Handling Encryption in Analyses for Secure Information Flow

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

ABSTRACT

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

1. INTRODUCTION

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

If one wants to prove correct an analysis checking programs for secure information flow, one has to formalise, when the public outputs of the program contain information about secret inputs that is useful for an adversary. The usual formalisation is *noninterference* [6], which states that the public outputs must not contain *any* information about the private inputs *at all*.

Consider a program that takes two inputs — an encryption key, k, and a message, M, and outputs $\mathcal{E}nc(k, M)$ — the encryption of M under key k. Obviously, $\mathcal{E}nc(k, M)$ contains information about M as it is in principle possible to find M from $\mathcal{E}nc(k, M)$. Hence, if M is secret and $\mathcal{E}nc(k, M)$ is public, then the noninterference property does not hold. On the other hand, the security of $\mathcal{E}nc$ requires that an adversary, whose resources (computation time and space) have certain bounds, cannot derive anything about M from $\mathcal{E}nc(k, M)$. If we consider only such bounded adversaries then we could deem this program secure. The bounds on working time (and space) of the adversaries are lax enough (probabilistic polynomial time) for all realistic adversaries to satisfy them. This motivated the definition of secure information flow we gave in [7]. We demanded that the public outputs of the program allow no adversary, working in probabilistic polynomial time, to deduce anything about the secret inputs that was not deducible without knowing the public outputs.

The contribution of the current paper is a new program analysis for secure information flow. It is based on the security definition of [7]. The analysis given in [7] had several restrictions on how the program was allowed to use the encryption keys. Basically, the only thing one could do with a key was to encrypt with it. A key could not be an operand of any other operation. This meant among other things that there was no way to decrypt anything. This also meant that a key could not be encrypted, i.e. it was impossible to have the expression "$\mathcal{E}nc(k, k')$" in the program. The analysis in [7] also failed to notice that one could check the equality of two keys by encrypting something with the first key and afterwards trying to decrypt it with the second one — the flow of information encoded in the (in)equality of the keys could remain unnoticed. Hence we required that the equality of encryption keys was always statically known. This ensured that no information could be recorded in the (in)equality of keys. Although one could argue that the set of analysable programs was still large enough to contain interesting programs, there was a desire to get rid of those restrictions. The analysis presented in the current paper fulfills that desire. We also show that this program analysis is in a certain sense at least as precise as the analysis in [7].

The abstract domain that the analysis uses is quite different from any other secure information flow analysis that we are aware of. The analysis records pairs of sets of variables, where the values of variables in the first set are *computationally independent* [20, Def. 18, Thm. 6] from the values of the variables in the second set. Thus the abstract domain is rather large (but nevertheless finite), so that the running time of a direct implementation becomes prohibitive. We consider the value of the analysis to mainly lie in cleanly abstracting from a complex concrete domain (a countable tuple of probabilistic functions; following the structure of

*Supported by Deutsche Forschungsgemeinschaft, under the Graduate Studies Program #623, "Quality Guarantees for Computer Systems"

the definition of secure encryption schemes) to a finite abstract domain. As such it will serve as a convenient starting point for further abstractions.

After describing some related work in Sec. 2 we introduce the notation used to define domains and functions between them in Sec. 3. The same section also defines the cryptographic notions used in the rest of the paper and attempts to give some intuition behind them. Sec. 4 presents the programming language, its semantics and the security definition, it mostly repeats [7]. Sec. 5 is devoted to introducing the abstract domain. The program analysis is given in Sec. 6 and proven correct in Sec. 7 with respect to the semantics and security definition given in Sec. 4. Finally, Sec. 8 shows that the new analysis favorably compares with the one given in [7].

2. RELATED WORK

Using program analysis for certification of secure information flow was pioneered by Denning and Denning [4, 5]. They annotated the program statements with the information flow between the variables caused by that statement, and analysed this flow. Volpano et al. [19] gave a definition of secure information flow and accompanying analysis without using any instrumentations.

Leino and Joshi [8] define a program to be secure if, no matter what its secret inputs are, the public outputs always look the same for the same non-secret inputs. "Looking the same" is not specified further, different security definitions can be obtained by plugging in different formalisations. The security definition that we are using can be seen as an instance of theirs.

Recently, some work has been done to define weaker notions of secure information flow which allow analysing programs containing cryptographic primitives without losing the precision one intuitively assigns to these primitives. Volpano and Smith [18, 17] have presented analyses of programs containing one-way functions as primitive operations. Unfortunately, one is quite restricted in using the one-way function if one wants to take advantage of the weakened security definition. Another approach has been our own [7], analysing programs containing encryption as a primitive operation, and having its own set of restrictions.

There have also been attempts to precisely formalise and analyse cryptographic protocols. Instead of the usual assumption that cryptographic primitives are perfectly secure—they are modeled as functions for which only a very restricted set of formulas holds ([3, 1] are among the most prominent examples), one attempts to take into account that the cryptographic primitives may be implemented by any algorithm satisfying some complex complexity-theoretical definition; dealing with those definitions is an issue that both the analyses for secure information flow and the analyses of cryptographic protocols must handle[1]. Mitchell et al. [9, 10, 11] extended the spi-calculus [1] with (polynomial) bounds on message lengths and execution time, and developed a probabilistic semantics for this extension. This has allowed them to prove the protocols correct with respect to polynomially bounded adversaries, where the cryptographic primitives that the protocols employ are real ones. These proofs are entirely hand-crafted, though; there are no mechanical means (like program analysis) to derive them. Pfitzmann et al. [13, 14, 15] have given a framework to faithfully abstract the cryptographic primitives, such that the proofs about protocols using these abstractions would also hold if the abstractions are replaced with the actual primitives. Abadi and Rogaway [2] have shown that the formal construction of messages from simpler ones by tupling and encryption is computationally justified—if two formal messages look the same (where "looking the same" is defined over the formal structure; it makes the contents of the encrypted submessages irrelevant), and if the encryption primitive satisfies certain requirements, then no polynomially bounded adversary can distinguish the actual representations of these messages as bit-strings.

[1]Note that while a program analysis correct with respect to noninterference is overly *pessimistic*, a protocol analysis assuming perfect cryptography is overly *optimistic*!

3. NOTATION AND PRELIMINARIES

3.1 Domains

For a set X, the set of all of its subsets is denoted by $\mathcal{P}(X)$. If X is partially ordered and $Y \subseteq X$, then Y is *lower* [resp. *upper*] *closed*, iff for all $x \in Y$ and $x' \leq x$ [resp. $x' \geq x$] holds $x' \in Y$. The set of all lower [resp. upper] closed subsets of X is denoted by $\mathcal{P}_{\mathrm{L}}(X)$ [resp. $\mathcal{P}_{\mathrm{U}}(X)$].

$X_\perp$ denotes the set $X \uplus \{\perp\}$ (here $\uplus$ denotes the disjoint union). $X_\perp$ is partially ordered — $\perp$ is the smallest element and the other elements are ordered as in X.

A *probability distribution* over a set X is a function $D : X \to [0..1]$, where $\sum_{x \in X} D(x) = 1$. Let $\mathcal{D}(X)$ denote the set of all probability distributions over X. For a probability distribution $D \in \mathcal{D}(X)$, the expression $x \leftarrow D$ denotes that the random variable x is distributed according to D. Multiset comprehensions, for example $\{\!|x \leftarrow D : f(x)|\!\}$, are used to define new probability distributions from existing ones. For $D \in \mathcal{D}(X)$ and $f : X \to Y$ the given example is a distribution $D' \in \mathcal{D}(Y)$ given by

$$\forall y \in Y : D'(y) = \sum_{x \in f^{-1}(y)} D(x) \ .$$

A partial order on X defines a partial order on $\mathcal{D}(X)$. $D_1 \leq D_2$, where $D_1, D_2 \in \mathcal{D}(X)$, iff

$$\forall X' \in \mathcal{P}_{\mathrm{U}}(X) : \sum_{x \in X'} D_1(x) \leq \sum_{x \in X'} D_2(x) \ .$$

Given a function from one domain to another one, we often want to talk about the "same" function working on a different, related domain. For this purpose we define a number of combinators transforming the types of functions. For a function $f : X \to \mathcal{D}(Y)$ let $ext^{\mathcal{D}}(f) : \mathcal{D}(X) \to \mathcal{D}(Y)$ be the function mapping a distribution $D \in \mathcal{D}(X)$ to $D' \in \mathcal{D}(Y)$, where

$$\forall y \in Y : D'(y) = \sum_{x \in X} D(x) \cdot (f(x))(y) \ .$$

ext is also defined for other domain constructors. If $f : X \to \mathcal{P}(Y)$ then $ext^{\mathcal{P}}(f)$ maps each set $Z \subseteq X$ to the set $\bigcup_{x \in Z} f(x)$. If $f : X \to Y_\perp$ then $ext^\perp(f)$ is identical to f, except that it additionally maps $\perp \in X_\perp$ to $\perp \in Y_\perp$. As a generalisation of $ext^\perp$, we define $\widehat{ext}^\perp(f) : X_\perp \to Y$ for each $f : X \to Y$, where the set Y has the smallest element. $\widehat{ext}^\perp(f)$ maps $\perp$ to this smallest element and behaves like f otherwise.

Let $\eta^{\mathcal{D}}(x) \in \mathcal{D}(X)$, where $x \in X$, denote the distribution where x occurs with the probability 1. For $f : X \to Y$ let $\mathcal{D}(f) : \mathcal{D}(X) \to \mathcal{D}(Y)$ denote the function $ext^{\mathcal{D}}(\eta^{\mathcal{D}} \circ f)$.

For a partially ordered set X let $\mathcal{R}(X)$ denote the same set with reversed order. For a family of sets $\{X_n\}_{n\in\mathbb{N}}$, let $\prod_{n\in\mathbb{N}} X_n$ denote their Cartesian product — the set of countable tuples, where the n-th component has the type X_n. For $f : \prod_{n\in\mathbb{N}}(X_n \to Y_n)$ and $x \in X_i$ let $f(x) \in Y_i$ denote the quantity $f_i(x)$. The application of $\mathcal{D}(\cdot)$, $ext^{\mathcal{D}}$, etc. on countable tuples is defined componentwise.

A partial order X is a complete lattice, if each subset of X has both greatest lower and least upper bounds. Given two complete lattices X and Y, a *Galois connection* (see also [12, Sec. 4.3]) between them is a pair of monotone functions $\alpha : X \to Y$ and $\gamma : Y \to X$, such that $\gamma \circ \alpha \geq id_X$ and $\alpha \circ \gamma \leq id_Y$, where id_X denotes the identity function on X and the functions are ordered pointwise. α is called the *abstraction function* and γ the *concretisation function*. For each $x \in X$ and $y \in Y$ the equalities

$$\gamma(y) = \bigvee\{x : x \in X, \alpha(x) \leq y\}$$
$$\alpha(x) = \bigwedge\{y : y \in Y, \gamma(y) \geq x\}$$

hold. Thus α uniquely determines γ and vice versa.

3.2 Cryptography

Two families of probability distributions $D = \{D_n\}_{n\in\mathbb{N}}$, $D' = \{D'_n\}_{n\in\mathbb{N}}$, where $D_n, D'_n \in \mathcal{D}(\{0,1\}^{\ell(n)})$ are *computationally indistinguishable* if for all PPT (*Probabilistic Polynomial-Time*) algorithms $\mathcal{A}$ the difference

$$\Pr\big[x \leftarrow D_n : \mathcal{A}(\mathbf{1}^n, x) = 1\big] - \Pr\big[x \leftarrow D'_n : \mathcal{A}(\mathbf{1}^n, x) = 1\big]$$

is negligible in n. The intuition behind this definition is the following: the algorithm $\mathcal{A}$ may find itself in one of two possible worlds. In the first world the variable x is distributed according to D, in the second world according to D'. The algorithm $\mathcal{A}$ outputs 1 if it guesses that it is in the first world, it outputs something else if it guesses that it is in the second world. The definition now states that those two worlds are so similar that the guess made by $\mathcal{A}$ does not significantly depend on the world it is placed into.

The notion of computational indistinguishability can also be defined for distributions over sets of other kinds of objects (not only for bit-strings), if it has been fixed, how the algorithm $\mathcal{A}$ is given access to those objects. As an important case, we give the definition of computational indistinguishability of distributions of (probabilistic) functions, whose domain and range are both sets of bit-strings.

Two families of probability distributions $D = \{D_n\}_{n\in\mathbb{N}}$, $D' = \{D'_n\}_{n\in\mathbb{N}}$, where $D_n, D'_n \in \mathcal{D}(\{0,1\}^{\ell(n)} \to \mathcal{D}(\{0,1\}^{\ell(n)}))$ are *computationally indistinguishable* if for all PPT algorithms with an oracle $\mathcal{A}^{(\cdot)}$ the difference

$$\Pr\big[f \leftarrow D_n : \mathcal{A}^{f(\cdot)}(\mathbf{1}^n) = 1\big] - \Pr\big[f \leftarrow D'_n : \mathcal{A}^{f(\cdot)}(\mathbf{1}^n) = 1\big]$$

is negligible in n.

An encryption scheme is a triple consisting of the *key-generation* algorithm $\mathcal{G}$, the *encryption* algorithm $\mathcal{E}$ and the *decryption* algorithm $\mathcal{D}$ (all working in PPT), where

- On input $\mathbf{1}^n$ (the bit-string, consisting of n 1-bits), $\mathcal{G}$ (randomly; according to its internal coin-tosses) outputs a bit-string k. $\mathcal{G}(\mathbf{1}^n)$ thus defines a probability distribution over the bit-strings.
- For all keys k that $\mathcal{G}(\mathbf{1}^n)$ may output and messages $x \in \mathsf{PlainText}_n \subseteq \{0,1\}^{\ell(n)}$ holds $\mathcal{D}(\mathbf{1}^n, k, \mathcal{E}(\mathbf{1}^n, k, x)) = x$.

Here n is the *security parameter* (bigger n means potentially longer keys and therefore more security) and $\ell \in \mathbb{Z}[n]$.

An encryption scheme is *repetition concealing* [2] if for all PPT algorithms $\mathcal{A}^{(\cdot)}$ the difference

$$\Pr\big[k \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n) = 1\big] - \Pr\big[k \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})}(\mathbf{1}^n) = 1\big] \quad (1)$$

is negligible in n.[2] It is *which-key concealing* if for all PPT algorithms $\mathcal{A}^{(\cdot),(\cdot)}$ the difference

$$\Pr\big[k, k' \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k',\cdot)}(\mathbf{1}^n) = 1\big] - \Pr\big[k \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n) = 1\big] \quad (2)$$

is negligible in n. We require that the encrpytion scheme that the analysed programs use is both repetition concealing and which-key concealing.

Abadi and Rogaway [2] also define that an encryption scheme is *which-key and repetition concealing* if the difference

$$\Pr\big[k, k' \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k',\cdot)}(\mathbf{1}^n) = 1\big] - \Pr\big[k \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|}),\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})}(\mathbf{1}^n) = 1\big] \quad (3)$$

is negligible for all PPT algorithms $\mathcal{A}^{(\cdot),(\cdot)}$. It turns out that an encryption scheme is which-key and repetition concealing iff it is which-key concealing and repetition concealing. We give the proof of this result in App. A, as we have been unable to find it in the literature.

Let $F = \{f_1, \ldots, f_k\}, G = \{g_1, \ldots, g_l\}$ be two sets of families of functions, where the type of f_i and g_j is $\prod_{n\in\mathbb{N}}(\mathcal{X}_n \to (\{0,1\}^* \to \mathcal{D}(\{0,1\}^*)))$, where $\mathcal{X} = \{\mathcal{X}_n\}_{n\in\mathbb{N}}$ is a family of sets. Let $D = \{D_n\}_{n\in\mathbb{N}}$ where D_n is a probability distribution over $\mathcal{X}_n$. We say that F and G are *computationally independent* in D if the distributions

$$\{\!|x \leftarrow D_n : (f_{1,n}(x), \ldots, f_{k,n}(x), g_{1,n}(x), \ldots, g_{l,n}(x))|\!\}$$

and

$$\{\!|x, y \leftarrow D_n : (f_{1,n}(x), \ldots, f_{k,n}(x), g_{1,n}(y), \ldots, g_{l,n}(y))|\!\}$$

[2] For the intuition: it is just an instance of the computational indistinguishability of the families of probability distributions over probabilistic functions.

are computationally indistinguishable, i.e. for all PPT algorithms $\mathcal{A}$ (with $|F|+|G|$ oracles) the difference

$$\Pr\big[x \leftarrow D_n : \mathcal{A}^{f_{1,n}(x)(\cdot),\ldots,f_{k,n}(x)(\cdot),g_{1,n}(x)(\cdot),\ldots,g_{l,n}(x)(\cdot)}(\mathbf{1}^n)=1\big]-$$
$$\Pr\big[x \leftarrow D_n, y \leftarrow D_n : \mathcal{A}^{f_{1,n}(x)(\cdot),\ldots,f_{k,n}(x)(\cdot),g_{1,n}(y)(\cdot),\ldots,g_{l,n}(y)(\cdot)}(\mathbf{1}^n)=1\big]$$

must be negligible in n. We are mostly going to apply this definition in the case where $\mathcal{X}_n = \mathcal{Y}_{n\perp}$ (and $\mathcal{Y}_n$ is the set of program states). In this case $\perp$ will denote nontermination (or running longer than a predetermined amount of time) and the distribution D is implemented by an algorithm that either returns an element of $\mathcal{Y}_n$ or runs for a too long time. If the algorithm implementing D runs for a too long time then we want that the algorithm $\mathcal{A}$ does not get a chance to run at all. We capture this by loosening the definition of computational independence a little. We require that

$$\Pr\big[x \leftarrow D, y \leftarrow D : x \neq \perp \wedge y \neq \perp \wedge \mathcal{A}^{f_{1,n}(x)(\cdot),\ldots,f_{k,n}(x)(\cdot),g_{1,n}(x)(\cdot),\ldots,g_{l,n}(x)(\cdot)}(\mathbf{1}^n)=1\big]-$$
$$\Pr\big[x \leftarrow D, y \leftarrow D : x \neq \perp \wedge y \neq \perp \wedge \mathcal{A}^{f_{1,n}(x)(\cdot),\ldots,f_{k,n}(x)(\cdot),g_{1,n}(y)(\cdot),\ldots,g_{l,n}(y)(\cdot)}(\mathbf{1}^n)=1\big]$$

is negligible in n. Thus $\perp$ is indistinguishable from everything else.

4. SECURITY DEFINITION

We consider a simple imperative programming language. Let **Var** be the set of variables and **Op** be the set of arithmetic, relational, boolean, etc. operators. The syntax of programs is given by the following grammar:

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

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 are handling encryption, thus we must have a binary operator $\mathcal{E}nc \in \mathbf{Op}$ that denotes the encryption operation. We also need a nullary key generation operator $\mathcal{G}en \in \mathbf{Op}$ that is used to create "good" keys. The analysis handles these operators differently from the other ones, taking into account that they are expressing secure encryption[3]. This special handling requires that $(\llbracket\mathcal{G}en\rrbracket, \llbracket\mathcal{E}nc\rrbracket)$ constitutes a which-key and repetition concealing encryption scheme.

The concrete semantics of programs is denotational — it is a function mapping the inputs of the program to its outputs. In [7] we specified that the type of the semantics $\mathcal{C}_{\mathrm{Path}}\llbracket\mathsf{P}\rrbracket$ of the program P is (an equivalent of) $\prod_{n\in\mathbb{N}}(\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbf{Path}_{\mathsf{P}})_\perp))$. Here

- n is the security parameter. Similarly to the encryption scheme, the semantics is also parametrised by it.
- $\mathbf{State}_n = \mathbf{Var} \to \mathbf{Val}_n$, where $\mathbf{Val}_n = \{0,1\}^{\ell(n)}$.

[3]Decryption does not have to be handled separately, thus it was not mentioned here.

- $\mathbf{Path}_{\mathsf{P}}$ is the set of all paths from start to end in the flowchart of the program P.

The set of paths was introduced to keep track of the computation length — it is obvious that when talking about the computational security of information flow, we must only consider program runs taking "polynomial" time, thus the computation length is an issue. Taking $\prod_{n\in\mathbb{N}}(\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp))$ as the type of the semantics (the semantics maps each initial state to a probability distribution over tuples $\langle$final state, computation length$\rangle$, with a possibility for nontermination) would be the straightforward way to introduce computation lengths into semantics. In [7] we were more exact, by recording the entire execution path, instead of only its length. In the current paper we are going to use the semantics that simply records the computation length and denote it by $\mathcal{C}_{\mathrm{len}}\llbracket\mathsf{P}\rrbracket$.

To define the semantics $\mathcal{C}_{\mathrm{len}}\llbracket\mathsf{P}\rrbracket$ we first need semantics for each operator $o \in \mathbf{Op}$. For an operator o of arity k, the semantics is a family of functions $\llbracket o\rrbracket : \prod_{n\in\mathbb{N}}(\mathbf{Val}_n^k \to \mathbf{Val}_n)$. The semantics $\mathcal{C}_{\mathrm{len}}\llbracket\mathsf{P}\rrbracket$ is defined in Fig. 1.

For using the information about the computation length define for each $q \in \mathbb{Z}[n]$ a family of functions $\hat{\pi}^{(q)} : \prod_{n\in\mathbb{N}}(\mathbf{State}_n \times \mathbb{N} \to \mathbf{State}_{n\perp})$ where

$$(\hat{\pi}^{(q)})_n((S_n,l)) = \begin{cases} S_n, & \text{if } l \leq q(n) \\ \perp, & \text{if } l > q(n) \end{cases}$$

and define the semantics $\mathcal{C}_{\mathrm{len}}^{(q)}\llbracket\mathsf{P}\rrbracket$, having the type $\prod_{n\in\mathbb{N}}(\mathbf{State}_n \to \mathcal{D}(\mathbf{State}_{n\perp}))$, by

$$\mathcal{C}_{\mathrm{len}}^{(q)}\llbracket\mathsf{P}\rrbracket := \mathcal{D}(\mathit{ext}^\perp(\hat{\pi}^{(q)})) \circ \mathcal{C}_{\mathrm{len}}\llbracket\mathsf{P}\rrbracket\ .$$

Let $X, Y \subseteq \mathbf{Var}$. We say that the initial values of the variables in X are *irrecoverable* from the final values of the variables in Y, given that the initial state of the program is distributed according to $D_s = \{D_{n,s}\}_{n\in\mathbb{N}}$, $D_{n,s} \in \mathcal{D}(\mathbf{State}_n)$, if for each polynomial-time computable predicate $B = \{B_n\}_{n\in\mathbb{N}}$, $B_n : (X \to \mathbf{Val}_n) \to \{0,1\}$, each $q \in \mathbb{Z}[n]$ and each PPT algorithm $\mathcal{A}$ there exists a PPT algorithm $\mathcal{B}$, such that

$$\Pr\big[S_{n,s} \leftarrow D_{n,s}, S_n \leftarrow \mathcal{C}_{\mathrm{len}}^{(q)}\llbracket\mathsf{P}\rrbracket(S_{n,s}) : S_n \neq \perp \wedge \mathcal{A}(\mathbf{1}^n, S_n|_Y) = B_n(S_{n,s}|_X)\big]-$$
$$\Pr\big[S_{n,s} \leftarrow D_{n,s}, S_n \leftarrow \mathcal{C}_{\mathrm{len}}^{(q)}\llbracket\mathsf{P}\rrbracket(S_{n,s}) : S_n \neq \perp \wedge \mathcal{B}(\mathbf{1}^n) = B_n(S_{n,s}|_X)\big] \quad (4)$$

is negligible in n.

We have departed here from [7] by presenting a termination-*in*sensitive irrecoverability definition. Insensitivity to termination is achieved by not letting the algorithm $\mathcal{A}$ run if the program P runs too long (the condition $S_n \neq \perp$). Termination-sensitive and -insensitive definitions are not very different, though. The program P could be transformed by running a clock "parallel" (i.e. interleaved) to it and terminating the program if the value of the clock becomes too large (larger than a polynomial in the security parameter). Such a modified program always terminates in polynomial

$$\boxed{\mathcal{C}_{\mathrm{len}}[\![x := o(x_1,\ldots,x_k)]\!]_n(S_n) = \{\!|x_{\mathrm{val}} \leftarrow [\![o]\!]_n(S_n(x_1),\ldots,S_n(x_k)) \;:\; (S_n[x \mapsto x_{\mathrm{val}}], 1)|\!\}} \;.$$

$$\boxed{\mathcal{C}_{\mathrm{len}}[\![skip]\!]_n(S_n) = \eta^{\mathcal{D}}((S_n, 0))} \;.$$

$$\boxed{\mathcal{C}_{\mathrm{len}}[\![\mathsf{P}_1;\mathsf{P}_2]\!]_n = \mathrm{compose}_n(\mathcal{C}_{\mathrm{len}}[\![\mathsf{P}_1]\!]_n, \mathcal{C}_{\mathrm{len}}[\![\mathsf{P}_2]\!]_n)}$$

where $\mathrm{compose}_n : (\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp)) \times (\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp)) \to (\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp))$ is defined as

$$\mathrm{compose}_n(\mathsf{S}_{1,n}, \mathsf{S}_{2,n})(S_n) = \{\!|SP_1 \leftarrow \mathsf{S}_{1,n}(S_n), SP_2 \leftarrow \langle\!\langle SP_1 \stackrel{?}{=} \perp \;\mathbf{?}\; \eta^{\mathcal{D}}(\perp) \,\mathbf{:}\, \mathsf{S}_{2,n}(\pi_1(SP_1))\rangle\!\rangle \;:\; \langle\!\langle SP_2 \stackrel{?}{=} \perp \;\mathbf{?}\; \perp \mathbf{:}\, (\pi_1(SP_2), \pi_2(SP_1) + \pi_2(SP_2))\rangle\!\rangle|\!\}$$

where $\langle\!\langle b \;\mathbf{?}\; x \,\mathbf{:}\, y\rangle\!\rangle$ is equal to x if b is true, and is equal to y otherwise.

$$\boxed{\mathcal{C}_{\mathrm{len}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2]\!]_n = \mathrm{cond}_n(b, \mathcal{C}_{\mathrm{len}}[\![\mathsf{P}_1]\!]_n, \mathcal{C}_{\mathrm{len}}[\![\mathsf{P}_2]\!]_n)}$$

where $\mathrm{cond}_n : \mathbf{Var} \times (\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp)) \times (\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp)) \to (\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp))$ is defined as

$$\mathrm{cond}_n(b, \mathsf{S}_{1,n}, \mathsf{S}_{2,n})(S_n) = \{\!|SP \leftarrow \langle\!\langle S_n(b) \;\mathbf{?}\; \mathsf{S}_{1,n}(S_n) \,\mathbf{:}\, \mathsf{S}_{2,n}(S_n)\rangle\!\rangle \;:\; \langle\!\langle SP \stackrel{?}{=} \perp \;\mathbf{?}\; \perp \mathbf{:}\, (\pi_1(SP), \pi_2(SP) + 1)\rangle\!\rangle|\!\} \;.$$

$$\boxed{\mathcal{C}_{\mathrm{len}}[\![\mathit{while}\ b\ \mathit{do}\ \mathsf{P}]\!] = \mathit{lfp}\ F}$$

where $F : (\prod_{n\in\mathbb{N}}(\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp))) \to (\prod_{n\in\mathbb{N}}(\mathbf{State}_n \to \mathcal{D}((\mathbf{State}_n \times \mathbb{N})_\perp)))$ is defined by

$$[F(\mathsf{S})]_n = \mathrm{cond}_n(b, \mathrm{compose}_n(\mathcal{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, \mathsf{S}_n), \mathcal{C}_{\mathrm{len}}[\![skip]\!]_n) \;.$$

Figure 1: The semantics $\mathcal{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$

time and the termination-insensitive irrecoverability for it is equal to the termination-sensitive irrecoverability for the original program.

If $q_1 \le q_2$ where $q_1, q_2 \in \mathbb{Z}[n]$ and the order on $\mathbb{Z}[n]$ is defined pointwise, then $\mathcal{C}^{(q_1)}_{\mathrm{len}}[\![\mathsf{P}]\!] \le \mathcal{C}^{(q_2)}_{\mathrm{len}}[\![\mathsf{P}]\!]$, where the order is defined by extending the ordering on $\mathcal{D}(\mathbf{State}_{n\perp})$ pointwise and componentwise.

5. ABSTRACTION

In [7] we assumed that we were given a set of secret input variables $\mathbf{Var}_{\mathrm{S}} \subseteq \mathbf{Var}$ and an initial probability distribution D_s. We then proceeded to record for each program point the sets of variables $Y \subseteq \mathbf{Var}$, such that the initial values of the variables in $\mathbf{Var}_{\mathrm{S}}$ might have been recoverable from the current values of the variables in Y. We were only able to do this if the variables containing encryption keys were completely independent from everything else and if the dependence pattern between different variables containing the keys was statically known. In the analysis that we are going to present, we additionally record those dependence patterns.

Actually, by finding the dependence patterns we already have determined whether the program has secure information flow. Namely, assume that the program never assigns to the variables in $\mathbf{Var}_{\mathrm{S}}$. This assumption obviously does not lessen the generality, as we can transform every program to such form by introducing new variables. Denoting the set of public output variables by $\mathbf{Var}_{\mathrm{P}}$, we are simply interested whether the values of the variables in $\mathbf{Var}_{\mathrm{S}}$ and the values of the variables in $\mathbf{Var}_{\mathrm{P}}$ are independent of each other at the end of the program. If they are independent, then the information flow is secure.

Let $\widetilde{\mathbf{Var}}$ be the following set of families of functions of type $\prod_{n\in\mathbb{N}}(\mathbf{State}_n \to (\{0,1\}^* \to \mathcal{D}(\{0,1\}^*)))$:

- $\overline{x}$ for $x \in \mathbf{Var}$. $\overline{x}$ applied to a state $S_n \in \mathbf{State}_n$ is the constant function that always returns the value $S_n(x)$.
- $\overline{\mathcal{E}nc(k,\cdot)}$ for $k \in \mathbf{Var}$. $\overline{\mathcal{E}nc(k,\cdot)}$ applied to a state $S_n \in \mathbf{State}_n$ returns the function that encrypts its argument under the key $S_n(k)$.

We now define a *representation function* [12, Sec. 4.1.2] β from $\prod_{n\in\mathbb{N}}\mathcal{D}(\mathbf{State}_{n\perp})$ to $\mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}}) \stackrel{\leftrightarrow}{\times} \mathcal{P}(\widetilde{\mathbf{Var}})))$, where $\stackrel{\leftrightarrow}{\times}$ denotes the set of *unordered* pairs. $\beta(D)$, where $D \in \prod_{n\in\mathbb{N}}\mathcal{D}(\mathbf{State}_{n\perp})$, contains all those pairs of sets $(X, Y) \in \mathcal{P}(\widetilde{\mathbf{Var}}) \stackrel{\leftrightarrow}{\times} \mathcal{P}(\widetilde{\mathbf{Var}})$, where X and Y are computationally independent in D. Clearly $\beta(D)$ is lower closed: if $X' \subseteq X$ and $Y' \subseteq Y$ and there is an algorithm showing the dependence of X' and Y', then the same algorithm can be used to show the dependence of X and Y. The function β is monotone. The proof of this statement is given in App. A.

The representation function β gives us a Galois connection between $\mathcal{P}_{\mathrm{L}}(\prod_{n\in\mathbb{N}}\mathcal{D}(\mathbf{State}_{n\perp}))$ and $\mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}}) \stackrel{\leftrightarrow}{\times} \mathcal{P}(\widetilde{\mathbf{Var}})))$. For a set of distributions $\mathfrak{D}$ define $\alpha(\mathfrak{D}) := \bigcap_{D\in\mathfrak{D}} \beta(D)$.

Using $\mathcal{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ we define one more semantic function $\mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!]$

whose domain and range are both $\mathcal{P}_{\mathrm{L}}(\prod_{n\in\mathbb{N}}\mathcal{D}(\mathbf{State}_{n\perp}))$, as follows:

$$\mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!] := ext^{\mathcal{P}_{\mathrm{L}}}\Big(\lambda D.\mathsf{cl}_{\mathrm{L}}\Big(\{q\in\mathbb{Z}[n] : \\ ((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!]))(D)\}\Big)\Big) \quad . \quad (5)$$

Here $D\in\prod_{n\in\mathbb{N}}(\mathcal{D}(\mathbf{State}_{n\perp}))$. The argument of $\mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!]$ is a set of initial probability distributions. Each of the functions $\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!]$ (appropriately extended) is applied to each initial probability distribution and the set of resulting final probability distributions is taken.

Proposition 1. *Let P be a program, $X,Y\subseteq\mathbf{Var}$ and $D_s\in\prod_{n\in\mathbb{N}}(\mathcal{D}(\mathbf{State}_n))$. Assume that the program P does not assign to the variables in X. If*

$$(\overline{X},\overline{Y})\in\alpha(\mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!](\mathsf{cl}_{\mathrm{L}}\{D_s\})), \quad (6)$$

where $\overline{X}:=\{x\in X:\overline{x}\}$, then the initial values of the variables in X are irrecoverable from the final values of the variables in Y, given that the initial state of the program is distributed according to D_s.

Proof. From the definitions of α and $\mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!]$ follows that (6) holds iff for all $q\in\mathbb{Z}[n]$ the sets of variables X and Y are independent of each other in the distribution

$$\{\!|S_n : S_{n,s}\leftarrow D_{n,s}, S_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s})|\!\} \quad . \quad (7)$$

I.e. for all PPT algorithms $\dot{\mathcal{A}}$ the difference of probabilities

$$\begin{aligned}&\Pr\big[S_{n,s}\leftarrow D_{n,s}, S'_{n,s}\leftarrow D_{n,s},\\ &\quad S_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}), S'_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}) :\\ &\qquad S_n\neq\perp\wedge S'_n\neq\perp\wedge\dot{\mathcal{A}}(\mathbf{1}^n,S_n|_X,S_n|_Y)=1\big]-\\ &\Pr\big[S_{n,s}\leftarrow D_{n,s}, S'_{n,s}\leftarrow D_{n,s},\\ &\quad S_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}), S'_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}) :\\ &\qquad S_n\neq\perp\wedge S'_n\neq\perp\wedge\dot{\mathcal{A}}(\mathbf{1}^n,S_n|_X,S'_n|_Y)=1\big] \quad (8)\end{aligned}$$

is negligible in n. In particular, this holds when the algorithm $\dot{\mathcal{A}}$ is defined as follows: let $\mathcal{A}$ be any PPT algorithm and let $B=\{B_n\}_{n\in\mathbb{N}}$, $B_n:(X\to\mathbf{Val}_n)\to\{0,1\}$, be a polynomial-time computable predicate. $\dot{\mathcal{A}}(\mathbf{1}^n,\mathcal{X},\mathcal{Y})$ applies the algorithm $\mathcal{A}$ to $\mathcal{Y}$ and the algorithm computing B to $\mathcal{X}$ and returns whether the two returned values are equal (compare that definition with (4)). Let $\mathcal{B}$ denote the following algorithm: $\mathcal{B}(\mathbf{1}^n)$ generates a state S'_n according to the distribution (7) (it is polynomially constructible). If $S'_n\neq\perp$ then $\mathcal{B}$ applies $\mathcal{A}$ to $S'_n|_Y$ and returns whatever it returns. Otherwise $\mathcal{B}$ returns "failure". Clearly (8) is equal to

$$\begin{aligned}&\Pr\big[S_{n,s}\leftarrow D_{n,s}, S'_{n,s}\leftarrow D_{n,s},\\ &\quad S_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}), S'_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}) :\\ &\qquad S_n\neq\perp\wedge S'_n\neq\perp\wedge\mathcal{A}(\mathbf{1}^n,S_n|_Y)=B_n(S_n|_X)\big]-\\ &\Pr\big[S_{n,s}\leftarrow D_{n,s}, S'_{n,s}\leftarrow D_{n,s},\\ &\quad S_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}), S'_n\leftarrow\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!](S_{n,s}) :\\ &\qquad S_n\neq\perp\wedge S'_n\neq\perp\wedge\mathcal{B}(\mathbf{1}^n)=B_n(S_n|_X)\big] \quad .\end{aligned}$$

This difference of probabilities is negligible iff (4) is negligible (note that $S_n|_X=S_{n,s}|_X$, because the program does not assign to variables in X) or (7) lays only negligible weight on elements different from $\perp$. In the first case we are done, in the second case nothing about the initial values of variables is recoverable from their final values. $\square$

Note that the reverse claim does not hold. For example, after executing the program $x:=\mathcal{E}nc(k,y)$, where in the initial distribution the variables k and y are independent, the variables x and k do not have to be independent — if we are given a cryptotext and a key then we may be able to determine[4] whether this cryptotext was created with this key or not. On the other hand, x may not help computing the value of any predicate over k, otherwise the encryption scheme would not be which-key concealing. But see also Sec. 8.

The abstract semantics $\mathcal{W}[\![\mathsf{P}]\!]$ depends on the assumed set of variables $\mathbf{Var}$ (and the associated set $\widetilde{\mathbf{Var}}$) of the program P. This assumed set obviously has to contain all variables that are mentioned in P, but it could also contain other variables. In giving the analysis of P we need to consider the abstract semantices of its subprograms with respect to different sets of variables. Hence we make the assumed set of variables explicit: $\mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!]$.

6. ANALYSIS

In this section we define an abstract semantics $\mathcal{W}[\![\mathsf{P}]\!]$ whose domain and range are both $\mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}})\,\overline{\times}\,\mathcal{P}(\widetilde{\mathbf{Var}})))$. It will have the property $\mathcal{W}[\![\mathsf{P}]\!]\circ\alpha\geq\alpha\circ\mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!]$. The abstract semantics does not use the structure of the elements of the set $\widetilde{\mathbf{Var}}$, it simply treats them as names, thus it is operating over a finite structure.

In [7, Sec. 5] we stated several constraints that the programs had to obey to be analysable by the analysis given there. Some of these constraints still have to hold for the analysis we are going to present, but none of them could be considered as shortcomings of the analysis.

- For each operator $o\in\mathbf{Op}$, the semantics $[\![o]\!]$ of o must be computable in polynomial time — there must exist a PPT algorithm $\mathcal{O}$, such that $[\![o]\!]_n(\cdot)=\mathcal{O}(\mathbf{1}^n,\cdot)$ for all $n\in\mathbb{N}$ (we require the equality of probability distributions). This requirement ensures that the concrete semantics of the program is computable in polynomial time. Obviously we cannot analyse programs whose semantics is not polynomial-time computable.

- The initial probability distribution D_s, according to which the inputs of the program are distributed, must be polynomial time constructible — there must exist a PPT algorithm $\mathcal{D}$, such that $D_{n,s}=\mathcal{D}(\mathbf{1}^n)$. The clarification is similar to the previous constraint — we cannot account for effects uncomputable in polynomial time.

- At a statement $x:=\mathcal{E}nc(k,y)$, the variable k must contain a value that was created with the $\mathcal{G}en$-operator. Our analysis does not check that this requirement holds.

[4] the definitions of secure encryption do not demand the opposite

$$\frac{\begin{array}{c}(X,Y)\in\mathcal{X}\\ \overline{x},\overline{\mathcal{E}nc(x,\cdot)}\notin X\cup Y\end{array}}{(X,Y)\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=o(x_1,\ldots,x_k)\rrbracket(\mathcal{X})} \tag{9}$$

$$\frac{\begin{array}{c}(X,Y)\in\mathcal{X}\\ \overline{x},\overline{\mathcal{E}nc(x,\cdot)}\notin X\cup Y\\ O:=\{\overline{x_1},\ldots,\overline{x_k}\}\backslash Y\\ Z\subseteq Y\\ (Z\cup O,X\cup(Y\backslash Z))\in\mathcal{X}\end{array}}{(X,Y\cup\{\overline{x}\})\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=o(x_1,\ldots,x_k)\rrbracket(\mathcal{X})} \tag{10}$$

$$\frac{\begin{array}{c}(X,Y)\in\mathcal{X}\\ \overline{x},\overline{\mathcal{E}nc(x,\cdot)}\notin X\cup Y\\ O:=\{\overline{y},\overline{\mathcal{E}nc(k,\cdot)}\}\backslash Y\\ Z\subseteq Y\\ (Z\cup O,X\cup(Y\backslash Z))\in\mathcal{X}\end{array}}{(X,Y\cup\{\overline{x}\})\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=\mathcal{E}nc(k,y)\rrbracket(\mathcal{X})} \tag{10'}$$

$$\frac{\begin{array}{c}(X,Y)\in\mathcal{X}\\ \overline{x}\notin X\cup Y\end{array}}{(X,Y\cup\{\overline{x},\overline{\mathcal{E}nc(x,\cdot)}\})\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=\mathcal{G}en()\rrbracket(\mathcal{X})} \tag{10''}$$

$$\frac{\begin{array}{c}(X,Y)\in\mathcal{X}\\ \overline{x},\overline{\mathcal{E}nc(x,\cdot)}\notin X\cup Y\\ (\{\overline{\mathcal{E}nc(k,\cdot)}\},X\cup(Y\backslash\{\overline{\mathcal{E}nc(k,\cdot)}\})\cup\{\overline{y}\})\in\mathcal{X}\end{array}}{(X,Y\cup\{\overline{x}\})\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=\mathcal{E}nc(k,y)\rrbracket(\mathcal{X})} \tag{11}$$

$$\frac{\begin{array}{c}(X,Y)\in\mathcal{X}\\ \overline{x}\notin X\cup Y\end{array}}{\begin{array}{l}(X\cup\{\overline{\mathcal{E}nc(x,\cdot)}\},Y\cup\{\overline{\mathcal{E}nc(x,\cdot)}\})\\ \qquad\qquad\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=\mathcal{G}en()\rrbracket(\mathcal{X})\end{array}} \tag{12}$$

$$\frac{\begin{array}{c}(X,Y)\in\mathcal{X}\\ \overline{x},\overline{\mathcal{E}nc(x,\cdot)}\notin X\cup Y\\ \llbracket o\rrbracket \text{ is deterministic}\end{array}}{(X\cup\{\overline{x}\},Y\cup\{\overline{x}\})\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=o()\rrbracket(\mathcal{X})} \tag{13}$$

Figure 2: The semantics $\mathcal{W}_{\mathbf{Var}}\llbracket\cdot\rrbracket$ for assignments

> It would be easy to check it by introducing a simple type system having just two possible types — "key" and "everything". The operator $\mathcal{G}en$ produces values of type "key" and the type of the first argument of $\mathcal{E}nc$ must be "key". The rationale of this requirement is, that one must ensure that the key one is using for the encryption is a good one.

We also require that at statements $x := o(x_1,\ldots,x_k)$, the variable x is not one of the variables $x_1,\ldots,x_k$. This requirement obviously does not lessen the generality and its purpose is to make defining $\mathcal{W}\llbracket\mathsf{P}\rrbracket$ easier.

The abstract semantics of assignments (together with special cases for encryption and key generation) is given in Fig. 2. The specification has the form

$$\frac{\begin{array}{c}\text{condition}_1\\ \ldots\\ \text{condition}_l\end{array}}{(X,Y)\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=o(x_1,\ldots,x_k)\rrbracket}$$

where $\text{condition}_1,\ldots,\text{condition}_l$ are sufficient conditions for $(X,Y)\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=o(x_1,\ldots,x_k)\rrbracket(\mathcal{X})$ to hold. The free variables in the conditions are bound by existential quantification. If $(X,Y)\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=o(x_1,\ldots,x_k)\rrbracket(\mathcal{X})$, $X'\subseteq X$ and $Y'\subseteq Y$ then $(X',Y')\in\mathcal{W}_{\mathbf{Var}}\llbracket x:=o(x_1,\ldots,x_k)\rrbracket(\mathcal{X})$, too; this has not been made explicit in Fig. 2. Note that the general cases for $x := o(x_1,\ldots,x_k)$ also hold for $\mathcal{E}nc$ and $\mathcal{G}en$. Sec. 7 is devoted to proving the analysis correct.

$\mathcal{W}_{\mathbf{Var}}\llbracket \mathit{skip}\rrbracket$ is obviously the identity function on the carrier of the analysis $\mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}})\mathbin{\overset{\leftrightarrow}{\times}}\mathcal{P}(\widetilde{\mathbf{Var}})))$. Equally obviously, $\mathcal{W}_{\mathbf{Var}}\llbracket\mathsf{P}_1;\mathsf{P}_2\rrbracket=\mathcal{W}_{\mathbf{Var}}\llbracket\mathsf{P}_2\rrbracket\circ\mathcal{W}_{\mathbf{Var}}\llbracket\mathsf{P}_1\rrbracket$.

Next we define $\mathcal{W}_{\mathbf{Var}}\llbracket \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2\rrbracket$. Let N be a new variable. Denote $\mathcal{W}_1=\mathcal{W}_{\mathbf{Var}\uplus\{N\}}\llbracket N:=b;\mathsf{P}_1\rrbracket$ and $\mathcal{W}_2=\mathcal{W}_{\mathbf{Var}\uplus\{N\}}\llbracket N:=b;\mathsf{P}_2\rrbracket$. Let $\mathbf{Var}_{\mathrm{asgn}}\subseteq\mathbf{Var}$ be the set of variables that are assigned to in either P_1 or P_2 (or both). Let $X\subseteq\overline{\mathbf{Var}_{\mathrm{asgn}}}$ and $X',Y'\subseteq(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}_{\mathrm{asgn}}})$. We have $(X\cup X',Y')\in\mathcal{W}_{\mathbf{Var}}\llbracket \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2\rrbracket(\mathcal{X})$, if $(\{\overline{N}\}\cup X\cup X',Y')$ belongs to both $\mathcal{W}_1(\mathcal{X})$ and $\mathcal{W}_2(\mathcal{X})$, or if $X=\emptyset$ and $(X',Y')\in\mathcal{X}$.

$\mathcal{W}_{\mathbf{Var}}\llbracket \mathit{while}\ b\ \mathit{do}\ \mathsf{P}\rrbracket$ is defined in the following way. For a set of variables $\mathbf{Var}'\supseteq\mathbf{Var}$ and $N\in\mathbf{Var}'$ define the functions $\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}'}$, where $i\in\mathbb{N}$, whose domain and range are both $\mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}'})\mathbin{\overset{\leftrightarrow}{\times}}\mathcal{P}(\widetilde{\mathbf{Var}'})))$:

$$\mathcal{S}^{\mathsf{P},N}_{0,\mathbf{Var}'}=\mathcal{W}_{\mathbf{Var}'}\llbracket N:=\varphi(b,N)\rrbracket$$
$$\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}'}=\mathcal{S}^{\mathsf{P},N}_{0,\mathbf{Var}'}\vee(\mathcal{S}^{\mathsf{P},N}_{i-1,\mathbf{Var}'}\circ\mathcal{W}_{\mathbf{Var}'}\llbracket N:=\varphi(b,N);\mathsf{P}\rrbracket)\ .$$

Here φ is just a usual binary operator. Its task is to make N dependent of both b and N. Note that here the operation $\vee$ is pointwise *intersection.*

$\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}'}$ characterises the information flow of the program *while* b *do* P, where

- the information that is contained in the control flow of the program, is kept track of through the variable N;
- the loop is iterated at most i times.

Let $m\in\mathbb{N}$ be such that $\mathcal{S}^{\mathsf{P},N}_{m,\mathbf{Var}\uplus\{N\}}=\mathcal{S}^{\mathsf{P},N}_{m+1,\mathbf{Var}\uplus\{N\}}$. Such an m must exist because $\mathcal{S}^{\mathsf{P},N}_{i+1,\mathbf{Var}'}\geq\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}'}$ for all $i\in\mathbb{N}$ and they all belong to a finite set. We define

$$\begin{array}{l}\mathcal{W}_{\mathbf{Var}}\llbracket \mathit{while}\ b\ \mathit{do}\ \mathsf{P}\rrbracket(\mathcal{X})=\\ \{(X,Y)\,:\,X\subseteq\widetilde{\mathbf{Var}},Y\subseteq(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}_{\mathrm{asgn}}}),\\ (\{\overline{N}\}\cup X,Y)\in\mathcal{S}^{\mathsf{P},N}_{m,\mathbf{Var}\uplus\{N\}}(\mathcal{W}_{\mathbf{Var}\uplus\{N\}}\llbracket N:=\mathrm{const}\rrbracket(\mathcal{X}))\}\cup\\ \qquad(\mathcal{X}\cap(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}_{\mathrm{asgn}}})\mathbin{\overset{\leftrightarrow}{\times}}(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}_{\mathrm{asgn}}}))\ .\end{array}$$

7. CORRECTNESS PROOF

PROPOSITION 2. *For each program* P*, whose set of variables is* $\mathbf{Var}$*, satisfying the conditions at the beginning of Sec. 6, the inequality* $\mathcal{W}_{\mathbf{Var}}\llbracket\mathsf{P}\rrbracket\circ\alpha\geq\alpha\circ\mathcal{C}_{\mathrm{work}}\llbracket\mathsf{P}\rrbracket$ *holds, if the functions* $\mathcal{W}_{\mathbf{Var}}\llbracket\mathsf{P}\rrbracket\circ\alpha$ *and* $\alpha\circ\mathcal{C}_{\mathrm{work}}\llbracket\mathsf{P}\rrbracket$ *are restricted to sets of* polynomial-time constructible *distributions.*

The proposition is proved by the induction over the structure of P. For $\mathsf{P}=\mathit{skip}$ the proposition obviously holds, as $\mathcal{W}_{\mathbf{Var}}[\![skip]\!]$ and $\mathcal{C}_{\mathrm{work}}[\![skip]\!]$ are both identity functions. The proofs for other constructs follow.

7.1 Assignments

Let the program P be $x := o(x_1,\ldots,x_k)$. We must show the correctness of each rule in Fig. 2. We are going to use the following line of thought: Let R be an instance of a rule in Fig. 2 (hence defining $\mathcal{X}, X, Y$ and possibly other quantities). Let $X', Y' \subseteq \widetilde{\mathbf{Var}}$ be such that $(X', Y') \in \mathcal{W}_{\mathbf{Var}}[\![x := o(x_1,\ldots,x_k)]\!](\mathcal{X})$ is the consequent of R.

LEMMA 3. *Suppose that for each polynomial-time constructible $D \in \prod_{n\in\mathbb{N}}(\mathcal{D}(\mathbf{State}_{n\perp}))$, where $\beta(D) \supseteq \mathcal{X}$ (the representation function β was defined in Sec. 5), there exists a PPT algorithm $\mathcal{A}$, such that*

- *when the input of $\mathcal{A}(\mathbf{1}^n,\cdot)$ is distributed according to*
$$\{\!| S_n \leftarrow D_n, S'_n \leftarrow D_n \;:\; \langle\!\langle S_n \neq \perp \wedge S'_n \neq \perp \;?\; \{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S_n)\}_{\overline{v}\in Y} : \perp\rangle\!\rangle |\!\} \quad (14)$$
then its output is distributed according to
$$\{\!| S_n \leftarrow D'_n, S'_n \leftarrow D'_n \;:\; \langle\!\langle S_n \neq \perp \wedge S'_n \neq \perp \;?\; \{\overline{v}(S_n)\}_{\overline{v}\in X'}, \{\overline{v}(S_n)\}_{\overline{v}\in Y'} : \perp\rangle\!\rangle |\!\} \quad (15)$$
where $D' = \big((ext^{\mathcal{D}} \circ \widehat{ext}^{\perp})(\mathcal{C}_{\mathrm{len}}^{(\lambda n.1)}[\![\mathsf{P}]\!])\big)(D)$. I.e. the concrete semantics of P transforms the distribution D to the distribution D'.
- *when the input of $\mathcal{A}(\mathbf{1}^n,\cdot)$ is distributed according to*
$$\{\!| S_n \leftarrow D_n, S'_n \leftarrow D_n \;:\; \langle\!\langle S_n \neq \perp \wedge S'_n \neq \perp \;?\; \{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y} : \perp\rangle\!\rangle |\!\} \quad (16)$$
then its output is distributed according to
$$\{\!| S_n \leftarrow D'_n, S'_n \leftarrow D'_n \;:\; \langle\!\langle S_n \neq \perp \wedge S'_n \neq \perp \;?\; \{\overline{v}(S_n)\}_{\overline{v}\in X'}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y'} : \perp\rangle\!\rangle |\!\} \;. \quad (17)$$

In this case $(X', Y') \in (\alpha \circ \mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!] \circ \gamma)(\mathcal{X})$.

Thus, if the lemma is applicable to all instances of all rules R, then we have $\mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}) \subseteq (\alpha \circ \mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!] \circ \gamma)(\mathcal{X})$, which is equivalent to the statement of the proposition.

PROOF. Suppose $(X', Y') \notin (\alpha \circ \mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!] \circ \gamma)(\mathcal{X})$. Then there exists a distribution $D \in \gamma(\mathcal{X})$ (equivalently, $\beta(D) \supseteq \mathcal{X}$) and a distribution $D' \in \mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!](\mathsf{cl}_{\mathrm{L}}\{D\})$, such that $(X', Y') \notin \beta(D')$. As
$$\mathcal{C}_{\mathrm{work}}[\![\mathsf{P}]\!](\mathsf{cl}_{\mathrm{L}}\{D\}) = \mathsf{cl}_{\mathrm{L}}\big\{\big((ext^{\mathcal{D}} \circ \widehat{ext}^{\perp})(\mathcal{C}_{\mathrm{len}}^{(\lambda n.1)}[\![\mathsf{P}]\!])\big)(D)\big\}$$
and β is monotone, we may assume that D' is specified in terms of D in the same way as in the statement of the lemma. The fact $(X', Y') \in \beta(D')$ means that there exists a PPT algorithm $\mathcal{B}$ distinguishing the distributions (15) and (17). The composition of the algorithms $\mathcal{B}$ and $\mathcal{A}$ distinguishes the distributions (14) and (16); this means that $(X, Y) \notin \beta(D)$. But then also $(X, Y) \notin \mathcal{X}$ which is a contradiction with the definitions of $X, Y, \mathcal{X}$. ☐

We now show that the required algorithm $\mathcal{A}$ indeed exists for all the rules R. Let the input of $\mathcal{A}$ be the tuple $(\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y})$, where S_n and S'_n may be either the same state or independent states generated according to D_n.

Rule (9). $\mathcal{A}$ implements the identity function.

Rule (10). The algorithm $\mathcal{A}$ first generates a state S''_n according to the distribution D_n. It can now construct the tuple
$$(\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y\backslash Z}, \{\overline{v}(S''_n)\}_{\overline{v}\in Z\cup O})$$
which, according to the last antecedent of the rule (10), is computationally indistinguishable from the tuple
$$(\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y\backslash Z}, \{\overline{v}(S'_n)\}_{\overline{v}\in Z\cup O}) = (\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y\cup O}) \;.$$
This is trivially true if S_n and S'_n are the same state. If S_n and S'_n are independent states then this statement follows from the lower-closedness of $\mathcal{X}$ — we also have $(Z \cup O, Y\backslash Z) \in \mathcal{X}$.

The algorithm $\mathcal{A}$ now knows the values of all the variables $x_1,\ldots,x_k$ in the state S'_n. It can thus find the value of the variable x after the program $x := o(x_1,\ldots,x_k)$ has been executed in the state S'_n.

The variants of the rule (10) just reflect the meanings of different elements of $\widetilde{\mathbf{Var}}$. The algorithm $\mathcal{A}$ is almost the same for them.

Rule (11). The algorithm $\mathcal{A}$ generates a new key $\underline{k} \leftarrow [\![\mathcal{G}en]\!]_n$ and outputs
$$(\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y}, [\![\mathcal{E}nc]\!]_n(\underline{k}, \mathbf{0}^n)) \;. \quad (18)$$
This is indeed computationally indistinguishable from (15) (if S_n and S'_n are the same) or (17) (if S_n and S'_n are independent). Otherwise one of the algorithms $\dot{\mathcal{A}}^{(\cdot)}$ and $\ddot{\mathcal{A}}^{(\cdot)}$ described below show, that $[\![\mathcal{E}nc]\!]$ is not repetition-concealing.

Let $\mathcal{B}$ be the algorithm distinguishing (18) from either (15) or (17). $\dot{\mathcal{A}}^{(\cdot)}(\mathbf{1}^n)$ works as follows: it first generates S_n and S'_n, which are either the same or independent, depending on the case for which $\mathcal{A}$ fails. Let $\underline{k}$ be the key that the oracle of $\dot{\mathcal{A}}^{(\cdot)}$ uses. $\dot{\mathcal{A}}$ submits the value of the variable y in the state S'_n to the oracle, obtaining either $[\![\mathcal{E}nc]\!]_n(\underline{k}, S'_n(y))$ or $[\![\mathcal{E}nc]\!]_n(\underline{k}, \mathbf{0}^{\ell(n)})$. It can now create either the tuple
$$(\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y\backslash\{\overline{\mathcal{E}nc(k,\cdot)}\}}, [\![\mathcal{E}nc]\!]_n(\underline{k},\cdot), [\![\mathcal{E}nc]\!]_n(\underline{k}, S'_n(y))) \quad (19)$$
or the tuple
$$(\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y\backslash\{\overline{\mathcal{E}nc(k,\cdot)}\}}, [\![\mathcal{E}nc]\!]_n(\underline{k}, \mathbf{0}^{|\cdot|}), [\![\mathcal{E}nc]\!]_n(\underline{k}, \mathbf{0}^{\ell(n)})) \;. \quad (20)$$
According to the last consequent of the rule (11), the tuple (19) is computationally indistinguishable from either (15) or (17). The tuple (20) may or may not be computationally

indistinguishable from the tuple

$$(\{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y\setminus\{\overline{\mathcal{E}nc(k,\cdot)}\}},$$
$$[\![\mathcal{E}nc]\!]_n(\underline{k},\cdot), [\![\mathcal{E}nc]\!]_n(\underline{k},\mathbf{0}^{\ell(n)})), \quad (21)$$

which, according to the last consequent of the rule (11), is indistinguishable from (18). Suppose that (20) and (21) are indistinguishable. The algorithm $\dot{\mathcal{A}}$ now calls the algorithm $\mathcal{B}$ on the tuple (19) or (20). $\mathcal{B}$ returns, whether its input was (19) or (20), thus telling, whether the oracle of $\dot{\mathcal{A}}^{(\cdot)}$ was $[\![\mathcal{E}nc]\!]_n(\underline{k},\cdot)$ or $[\![\mathcal{E}nc]\!]_n(\underline{k},\mathbf{0}^{|\cdot|})$.

If (20) and (21) are distinguishable, then let $\dot{\mathcal{B}}$ be the algorithm distinguishing them. $\ddot{\mathcal{A}}^{(\cdot)}(\mathbf{1}^n)$ works as follows: it first generates S_n and S'_n, which are either the same or independent. Let $\underline{k}$ be the key used by the oracle. $\ddot{\mathcal{A}}$ generates $[\![\mathcal{E}nc]\!]_n(\underline{k},\mathbf{0}^{\ell(n)})$ by submitting $\mathbf{0}^{\ell(n)}$ to the oracle. Now it can generate either the tuple (20), if its oracle is $[\![\mathcal{E}nc]\!]_n(\underline{k},\mathbf{0}^{|\cdot|})$, or (21), if its oracle is $[\![\mathcal{E}nc]\!]_n(\underline{k},\cdot)$. The algorithm $\dot{\mathcal{B}}$ is used to distinguish those two cases.

Rule (12). The algorithm $\mathcal{A}$ should generate either one (to construct (15) from (14)) or two (to construct (17) from (16)) new encryption keys. But it is impossible to distinguish how many new keys were created, this follows directly from the which-key concealedness of the encryption scheme. To complete the specification of $\mathcal{A}$, just fix one of those two cases.

Rule (13). Similarly to the previous rule, $\mathcal{A}$ has to generate a new value for the variable x either once or twice. But if the new value is generated twice, the same value is created at both times.

7.2 Sequential Composition

The statement of the proposition follows immediately from $\mathcal{C}_{\mathsf{work}}[\![\mathsf{P}_1;\mathsf{P}_2]\!] = \mathcal{C}_{\mathsf{work}}[\![\mathsf{P}_2]\!]\circ\mathcal{C}_{\mathsf{work}}[\![\mathsf{P}_1]\!]$. The truth of the last formula follows easily from the definitions of $\mathcal{C}_{\mathsf{work}}[\![\cdot]\!]$, $\mathcal{C}_{\mathrm{Path}}[\![\cdot]\!]$ and from the fact that for a polynomial $q\in\mathbb{Z}[n]$ we have

$$\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}_1;\mathsf{P}_2]\!] \le ext^{\mathcal{D}}(\widehat{ext}^{\perp}(\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}_2]\!]))\circ\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}_1]\!] \le \mathcal{C}^{(2q)}_{\mathrm{len}}[\![\mathsf{P}_1;\mathsf{P}_2]\!],$$

giving us that the lower closure of the set (see (5))

$$\{q\in\mathbb{Z}[n] : ((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}_1;\mathsf{P}_2]\!]))(D)\}$$

is equal to the lower closure of the set

$$\{q_1\in\mathbb{Z}[n], q_2\in\mathbb{Z}[n] : \Big(((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q_2)}_{\mathrm{len}}[\![\mathsf{P}_2]\!]))\circ ((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q_1)}_{\mathrm{len}}[\![\mathsf{P}_1]\!]))\Big)(D)\} \ .$$

7.3 Branches

Let P be the program *if* b *then* P_1 *else* P_2 and let $D\in\prod_{n\in\mathbb{N}}\mathcal{D}(\mathbf{State}_{n\perp})$. We will show that $\mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\beta(D))\ge\alpha(\mathcal{C}_{\mathsf{work}}[\![\mathsf{P}]\!](\mathsf{cl}_{\mathrm{L}}\{D\}))$ (remember that here "$\ge$" means "$\subseteq$"). Assume the contrary, i.e. there exist $X\in\overline{\mathbf{Var}}_{\mathrm{asgn}}$, $X',Y'\in(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}}_{\mathrm{asgn}})$, such that $(X\cup X',Y')\in\mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\beta(D))$, but $(X\cup X',Y')\notin\alpha(\mathcal{C}_{\mathsf{work}}[\![\mathsf{P}]\!](\mathsf{cl}_{\mathrm{L}}\{D\}))$. The last non-inclusion means, that there exists such $q\in\mathbb{Z}[n]$ that $X\cup X'$ and Y' are not independent in the distribution $((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!]))(D)$. Let $\mathcal{B}$ be the algorithm showing the dependence of $X\cup X'$ and Y' in that distribution.

Let P'_1 be the program *if* b *then* P_1 *else stuck* and P'_2 be the program *if* b *then stuck else* P_2, where *stuck* is a program that never terminates. For at least one $i\in\{1,2\}$, the sets $X\cup X'$ and Y' are not independent in the distribution $((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{Path}}[\![\mathsf{P}'_i]\!]))(D)$ because the distribution $((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}]\!]))(D)$ is equal to the pointwise "sum" of the distributions $((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{Path}}[\![\mathsf{P}'_i]\!]))(D)$ where $i\in\{1,2\}$. Indeed, the same algorithm $\mathcal{B}$ can be used to show their dependence in at least one of those distributions. Assume w.l.o.g. that $i=1$.

The sets $\{\overline{N}\}\cup X\cup X'$ and Y' must be dependent in the distribution $((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{len}}[\![N:=b;\mathsf{P}_1]\!]))(D)$. Their dependence is shown by the following algorithm: first check the value of N. If it is false, then output "don't know". Otherwise (and this happens non-negligibly often) call the algorithm $\mathcal{B}$ on the values of the variables in $X\cup X'$ and Y' and return whatever it returns. This algorithm has the same success probability as the algorithm $\mathcal{B}$ on $((ext^{\mathcal{D}}\circ\widehat{ext}^{\perp})(\mathcal{C}^{(q)}_{\mathrm{len}}[\![\mathsf{P}'_1]\!]))(D)$.

We thus have

$$(\{\overline{N}\}\cup X\cup X',Y')\notin\alpha(\mathcal{C}_{\mathsf{work}}[\![N:=b;\mathsf{P}_1]\!](\mathsf{cl}_{\mathrm{L}}\{D\}))$$

and from the induction hypothesis (program "$N:=b;\mathsf{P}_1$" has less branches and the same number of nested loops as P) follows

$$(\{\overline{N}\}\cup X\cup X',Y')\notin\mathcal{W}_{\mathbf{Var}\uplus\{N\}}[\![N:=b;\mathsf{P}_1]\!](\beta(D)) \ .$$

But this contradicts the definition of $\mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!]$.

7.4 Loops

Let P be the program *while* b *do* P'. Let Q_0 be a program that never returns and let the programs $\{\mathsf{Q}_n\}_{n\in\mathbb{N}}$ be defined by

$$\mathsf{Q}_i := \textit{if } b \textit{ then } \mathsf{P}';\mathsf{Q}_{i-1} \textit{ else skip} \ .$$

The concrete semantics of P (see Fig. 1) is defined as the least upper bound of concrete semantices of the programs Q_i. Applying the induction hypothesis to the programs Q_i (they have less nested loops than P) gives

$$\Big(\bigvee_{i\in\mathbb{N}}\mathcal{W}_{\mathbf{Var}}[\![\mathsf{Q}_i]\!]\Big)\circ\alpha\ge\alpha\circ\mathcal{C}_{\mathsf{work}}[\![\mathsf{P}]\!]$$

(the abstract semantics of a program that never returns is the function that maps each element to the bottom element of $\mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}})\,\overline{\times}\,\mathcal{P}(\widetilde{\mathbf{Var}})))$, i.e. to the entire set $\mathcal{P}(\widetilde{\mathbf{Var}})\,\overline{\times}\,\mathcal{P}(\widetilde{\mathbf{Var}})$).

To complete the proof, we have to show that for each $i\in\mathbb{N}$ and $\mathcal{X}\in\mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}})\,\overline{\times}\,\mathcal{P}(\widetilde{\mathbf{Var}})))$,

$$\mathcal{W}_{\mathbf{Var}}[\![\mathsf{Q}_i]\!](\mathcal{X})\supseteq \{(X,Y) : X\subseteq\widetilde{\mathbf{Var}}, Y\subseteq(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}}_{\mathrm{asgn}}), (\{\overline{N}\}\cup X,Y)\in\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}\uplus\{N\}}(\mathcal{W}_{\mathbf{Var}\uplus\{N\}}[\![N:=\mathrm{const}]\!](\mathcal{X}))\}\cup (\mathcal{X}\cap(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}}_{\mathrm{asgn}})\,\overline{\times}\,(\widetilde{\mathbf{Var}}\backslash\overline{\mathbf{Var}}_{\mathrm{asgn}}))$$

holds. This is proved by induction over i. The proof is simple but lengthy, and is omitted.

The idea of the proof is the following: both $\mathcal{W}_{\mathbf{Var}}[\![\mathsf{Q}_i]\!]$ and $\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}\uplus\{N\}}$ characterise the information flow of the program *while b do* P', where the loop is iterated at most i times. They differ in ways how the information contained in the control flow is kept track of. $\mathcal{W}_{\mathbf{Var}}[\![\mathsf{Q}_i]\!]$ introduces i new variables, where each new variable captures the decision made in checking the value of b before one single iteration. In the same time, $\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}\uplus\{N\}}$ uses a single variable N that captures the decisions made in checking the value of b before all iterations. So, intuitively $\mathcal{W}_{\mathbf{Var}}[\![\mathsf{Q}_i]\!]$ appears to be more complex (and thus more precise) than $\mathcal{S}^{\mathsf{P},N}_{i,\mathbf{Var}\uplus\{N\}}$.

7.5 Conclusion

THEOREM 4. *Let* P *be a program satisfying the conditions at the beginning of Sec. 6. Let* $\mathbf{Var}$ *be the set of variables of* P. *Let* $X, Y \subseteq \mathbf{Var}$ *and* $D_s \in \prod_{\mathcal{D}(\mathbf{State}_n)}$, *where* D_s *is polynomial-time constructible. Assume that the program* P *does not assign to the variables in* X. *If*

$$(\overline{X}, \overline{Y}) \in \mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\beta(D_s))$$

then the initial values of the variables in X *are irrecoverable from the final values of the variables in* Y, *given that the initial state of the program is distributed according to* D_s.

This result follows directly from propositions 1 and 2.

8. COMPARISON WITH [7]

We are going to show that the analysis we just presented is at least as precise than the one we presented in [7]. There is an intuitive comparison of the abstract domains of the analyses in the current paper and in [7]: the analysis in the current paper records such pairs of sets of variables that are independent of each other, whereas the analysis in [7] only records such pairs, whose one component is the set of all secret variables $\mathbf{Var}_{\mathrm{S}}$.

More precisely, the analysis $\mathcal{V}_{\mathbf{Var}}[\![\mathsf{P}]\!]$ in [7] records all those sets of variables X, that *may be dependent* of the set of variables $\mathbf{Var}_{\mathrm{S}}$. It is obvious that the set of all such sets X is upper closed. Thus the type of the analysis is[5] $\mathcal{P}_{\mathrm{U}}(\mathcal{P}(\mathbf{Var})) \to \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\mathbf{Var}))$.

The program P had to satisfy additional constraints for the analysis $\mathcal{V}_{\mathbf{Var}}[\![\mathsf{P}]\!]$ to be correct. One of them required that the only way the program could use the keys held in its variables was to use them for encrypting other values. This made the variables holding keys pretty independent of anything else (and also of each other). Another constraint required that at each program point, there existed a (statically known) equivalence relation E over variables holding keys, such that two variables are equal iff they are related by E.

Fig. 3 defines $\mathcal{V}_{\mathbf{Var}}[\![\mathsf{P}]\!]$ for assignments. The rule (23) is to be used for encryptions and the rule (22) for everything else. $\mathcal{V}_{\mathbf{Var}}[\![\cdot]\!]$ for *skip* and sequences is again identity function and

$$\frac{\begin{array}{c} X \in \mathcal{X} \\ x \notin X \\ o \neq \mathcal{E}nc \\ Z := X \backslash \{x_1, \ldots, x_k\} \cup \{x\} \end{array}}{X, Z \in \mathcal{V}_{\mathbf{Var}}[\![x := o(x_1, \ldots, x_k)]\!](\mathcal{X})} \qquad (22)$$

$$\frac{\begin{array}{c} X \in \mathcal{X} \\ x \notin X \\ k' \mathrel{\mathsf{E}} k \\ Z := X \backslash \{y\} \cup \{x, k'\} \end{array}}{X, Z \in \mathcal{V}_{\mathbf{Var}}[\![x := \mathcal{E}nc(k, y)]\!](\mathcal{X})} \qquad (23)$$

Figure 3: The semantics $\mathcal{V}_{\mathbf{Var}}[\![\cdot]\!]$ for assignments

composition, respectively. $\mathcal{V}_{\mathbf{Var}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2]\!](\mathcal{X})$, where $\mathcal{X} \subseteq_{\mathrm{U}} \mathcal{P}(\mathbf{Var})$, is equal to the set

$$\begin{array}{l}\{X \ : \ i \in \{1,2\}, X \in \mathcal{V}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_i]\!](\mathcal{X}), N \notin X\} \cup \\ \{X \backslash \{N\} \cup \{v\} \ : \\ \quad i \in \{1,2\}, X \in \mathcal{V}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_i]\!], v \in \mathbf{Var}_{\mathrm{asgn}}\},\end{array}$$

where $\mathbf{Var}_{\mathrm{asgn}}$ is the set of variables that are assigned to in at least one of the branches. $\mathcal{V}_{\mathbf{Var}}[\![\mathit{while}\ b\ \mathit{do}\ \mathsf{P}]\!]$ is defined similarly to $\mathcal{W}_{\mathbf{Var}}[\![\cdot]\!]$ (by defining auxiliary functions characterising the execution of the loop for at most a fixed number of iterations).

Let P be a program whose set of secret variables is $\mathbf{Var}_{\mathrm{S}}$. Let $\mathbf{Var}_{\mathrm{key}} \subseteq \mathbf{Var}$ be the set of variables whose initial values are used as keys in P. Let the set $\widehat{\mathbf{Var}_{\mathrm{S}}} \subseteq \widetilde{\mathbf{Var}}$ consist of the following elements:

- $\overline{\mathcal{E}nc(k, \cdot)}$ for all $k \in \mathbf{Var}_{\mathrm{S}} \cap \mathbf{Var}_{\mathrm{key}}$;
- $\overline{x}$ for all $x \in \mathbf{Var}_{\mathrm{S}} \backslash \mathbf{Var}_{\mathrm{key}}$.

PROPOSITION 5. *Let the program* P *and the initial probability distribution* D_s *be such that the aforementioned constraints for the correctness of* $\mathcal{V}_{\mathbf{Var}}[\![\cdot]\!]$ *are satisfied. Assume that the program does not assign to its secret variables. Let* $\mathcal{X}_{\mathcal{V}} \in \mathcal{P}_{\mathrm{U}}(\mathcal{P}(\mathbf{Var}))$ *and* $\mathcal{X}_{\mathcal{W}} \in \mathcal{R}(\mathcal{P}_{\mathrm{L}}(\mathcal{P}(\widetilde{\mathbf{Var}}) \,\overline{\times}\, \mathcal{P}(\widetilde{\mathbf{Var}})))$ *be the abstractions of* D_s *for the analyses* $\mathcal{V}_{\mathbf{Var}}[\![\cdot]\!]$ *and* $\mathcal{W}_{\mathbf{Var}}[\![\cdot]\!]$, *respectively, for which the following claims hold:*

1. *For all* $X \subseteq \mathbf{Var}$: *if* $X \notin \mathcal{X}_{\mathcal{V}}$ *then* $(\widehat{\mathbf{Var}_{\mathrm{S}}}, \overline{X}) \in \mathcal{X}_{\mathcal{W}}$.
2. *If* $\underline{k} \in \mathbf{Var}$ *and* $X \subseteq \widetilde{\mathbf{Var}}$, *such that for no* $\overline{k'} \in X$ *holds* $\underline{k} \mathrel{\mathsf{E}} k'$, *then* $(\{\overline{\mathcal{E}nc(\underline{k}, \cdot)}\}, X) \in \mathcal{X}_{\mathcal{W}}$.
3. *If* $X \subseteq \mathbf{Var}$ *and* $k \in X$, *where the variable* k *is holding a key, and if* $X \in \mathcal{X}_{\mathcal{V}}$, *then also* $X \backslash (k/\,\mathsf{E}) \cup \{k\} \in \mathcal{X}_{\mathcal{V}}$, *where* $(k/\,\mathsf{E})$ *denotes the equivalence class of* k. *This condition says that if we can find something about the values of secret variables using the values of the variables in* X, *then we only need to use one of the several variables holding the same value.*

[5] In [7], the analysis was not presented in syntax-directed way, but as a data flow analysis on the flowchart of the program. This is more suitable for an implementation but, in my opinion, less suitable for presentation.

(note that all these properties hold if $\mathcal{X}_\mathcal{V}$ and $\mathcal{X}_\mathcal{W}$ are both the best approximations of D_s). Then for all $X \subseteq \mathbf{Var}$ holds

$$X \notin \mathcal{V}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}_\mathcal{V}) \Longrightarrow (\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X}) \in \mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}_\mathcal{W}) \ .$$

PROOF. The proof goes by induction over the program structure. We show that not only the claim 1 holds at the end of the program P (that's what the proposition states), but the claims 2 and 3 as well. Let $X \subseteq \mathbf{Var}$ be such that $X \notin \mathcal{V}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}_\mathcal{V})$.

Assignments — encryption. Assume P is the program $x := \mathcal{E}nc(k, y)$. If $x \notin X$ then the rule (23) gives $X \notin \mathcal{X}_\mathcal{V}$ and thus also $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X}) \in \mathcal{X}_\mathcal{W}$. The rule (9) now gives $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X}) \in \mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}_\mathcal{W})$. If $x \in X$ then there are two possibilities: either there exists a $k' \in X$ such that $k \,\mathsf{E}\, k'$ or there exists no such k'. In the first case the claim 3 gives that we may assume w.l.o.g. that $k \in X$ (the claim 3 says that assuming that the entire equivalence class $k/\,\mathsf{E}$ belongs to X is w.l.o.g.). From the rule (23) follows $X\backslash\{x\} \cup \{y\} \notin \mathcal{X}_\mathcal{V}$, thus $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X}\backslash\{\overline{x}\} \cup \{\overline{y}\}) \in \mathcal{X}_\mathcal{W}$ and the rule (10) gives $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X} \cup \{\overline{y}\}) \in \mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}_\mathcal{W})$ (in the rule instantiate $Z := \emptyset$). In the second case (there is no $k' \in X$ such that $k \,\mathsf{E}\, k'$) we only have $X\backslash\{x\} \notin \mathcal{X}_\mathcal{V}$, but the claim 2 and the fact $(k/\,\mathsf{E}) \cap X = \emptyset$ give us that the last antecedent of the rule (11) holds, if the instantiations $X := \overline{X}\backslash\{\overline{x}\}$ and $Y := \widehat{\mathbf{Var}}_\mathrm{S}$ are made in this rule. Thus $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X}) \in \mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}_\mathcal{W})$.

Similar argument shows that the claim 2 holds at the end of the program P. Just replace the set $\widehat{\mathbf{Var}}_\mathrm{S}$ by the set $\{\overline{\mathcal{E}nc(\underline{k}, \cdot)}\}$ in the previous paragraph.

Also, the rule (23) shows that the claim 3 holds at the end of the program P. If there exists $k'' \in X$ such that $k \,\mathsf{E}\, k''$, then instantiate $k' := k''$.

Assignments — key generation. Assume P is the program $k := \mathcal{G}en()$. The rule (22) gives us $X\backslash\{k\} \notin \mathcal{X}_\mathcal{V}$ and either the rule (9) (if $k \notin X$) or the rule (10) (if $k \in X$) gives $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X}) \in \mathcal{W}_{\mathbf{Var}}[\![\mathsf{P}]\!](\mathcal{X}_\mathcal{W})$.

The claim 2 follows from applying the rule (12) to the fact $(\emptyset, X\backslash\{\overline{\mathcal{E}nc(k, \cdot)}\}) \in \mathcal{X}_\mathcal{W}$. The claim 3 is obvious.

Assignments — other. Assume P is the program $x := o(x_1, \ldots, x_k)$. The case $x \notin X$ is identical to the same case for encryptions. If $x \in X$ then the rule (22) gives $X\backslash\{x\} \cup \{x_1, \ldots, x_k\} \notin \mathcal{X}_\mathcal{V}$ and an application of the rule (10) (where Z is instantiated to $\emptyset$) completes the proof. The claims 2 and 3 obviously hold.

Sequences. Obvious. P is a program $\mathsf{P}_1; \mathsf{P}_2$, the induction hypothesis says that if all claims hold at the beginning of P_1, then they also hold at the end of P_1, i.e. at the beginning of P_2 and thus also at the end of P_2.

Branches. Let P be the program *if b then* P_1 *else* P_2 and let $\mathbf{Var}_\mathrm{asgn} \subseteq \mathbf{Var}$ be the set of variables that are assigned to in either P_1 or P_2. If $X \cap \mathbf{Var}_\mathrm{asgn} = \emptyset$ then the proof is similar to the case of assignments, when the left hand side of the assignment was not a member of X. If $X \cap \mathbf{Var}_\mathrm{asgn} \neq \emptyset$ then $X \cup \{N\}$ is a member of neither $\mathcal{V}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_1]\!](\mathcal{X}_\mathcal{V})$ nor $\mathcal{V}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_2]\!](\mathcal{X}_\mathcal{V})$. According to the induction hypothesis, the following claims hold:

1'. $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X} \cup \{\overline{N}\}) \in \mathcal{W}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_1]\!](\mathcal{X}_\mathcal{W})$ and $(\widehat{\mathbf{Var}}_\mathrm{S}, \overline{X} \cup \{\overline{N}\}) \in \mathcal{W}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_2]\!](\mathcal{X}_\mathcal{W})$.

2'. If $\underline{k} \in \mathbf{Var}$ and for no $k' \in X$ holds $\underline{k} \,\mathsf{E}\, k'$, then $(\{\overline{\mathcal{E}nc(\underline{k}, \cdot)}\}, \overline{X} \cup \{\overline{N}\}) \in \mathcal{W}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_1]\!](\mathcal{X}_\mathcal{W})$ and $(\{\overline{\mathcal{E}nc(\underline{k}, \cdot)}\}, \overline{X} \cup \{\overline{N}\}) \in \mathcal{W}_{\mathbf{Var}\uplus\{N\}}[\![N := b; \mathsf{P}_2]\!](\mathcal{X}_\mathcal{W})$.

For the program P, the claims 1 and 2 now follow immediately from the definition of $\mathcal{W}_{\mathbf{Var}}[\![$*if b then* P_1 *else* $\mathsf{P}_2]\!]$. From the definition of$\mathcal{V}_{\mathbf{Var}}[\![$*if b then* P_1 *else* $\mathsf{P}_2]\!]$ we get the claim 3.

Loops. Similar to branches. □

9. CONCLUSION

To conclude, the goals in designing the analysis $\mathcal{W}$ were the same as in designing the analysis $\mathcal{V}$ (see also [7, Sec. 7])—to analyse programs containing which-key and repetition concealing encryption as a primitive operation—but this time, these goals have been fully achieved. By modifying the set $\widetilde{\mathbf{Var}}$, it may be possible to extend the analysis $\mathcal{W}$ to handle primitives with different, weaker properties.

10. REFERENCES

[1] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.

[2] M. Abadi and P. Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 3–22, Sendai, Japan, Aug. 2000. Springer-Verlag.

[3] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, Feb. 1990.

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

[5] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[6] J. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, Apr. 1982. IEEE Computer Society Press.

[7] P. Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In Sands [16], pages 77–91.

[8] K. Leino and R. Joshi. A Semantic Approach to Secure Information Flow. In J. Jeuring, editor, *Mathematics of Program Construction, MPC '98*, volume 1422 of *LNCS*, pages 254–271, Marstrand, Sweden, June 1998. Springer-Verlag.

[9] P. Lincoln, J. Mitchell, M. Mitchell, and A. 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, Nov. 1998. ACM Press.

[10] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic Polynomial-Time Equivalence and Security Analysis. In J. Wing, J. Woodcock, and J. 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, Sept. 1999. Springer-Verlag.

[11] J. Mitchell. Probabilistic Polynomial-Time Process Calculus and Security Protocol Analysis. In Sands [16], pages 23–29.

[12] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[13] B. Pfitzmann, M. Schunter, and M. Waidner. Cryptographic Security of Reactive Systems. In S. Schneider and P. Ryan, editors, *Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*, Royal Holloway, University of London, 2000. Elsevier Science.

[14] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 245–254, Athens, Greece, Nov. 2000. ACM Press.

[15] B. Pfitzmann and M. 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.

[16] D. Sands, editor. *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *LNCS*, Genova, Italy, Apr. 2001. Springer-Verlag.

[17] D. 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.

[18] D. Volpano and G. 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, Jan. 2000. ACM Press.

[19] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

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

APPENDIX

A. PROOFS OF DETAILS

LEMMA 6. *An encryption scheme $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is which-key and repetition concealing iff it is which-key concealing and repetition concealing.*

PROOF. **Direction ($\Rightarrow$).** Suppose that $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not repetition concealing, i.e. there exists a PPT algorithm $\mathcal{A}^{(\cdot)}$ such that (1) is not negligible. Modify the algorithm $\mathcal{A}^{(\cdot)}$ so, that it takes two oracles instead of one, but queries only the left one. For such algorithm, the difference (3) is not negligible, i.e. $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not which-key and repetition concealing.

On the other hand, suppose that $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not which-key concealing, but is which-key and repetition concealing. This means that there exists an algorithm $\mathcal{A}^{(\cdot),(\cdot)}$ for which (2) is not negligible, but (3) is negligible. The difference of (2) and (3):

$$\begin{aligned}&\Pr\big[k \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n) = 1\big] - \\ &\quad \Pr\big[k \leftarrow \mathcal{G}(\mathbf{1}^n) : \mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|}),\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})}(\mathbf{1}^n) = 1\big] \qquad (24)\end{aligned}$$

is not negligible as well. Let $\bar{\mathcal{A}}^{(\cdot)}$ be an algorithm that runs the algorithm $\mathcal{A}^{(\cdot),(\cdot)}$, and whenever it queries an oracle (no matter whether the left or right one), $\bar{\mathcal{A}}^{(\cdot)}$ passes the query to its own single oracle. The difference (1) is not negligible for the algorithm $\bar{\mathcal{A}}^{(\cdot)}$, thus $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not repetition concealing. But we just showed that a which-key and repetition concealing encryption scheme must be repetition concealing. A contradiction.

Direction ($\Leftarrow$). Suppose that $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not which-key and repetition concealing, but is which-key concealing. This means that there exists an algorithm $\mathcal{A}^{(\cdot),(\cdot)}$ for which (2) is negligible, but (3) is not negligible. We again get that their difference (24) is not negligible and thus $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not repetition concealing. $\square$

LEMMA 7. *Let $D, D' \in \prod_{n\in\mathbb{N}} \mathcal{D}(\mathbf{State}_{n\perp})$, such that $D \leq D'$, and let $X, Y \subseteq \widetilde{\mathbf{Var}}$. If X and Y are computationally independent in D', then they are also computationally independent in D.*

PROOF. For each PPT algorithm $\mathcal{A}$, $n \in \mathbb{N}$ and $S_n, S'_n \in \mathbf{State}_n$ define

$$\begin{aligned}&\mathbf{P}_n(\mathcal{A}, S_n, S'_n) := \\ &\quad \Pr[\mathcal{A}(\mathbf{1}^n, \{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S_n)\}_{\overline{v}\in Y}) = 1] - \\ &\qquad \Pr[\mathcal{A}(\mathbf{1}^n, \{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S'_n)\}_{\overline{v}\in Y}) = 1],\end{aligned}$$

where the probabilities are taken over the random choices of $\mathcal{A}$. The computational independence of X and Y in D' means that the difference of probabilities

$$\begin{aligned}&\Pr\bigl[S_n, S_n' \leftarrow D_n' \ : \ S_n \neq \bot \wedge S_n' \neq \bot \wedge \\ &\qquad \mathcal{A}(\mathbf{1}^n, \{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S_n)\}_{\overline{v}\in Y}) = 1\bigr] - \\ &\Pr\bigl[S_n, S_n' \leftarrow D_n' \ : \ S_n \neq \bot \wedge S_n' \neq \bot \wedge \\ &\qquad\quad \mathcal{A}(\mathbf{1}^n, \{\overline{v}(S_n)\}_{\overline{v}\in X}, \{\overline{v}(S_n')\}_{\overline{v}\in Y}) = 1\bigr] \qquad (25)\end{aligned}$$

is negligible in n for all PPT algorithms $\mathcal{A}$. One can easily see that (25) is equal to

$$\sum_{S_n, S_n' \in \mathbf{State}_n} D_n'(S_n) \cdot D_n'(S_n') \cdot \mathbf{P}_n(\mathcal{A}, S_n, S_n') \ . \qquad (26)$$

Similarly, X and Y are computationally independent in D iff for all PPT algorithms $\mathcal{A}$ the sum

$$\sum_{S_n, S_n' \in \mathbf{State}_n} D_n(S_n) \cdot D_n(S_n') \cdot \mathbf{P}_n(\mathcal{A}, S_n, S_n') \qquad (27)$$

is negligible. But (27) is less or equal to (26), because for all $S_n \in \mathbf{State}_n$ holds $D_n(S_n) \leq D_n'(S_n)$. $\square$