \;:\; \mathcal{A}(\mathbf{1}^n, S_n|_Y) = B_n(c_n(S_{n,s}))\big] \ .$$

The second summand equals

$$\sum_{X \in \overline{\mathcal{M}}(Y)} \Pr\big[S_{n,s} \leftarrow D_{n,s} \;:\; \mathcal{B}(\mathbf{1}^n) = B(c_n(S_{n,s}))\big] -$$
$$\sum_{X \in \overline{\mathcal{M}}(Y)} \Pr\big[S_{n,s} \leftarrow D_{n,s}, (Z,f) \leftarrow \mathcal{A}_X(\mathbf{1}^n, S_{n,s}|_X) : Y \subseteq Z \wedge \mathcal{B}(\mathbf{1}^n) = B(c_n(S_{n,s}))\big],$$

where, according to the lemma, the second sum differs only negligibly from $\Pr\big[S_{n,s} \leftarrow D_{n,s} : \mathcal{B}(\mathbf{1}^n) = B(c_n(S_{n,s}))\big]$ (and the first is $|\overline{\mathcal{M}}(Y)| = l$ times that).

```
k1 := gen_key()
if b then
  k2 := k1
else
  k2 := gen_key()
x := enc(k1, y)
```

Knowing the equality of keys

```
k1 := gen_key()
if b then
  k2 := k1
else
  k2 := gen_key()
x1 := enc(k1, y1)
x2 := enc(k2, y2)
```

Knowing, which key was used

Fig. 3. Example programs

7 Discussion

We have presented an analysis that, for a certain class of programs, can decide whether these programs have computationally secure information flow. As an example consider the program consisting of a single statement `x := enc(k, y)`. Suppose that the variables `k` and `y` are secret and `x` is public. In this case, our analysis finds that the initial value of `y` can be recovered from the final value of `y` or from the final values of `x` and `k`. However, both of these possibilities require the knowledge of some secret output, and thus the program does not leak the value of `y`.

Our analysis reports, that nothing about the initial value of `y` can be found from the final value of `x`. On the other hand, the initial value of `y` obviously interferes with the final value of `x` and thus an analysis using non-interference as its security criterion must reject this program.

Statically knowing which keys are equal. We require that at all program points we can statically answer the question “Which variables of type ‘key’ are equal?” The reason for this requirement is that, by having a cryptotext and a key, it is usually possible to determine whether this cryptotext was produced with this key, because the encryption usually involves padding the message in a certain way, see, for example, [8]. At least our definitions allow this determination to be possible. As an example of using this information, see the program in the left of the Fig. 3. In this case, the adversary can find the initial value of `b` from the final values of `x` and `k2`, which our analysis does not reflect. The adversary has to check whether `x` was generated by encrypting with `k2` or not. The requirement placed on `E` guarantees that this extra information is already statically known.

Intuitively, it may seem that as `k2` is assigned to at statements whose execution is controlled by `b`, the variable `k2` should be recorded to be dependent on `b` after the `if-then-else`-statement. Actually `k2` is just a newly generated key at this place, thus, according to our semantics, it is not dependent on anything. On the other hand, *the pair* $\langle$`k1`, `k2`$\rangle$ depends on `b`. Unfortunately we are not yet able to handle such dependencies.

Still, we believe that even the class of programs satisfying all the constraints given in the beginning of Sec. 5 is interesting enough — and moreover, big enough — to contain realistic programs.

Requiring which-key and repetition concealedness. The question naturally arises, why do we require such properties from the cryptosystem that we use. The example in the right of Fig. 3 shows that which-key concealedness is indeed necessary (at least when we use the analysis presented in Sec. 5). If the encryption was not which-key concealing, then the adversary might be able to determine whether `x1` and `x2` were encrypted under the same key or not. This allows him to find the initial value of `b`. The analysis does not account for this. A similar example would show the necessity of repetition-concealedness. Although Abadi and Rogaway [2] show that which-key concealedness is not hard to achieve, it would be interesting to study what could be done (how should the analysis look like) with weaker encryption primitives — especially with ones that are not repetition-concealing.

Modeling secure information flow with noninterference. It is easy to write down an information-theoretic analogue to the definition of recoverability — it would look similar to Cohen's [3] strong dependency, especially to its deductive viewpoint. One can also give similar program analysis and prove the corresponding correctness result. The analysis $\mathcal{W}_{\mathrm{NI}}$ would look identical to $\mathcal{W}$; the only difference would be the absence of the special handling of encryption. The correctness proof would actually be simpler than the one presented in Sec. 6 — the lemma 1 would be easier to state, because the entities $\mathcal{A}_X$, $X \subseteq \mathbf{Var}$, would not be constrained to be PPT algorithms.

Comparison to Denning's and Myers's abstract semantics. Observing the analysis $\mathcal{W}_{\mathrm{NI}}$, we see that the minimal elements of $\mathcal{W}_{\mathrm{NI}}(N) \in \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}}))$ are all one-element sets. Thus, we could replace the set $\mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}}))$ with $\mathcal{P}(\widetilde{\mathbf{Var}})$ (by identifying the set $\mathsf{cl}_{\mathrm{U}}(\{\{x_1\},\dots,\{x_k\}\}) \in \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\widetilde{\mathbf{Var}}))$ with the set $\{x_1,\dots,x_k\} \in \mathcal{P}(\widetilde{\mathbf{Var}})$). We continue by replacing $\mathcal{P}(\widetilde{\mathbf{Var}})$ with $\widetilde{\mathbf{Var}} \rightarrow \{L, H\}$ by considering the characteristic functions $\mathcal{X}_{\{x_1,\dots,x_k\}} : \widetilde{\mathbf{Var}} \rightarrow \{0,1\}$ and identifying $0 \equiv L$, $1 \equiv H$. This gives us Denning's and Denning's [5] information flow analysis for the two-element lattice of security classes, also stated in their terms.

On covert flows. Our analysis is not termination-sensitive — it does not detect that a program's secret inputs may affect whether it terminates. Actually, termination is an issue if the security of the information flow is defined through noninterference. When using our definition, one needs to detect whether the program runs in polynomial time or not. Reitman and Andrews [12] have presented an axiomatic approach for checking the security of information flow, their method is termination sensitive and the way it has been made termination-sensitive is very explicit. Thus we should be able to apply their ideas to our analysis to make it termination-sensitive (or "superpolynomial-running-time-sensitive") as well. Another possible covert channel that we do not address is how long the program runs and what can be detected from its timing behaviour.

Acknowledgements. We are thankful to Reinhard Wilhelm and Thomas W. Reps. Their comments have certainly made this paper more readable.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A Core Calculus of Dependency. In proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, January 20–22, 1999, ACM Press, pp. 147–160, 1999.
2. Abadi, M., Rogaway, P.: Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.): proc. of the International Conference IFIP TCS 2000 Sendai, Japan, August 17-19, 2000 (LNCS 1872), Springer-Verlag pp. 3–22, 2000.
3. Cohen, E.: Information Transmission in Sequential Programs. In: DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J. (eds.): Foundations of Secure Computation. Academic Press, pp. 297–335. 1978.
4. Denning, D.E.: A Lattice Model of Secure Information Flow. Communications of the ACM **19**(5), pp. 236–243, 1976.
5. Denning, D.E., Denning, P.: Certification of Programs for Secure Information Flow. Communications of the ACM **20**(7), pp. 504–513, 1977.
6. Goldreich, O.: The Foundations of Modern Cryptography. In: Kaliski, B. (ed.): proc. of CRYPTO '97, Santa Barbara, CA, August 17–21, 1997 (LNCS 1294), Springer-Verlag, pp. 46–74, 1997.
7. Heintze, N., Riecke, J.G.: The SLam Calculus: Programming with Secrecy and Integrity. In proc. of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, January 19–21, 1998, ACM Press, pp. 365–377, 1998.
8. Kaliski, B., Staddon, J.: PKCS #1: RSA Cryptography Standard. Version 2.0. RSA Laboratories, September 1998.
9. Landi, W.: Interprocedural Aliasing in the Presence of Pointers. PhD thesis, Rutgers University, 1992.
10. Leino, K.R.M., Joshi, R.: A Semantic Approach to Secure Information Flow. In: Jeuring, J. (ed.): proc. of "Mathematics of Program Construction, MPC'98", Marstrand, Sweden, June 15–17, 1998 (LNCS 1422), Springer-Verlag, 254–271, 1998.
11. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, January 20–22, 1999, ACM Press, pp. 228–241, 1999.
12. Reitman, R.P., Andrews, G.R.: Certifying information flow properties of programs: an axiomatic approach. In proc. of the 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, TX, January 1979, ACM Press, pp. 283–290, 1979.
13. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. In: Swierstra, S.D. (ed.): proc. of the 8th European Symposium on Programming, ESOP'99, Amsterdam, The Netherlands, 22-28 March, 1999 (LNCS 1576), Springer-Verlag, pp. 40–58, 1999.
14. Volpano, D., Smith, G., Irvine, C.: A Sound Type System for Secure Flow Analysis. Journal of Computer Security **4**(2,3), pp. 167–187, 1996.
15. Volpano, D., Smith, G.: Verifying secrets and relative secrecy. In proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, MA, January 19–21, 2000, ACM Press, pp. 268–276, 2000.
16. Volpano, D.: Secure Introduction of One-way Functions. In proc. of the 13th IEEE Computer Security Foundations Workshop. Cambridge, UK, July 3–5, 2000.