Information-Flow Types for Homomorphic Encryptions

Cédric Fournet
Microsoft Research
fournet@microsoft.com

Jérémy Planul
MSR–INRIA Joint Centre
jeremy.planul@inria.fr

Tamara Rezk
INRIA Sophia Antipolis-Méditerranée
Tamara.Rezk@inria.fr

ABSTRACT

We develop a flexible information-flow type system for a range of encryption primitives, precisely reflecting their diverse functional and security features. Our rules enable encryption, blinding, homomorphic computation, and decryption, with selective key re-use for different types of payloads.

We show that, under standard cryptographic assumptions, any well-typed probabilistic program using encryptions is secure (that is, computationally non-interferent) against active adversaries, both for confidentiality and integrity. We illustrate our approach using ElGamal and Paillier encryption.

We present two applications of cryptographic verification by typing: (1) private search on data streams; and (2) the bootstrapping part of Gentry's fully homomorphic encryption. We provide a prototype typechecker for our system.

Categories and Subject Descriptors

K.6.m [**Security and Protection**]: Security; D.2.0 [**Software Engineering**]: Protection Mechanisms; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification techniques.

General Terms

Security, Verification, Design, Languages.

Keywords

Secure information flow, cryptography, confidentiality, integrity, non-interference, type systems.

1. INTRODUCTION

Information flow security is a well-established, high-level framework for reasoning about confidentiality and integrity, with a clear separation between security specifications and mechanisms. At a lower level, encryption provides essential mechanisms for confidentiality, with a wide range of algorithms reflecting different trade-offs between security, functionality, and efficiency. Thus, the secure usage of adequate algorithms for a particular system is far from trivial.

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

Even with plain encryption, the confidentiality and integrity of keys, plaintexts, and ciphertexts are interdependent: encryption with untrusted keys is clearly dangerous, and plaintexts should never be more secret than their decryption keys. Integrity also matters: attackers may swap ciphertexts, and thus cause the declassification of the wrong data after their successful decryption. Conversely, to protect against chosen-ciphertext attacks, it may be necessary to authenticate ciphertexts, even when plaintexts are untrusted.

Modern encryption schemes offer useful additional features, such as the ability to blind ciphertexts, thereby hiding dependencies between encrypted inputs and outputs, and more generally to compute homomorphically on encrypted data, for instance by multiplying ciphertexts instead of adding their plaintexts. These features are naturally explained in terms of information flows; they enable computations at a lower level of confidentiality—homomorphic operations can be delegated to an "honest but curious" principal—but they also exclude CCA2 security and require some care in the presence of active adversaries.

In this paper, we develop an information-flow type system for cryptography, with precise typing rules to reflect the diverse functional and security features of encryption schemes. Our goal is to understand them better, and to guide protocol designers and programmers. We rely on standard cryptographic assumptions, expressed as probabilistic polynomial-time indistinguishability games —this level of detail is necessary to reliably capture the information flows of the underlying algorithms. Thus, our adversaries are probabilistic polynomial-time programs, with limited read and write access to data, that attempt to gain information about higher-confidentiality data, or to influence higher-integrity data, by interacting with our programs. Our main theorem states that well typed probabilistic polynomial programs using any combination of encryptions, blinding, homomorphic functions, and decryptions are such that our adversaries succeed only with a negligible probability. Depending on the relative security levels of keys, plaintexts, and ciphertexts, we propose different typing rules. Our rules are sound with regards to standard cryptographic assumptions such as CPA or CCA2; they enable the selective re-use of keys for protecting different types of payloads, as well as blinding and homomorphic properties.

Secure Distributed Computations Our work is part of a research project on the synthesis and verification of distributed cryptographic implementations of programs from a description of their information flow security requirements. Fournet et al. [10] present such a security compiler that automatically generates code for encryption and authentication. As illustrated in our programming examples, an important motivation for a new type system is to justify its efficient use of cryptography. For instance, we strive to re-use the

same keys for protecting different types of data at different levels of security, so that we can reduce the overall cost of cryptographic protection. More generally, we would like to automatically generate well-typed code for performing blind computations, such as those supported by the systems of Henecka et al. [14] and Katz and Malka [15].

Formally, our work is based on the type system of Fournet and Rezk [9], who introduce the notion of *computational non-interference* against active adversaries to express information flow security in probabilistic polynomial-time cryptographic systems and present a basic type system for encryption and authentication mechanisms. In comparison, their typing rules are much more restrictive, and only support CCA2 public-key encryptions with a single payload type.

Applications We illustrate our approach using programming examples based on classic encryption schemes with homomorphic properties [8, 22]. We also develop two challenging applications of our approach. Both applications rely on a security lattice with intermediate levels, reflecting the structure of their homomorphic operations, and enabling us to prove confidentiality properties, both for honest-but-curious servers and for compromised servers controlled by an active attacker.

- We program and typecheck a practical protocol for private search on data streams proposed by Ostrovsky and Skeith III [21], based on a Paillier encryption of the search query. This illustrates that our types protect against both explicit and implicit information flows; for instance we crucially need to apply some blinding operations to hide information about secret loop indexes.

- We program and typecheck the bootstrapping part of Gentry's fully homomorphic encryption [11]. Starting from the properties of the bootstrappable algorithms given by Gentry–being CPA and homomorphic for its own decryption and for some basic operations—we obtain an homomorphic encryption scheme for an arbitrary function. This illustrates three important features of our type system: the ability to encrypt decryption keys (which is also important for typing key establishment protocols); the use of CPA encryption despite some chosen-ciphertext attacks; and an interesting instance of homomorphism where the homomorphic function is itself a decryption.

This paper exclusively treats public-key encryptions. We believe that their symmetric-key counterparts can be treated similarly. We also refer to Fournet and Rezk [9] for cryptographic types for authentication primitives, such as public-key signatures. This paper omits many details and all proofs. The full paper, programming examples, and a prototype typechecker for our system are available online at `msr-inria.inria.fr/projects/sec/cflow`.

2. PROGRAMS, TYPES, AND POLICIES

We use an imperative probabilistic WHILE language with security policies. A probabilistic semantics is necessary for modeling encryption security [13]. Shared memory is sufficient to model a wide range of interactions between programs and adversaries; for instance public untrusted variables model an open network.

Language The grammar for expressions and commands is

$$\begin{array}{rcl} e & ::= & x \mid op(e_1, \ldots, e_n) \\ P & ::= & x := e \mid x_1, \ldots, x_m := f(e_1, \ldots, e_n) \mid \\ & & P; P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mathsf{while}\ e\ \mathsf{do}\ P \mid \mathsf{skip} \end{array}$$

$$\textsc{AssignS}\ \frac{[\![e]\!](\mu) = v}{\langle x := e, \mu\rangle \rightsquigarrow_1 \langle\surd, \mu\{x \mapsto v\}\rangle}$$

$$\textsc{SeqS}\ \frac{\langle P, \mu\rangle \rightsquigarrow_p \langle P_1, \mu_1\rangle \qquad P_1 \neq \surd}{\langle P; P', \mu\rangle \rightsquigarrow_p \langle P_1; P', \mu_1\rangle} \qquad \textsc{SeqT}\ \frac{\langle P, \mu\rangle \rightsquigarrow_p \langle\surd, \mu_1\rangle}{\langle P; P', \mu\rangle \rightsquigarrow_p \langle P', \mu_1\rangle}$$

$$\textsc{SkipS}\ \langle \mathsf{skip}, \mu\rangle \rightsquigarrow_1 \langle\surd, \mu\rangle \qquad \textsc{Stable}\ \langle\surd, \mu\rangle \rightsquigarrow_1 \langle\surd, \mu\rangle$$

$$\textsc{CondTrue}\ \frac{[\![e]\!](\mu) \neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P', \mu\rangle \rightsquigarrow_1 \langle P, \mu\rangle}$$

$$\textsc{CondFalse}\ \frac{[\![e]\!](\mu) = 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P', \mu\rangle \rightsquigarrow_1 \langle P', \mu\rangle}$$

$$\textsc{WhileTrue}\ \frac{[\![e]\!](\mu) \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ P, \mu\rangle \rightsquigarrow_1 \langle P; \mathsf{while}\ e\ \mathsf{do}\ P, \mu\rangle}$$

$$\textsc{WhileFalse}\ \frac{[\![e]\!](\mu) = 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ P, \mu\rangle \rightsquigarrow_1 \langle\surd, \mu\rangle}$$

$$\textsc{Fun}\ \frac{p = [\![f]\!](\mu(y_1), \ldots, \mu(y_n))(\vec{v}) \qquad p > 0}{\langle \vec{x} := f(y_1, \ldots, y_n), \mu\rangle \rightsquigarrow_p \langle\surd, \mu\{\vec{x} \mapsto \vec{v}\}\rangle}$$

Figure 1: Probabilistic operational semantics

where op and f range over polynomial n-ary deterministic and probabilistic functions respectively, with arity $n \geq 0$.

Expressions e consist of variables and operations on data represented as bitstrings. We write op for nullary operations $op()$. We assume given standard functions for boolean and arithmetic constants ($0, 1, \ldots$) and operators ($||, +, \ldots$). We let $=_0$ be comparison on booleans, true when both its arguments are either 0 or non-0. We also have standard functions for pairs and functional arrays: we use $\langle e_0, e_1\rangle$ and $(e)_i$ for constructing and projecting pairs, and use $e[e_i]$ and $\mathrm{update}(x, e, e_i)$ for reading and updating arrays.

Commands P consist of variable assignments, using deterministic expressions and probabilistic functions (such as encryptions or key generations) composed into sequences, conditionals, and loops. We use syntactic sugar for arrays, writing $x[e_i] := e$ instead of $x := \mathrm{update}(x, e, e_i)$, and for loops, writing $\mathsf{for}\ x := e\ \mathsf{to}\ e'\ \mathsf{do}\ P$ instead of $x := e; \mathsf{while}\ x \leq e'\ \mathsf{do}\ \{P; x := x + 1\}$.

Semantics Configurations (s) range over pairs of a command and a memory, written $\langle P, \mu\rangle$, plus inert configurations, written $\langle\surd, \mu\rangle$, that represent termination with final memory μ.

Probabilistic reductions between distributions of configurations are based on a probabilistic transition relation $\rightsquigarrow_p$ defined in Figure 1. We lift these reduction steps to configuration distributions (d), and write $d \rightsquigarrow d'$ when, for all configurations s', we have $d'(s') = \sum_{s \rightsquigarrow_p s'} p \times d(s)$. We denote by $\rho_\infty(\langle P, \mu\rangle)$ the final distribution of memories after running program P with initial memory μ. We write $\Pr[\langle P, \mu\rangle; \varphi]$ for the probability that P terminates with a final memory that meets condition φ.

Confidentiality and Integrity We annotate variables, types, and commands with security labels. These labels specify the programmer's security intent, but they do not affect the behavior of programs. The security labels form a lattice $(\mathcal{L}, \leq)$ obtained as the

product of two lattices, for confidentiality $(\mathcal{L}_C, \leq_C)$ and for integrity $(\mathcal{L}_I, \leq_I)$. We write $\bot_\mathcal{L}$ and $\top_\mathcal{L}$ for the smallest and largest elements of $\mathcal{L}$, and $\sqcup$ and $\sqcap$ for the least upper bound and greatest lower bound of two labels, respectively. We write $\bot_C$, $\bot_I$, $\top_C$, and $\top_I$ for the smallest and largest elements of $\mathcal{L}_C$ and $\mathcal{L}_I$, respectively.

For a given label $\ell = (\ell_C, \ell_I)$ of $\mathcal{L}$, the confidentiality label ℓ_C specifies a read level for variables, while the integrity label ℓ_I specifies a write level; the meaning of $\ell \leq \ell'$ is that ℓ' is at least as confidential (can be read by fewer entities) and at most as trusted (can be written by more entities) than ℓ [20]. We let $C(\ell) = \ell_C$ and $I(\ell) = \ell_I$ be the projections that yield the confidentiality and integrity parts of a label. We overload $\leq_C$ and $\leq_I$, writing $\ell \leq_C \ell'$ for $C(\ell) \leq_C C(\ell')$ and $\ell \leq_I \ell'$ for $I(\ell) \leq_I I(\ell')$. Hence, the partial order on $\mathcal{L}$ is defined as $\ell \leq \ell'$ iff $\ell \leq_C \ell'$ and $\ell \leq_I \ell'$. In examples, we often use a four-point lattice defined by $LH < HH < HL$ and $LH < LL < HL$, where for instance LH stands for low confidentiality and high integrity.

Types for Information-Flow Security We use the following grammar for security types:

$\tau ::= t(\ell) \mid \tau * \tau \mid \mathsf{Array}\,\tau$	Security types
$t ::= \mathsf{Data}$	Payload Data types
$\mid \mathsf{Enc}\,\tau\,K q \mid \mathsf{Ke}\,\mathsf{E}\,K \mid \mathsf{Kd}\,\mathsf{E}\,K$	Encryption Data types

where $\ell \in \mathcal{L}$ is a security label, E is a set of security types, K is a key label, and q is an encryption index, as explained below. We have pairs at the level of the security types to keep track of tuples of values with different labels, e.g. for encrypting tuples.

Let L be the projection from security types to labels defined by $L(\tau * \tau') = L(\tau) \sqcap L(\tau')$, $L(\mathsf{Array}\,(\tau)) = L(\tau)$, and $L(t(\ell)) = \ell$. We overload $\leq$ for subtyping on security types (e.g. $t(\ell) \leq t(\ell')$ if $\ell \leq \ell'$) and as a relation between security types and labels (e.g. $t(\ell) \leq \ell$). We overload $\sqcup$ from labels to types (e.g. $t(\ell') \sqcup \ell = t(\ell' \sqcup \ell)$). Memory policies are functions Γ from variables to security types. For a given policy Γ, we overload $\leq$ as a relation between variables, security types, and labels (e.g. $x \leq \ell$ if $\Gamma(x) \leq \ell$).

We explain our datatypes for encryptions, but defer their typing rules to the next sections. $\mathsf{Enc}\,\tau\,K$ represents an encryption of a plaintext with security type τ; $\mathsf{Ke}\,\mathsf{E}\,K$ and $\mathsf{Kd}\,\mathsf{E}\,K$ represent encryption and decryption keys, respectively, with a set E that indicates the range of security types τ for the plaintexts that may be encrypted and decrypted with these keys. This set enables us to type code that uses the same key for encrypting values of different types, which is important for efficiency.

The key labels K are used to keep track of keys, grouped by their key-generation commands. In a given program, there should be only one keypair generation for a given label. These labels are attached to the types of the generated keypairs, and propagated to the types of any derived cryptographic materials. They are used to match the usage of key pairs and to prevent key cycles.

The encryption indexes q are used to distinguish between different datatypes for encryptions, for instance when they range over different groups before and after some homomorphic operations. In all other cases, we use a single, implicit index $q = 0$.

Active adversaries For a given $\alpha \in \mathcal{L}$, we let α-adversaries range over commands A that read variables in a set $V_\alpha^C = \{x \mid x \leq_C \alpha\}$ and write variables outside a set $V_\alpha^I = \{x \mid x \leq_I \alpha\}$, that is, $rv(A) \subseteq V_\alpha^C$ and $wv(A) \cap V_\alpha^I = \emptyset$. We consider programs obtained by composing commands with diverse levels of trust, including any α-adversaries as well as fixed, trusted commands. To this end, we write $P[\_]$ for a command context (with a grammar obtained from that of P by adding a hole _) and $P[P']$ for the command obtained by replacing each occurrence of _ with P'.

$$\textsc{Var}\ \vdash x : \Gamma(x) \qquad \textsc{Op}\ \frac{op : \tau_1 \ldots \tau_n \to \tau \quad \vdash e_i : \tau_i \ \text{ for } i = 1..n}{\vdash op(e_1, \ldots, e_n) : \tau} \qquad \textsc{SubE}\ \frac{\vdash e : \tau \quad \tau \leq \tau'}{\vdash e : \tau'}$$

$$\textsc{Assign}\ \frac{\vdash e : \Gamma(x)}{\vdash x := e : L(x)} \qquad \textsc{ProbFun}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \ \text{ for } e \in \vec{e} \quad \mathsf{Data}\,(\ell) \leq \Gamma(x) \ \text{ for } x \in \vec{x}}{\vdash \vec{x} := f(\vec{e}) : \ell}$$

$$\textsc{Seq}\ \frac{\vdash P : \ell \qquad \vdash P' : \ell}{\vdash P;\, P' : \ell} \qquad \textsc{Cond}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \quad \vdash P : \ell \qquad \vdash P' : \ell}{\vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' : \ell}$$

$$\textsc{While}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \qquad \vdash P : \ell}{\vdash \mathsf{while}\ e\ \mathsf{do}\ P : \ell} \qquad \textsc{SubC}\ \frac{\vdash P : \ell \qquad \ell' \leq \ell}{\vdash P : \ell'} \qquad \textsc{Skip}\ \vdash \mathsf{skip} : \top^\mathcal{L}$$

$$\textsc{Hole}\ \vdash \_ : (C(\alpha), \top_I)$$

Figure 2: Non-cryptographic typing rules with policy Γ.

Figure 2 defines a type system that prevents information flows in command contexts that do not rely on cryptography: two runs of the command obtained by inserting α-adversaries, starting with initial memories that coincide on all variables in V_α^C (resp. outside V_α^I), yield final memory distributions that also coincide on those variables. The typing rules are standard, except for Rule HOLE, which safely accounts for any α-adversary, and Rule PROBFUN, which is a probabilistic version of OP. In Rule OP, the first hypothesis gives the type signature of *op*:

$$\begin{array}{rl}
op & : \mathsf{Data}\,(\ell) * \cdots * \mathsf{Data}\,(\ell) \to \mathsf{Data}\,(\ell) \\
\langle\rangle & : \tau_0, \tau_1 \to (\tau_0 * \tau_1) \\
()_i & : (\tau_0 * \tau_1) \to \tau_i \\
[] & : \mathsf{Array}\,(\tau) \\
\text{cell} & : \mathsf{Array}\,(\tau), \mathsf{Data}\,(L(\tau)) \to \tau \\
\text{update} & : \mathsf{Array}\,(\tau), \tau, \mathsf{Data}\,(L(\tau)) \to \mathsf{Array}\,(\tau) \\
+ & : \mathsf{Array}\,(\tau), \tau \to \mathsf{Array}\,(\tau) \\
\text{size} & : \mathsf{Array}\,(\tau) \to \mathsf{Data}\,(L(\tau))
\end{array}$$

We assume that all primitive operations run in polynomial time in the size of their arguments, and that all primitive probabilistic functions yield distributions that are polynomial-time samplable. As we use cryptographic primitives, we assume that they take an additional parameter included in the initial memory whose length matches a security parameter η. We overload $\Pr[\langle P, \mu\rangle; \varphi]$ to denote a probability function parameterized by η. This function is deemed negligible when, for all $c > 0$, there exists η_c such that, for all $\eta \geq \eta_c$, we have $\Pr[\langle P, \mu\rangle; \varphi] \leq \eta^{-c}$.

3. TYPING ENCRYPTIONS

We consider cryptographic algorithms for asymmetric (public key) encryption. We model them in our language as probabilistic functions $\mathcal{G}_e$, $\mathcal{E}$, and $\mathcal{D}$ that meet both functional and security properties, given below.

DEFINITION 1 (ENCRYPTION SCHEME). *Let pubkeys, seckeys, plaintexts, and ciphertexts be sets of polynomially-bounded bitstrings indexed by η. An asymmetric encryption scheme is a triple of algorithms $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ such that*

1. *$\mathcal{G}_e$, used for* key generation*, ranges over pubkeys $\times$ seckeys;*

2. *$\mathcal{E}$, used for* encryption*, ranges over ciphertexts;*

3. *$\mathcal{D}$, used for* decryption*, ranges over plaintexts;*

4. *for all $k_e, k_d := \mathcal{G}_e()$, if $y := \mathcal{E}(m, k_e)$ with $m \in$ plaintexts, then $\mathcal{D}(y, k_d) = m$.*

Next, we recast the usual game-based definition of resistance against chosen plaintext attacks in our setting.

DEFINITION 2 (CPA SECURITY). *The scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ provides indistinguishability against adaptive chosen-plaintexts attacks when, for the commands*

$$E \doteq \mathsf{if}\ b = 0\ \mathsf{then}\ m := \mathcal{E}(x_0, k_e)\ \mathsf{else}\ m := \mathcal{E}(x_1, k_e)$$
$$\mathsf{CPA} \doteq b := \{0,1\}; k_e, k_d := \mathcal{G}_e(); A[E]$$

and for any polynomials command context A such that $b, k_d \notin rv(A)$ and $b, k_d, k_e, \eta \notin wv(A)$, the advantage of the adversary defined as $|\Pr[\mathsf{CPA}; b =_0 g] - \frac{1}{2}|$ is negligible.

The definition involves an indistinguishability game where the adversary command A attempts to guess (and write in variable g) whether the oracle encrypts x_0 or x_1. This notion of security does not involve the decryption algorithm. In particular, it does not cover chosen-ciphertext attacks.

Typing CPA encryptions Figure 3 gives cryptographic rules for typing the algorithms of a CPA encryption scheme.

In Rule GENE for key generation, the first hypothesis requires that the encryption scheme (implicitly parameterized by the key label K) be at least CPA secure. The next two hypotheses give matching encryption-key and decryption-key types to variables k_e and k_d, with the same key label K and the same range of plaintexts E. The constraint $\tau \leq_C \ell_d$ for every type $\tau \in \mathsf{E}$ states that the decryption key k_d is at least as confidential as every plaintext, thereby preventing confidentiality leaks by key compromise. (The constraint appears in GENE rather than DECRYPT, to ensure that all copies of the decryption key are sufficiently confidential.)

In Rule ENCRYPT, the first three hypotheses bind types to the ciphertext y, plaintext e, and key k_e involved in the encryption; these types are related by $\tau \in \mathsf{E}$ and K. The type of y carries two security labels: ℓ_y, the level of the encryption, and $L(\tau)$ within $\mathsf{Enc}\ \tau\ K$, the level of the encrypted plaintext. The label ℓ_y in the typing of k_e records an ordinary flow from k_e to y: by subtyping, we must have $L(k_e) \leq \ell_y$. Conversely, there is no constraint on the flow from e to y, so our typing rule may be sound only with cryptographic assumptions: for confidentiality, this flow reflects that encryption is a form of declassification: e can be more confidential than y. In that case, that is $\tau \not\leq_C y$, we say that K *declassifies* $C(\tau)$. For integrity, this flow reflects that encryption is also a form of endorsement: intuitively, y is only a carrier for e, not in itself an observable outcome of the program; its integrity indicates that y is the result of a correct encryption, even when e itself is not trusted. Finally, the disjunction is a robustness condition on the encryption key, requiring that its integrity be sufficient to protect the confidentiality of the plaintext. The first disjunct excludes that an α-adversary may affect the encryption key (for instance overwriting it with a key she knows). The second disjunct states that the confidentiality protection provided by the key is nil, since an α-adversary may directly read the plaintext. Hence, in case E contains plaintexts with different levels of secrecy, a key with relatively low integrity may still be used to encrypt a plaintext with relatively low confidentiality.

In Rule DECRYPT, the first three hypotheses on the left bind types to the variables x, y, and k_d involved in the decryption; these types are related by τ and K. The label $L(x)$ flows from y to x, as in a normal assignment. The label ℓ_d and the hypothesis $\ell_d \leq_I x$ record integrity flows from k_d to x, as in a normal assignment. Conversely, there is no constraint on the confidentiality flow from k_d to the plaintext x, so our typing rule may be sound only with cryptographic assumptions. Informally, this reflects that the decrypted plaintext yield no information on the decryption key itself, thereby enabling decryptions of plaintexts with different confidentiality levels. When $\ell_d \not\leq_C x$, we say that K *declassifies* $C(\ell_d)$.

The disjunction deals with the integrity of the ciphertext y and the decryption key k_d, with three cases: either

- both ciphertext and decryption key have high integrity; or
- the plaintext is at least as confidential as the key; or
- cryptographic protection is nil, since an α-adversary may read the decryption key then decrypt the plaintext.

The first case intuitively reflects our CPA security assumptions: it requires that the integrity of the ciphertext and the decryption key suffices to guarantee a correct decryption.

When $y \not\leq_I \alpha$ and $\ell_d \not\leq_C \alpha$, the second case covers some chosen-ciphertext attacks, which may come as a surprise: in that case, we say that K *depends on* $C(x)$ and we rely on an additional safety condition (see also Example 3 and 4). When $\ell_d \not\leq_I \alpha$ and $\ell_d \not\leq_C \alpha$, the second case also cover compromise of the decryption key, which may then flow to the decrypted value, as discussed in Example 2.

Examples We provide a series of examples that illustrate the relative information-flow security properties of encryptions. In the examples, unless specified otherwise, we assume that α is *LH* and k_e, k_d is a secure keypair, that is, we set $L(k_e) = LH$ and $L(k_d) = HH$ and assume that $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ is CPA. We also write for instance x_{LH} for a variable such that $L(x_{LH}) = LH$. We first encrypt a pair of mixed integrity into a ciphertext of high integrity:

EXAMPLE 1 (HIGHER-INTEGRITY ENCRYPTIONS). *Consider the command context*

$$k_e, k_d := \mathcal{G}_e(); x := \langle x_{HH}, x_{HL}\rangle; y_{LH} := \mathcal{E}(x, k_e); \_;$$
$$x' := \mathcal{D}(y_{LH}, k_d); x'_{HH} := (x')_0; x'_{HL} := (x')_1$$

Let $\Gamma(x) = \Gamma(x') = \mathsf{Data}\,(HH) * \mathsf{Data}\,(HL)$. This command typechecks; it is safe, inasmuch as the adversary can influence the values of x_{HL}, x'_{HL} and of the ciphertext y_{LH}, but not the final value of x'_{HH}.

Encrypting with a low-integrity key clearly leads to confidentiality leaks, as the adversary may overwrite the key with its own key before the encryption. Decrypting with a low-integrity decryption key may also be problematic, especially when the plaintext is not secret, as illustrated below:

EXAMPLE 2 (LOW-INTEGRITY KEYS). *Consider the command context*

$$k_e, k_d := \mathcal{G}_e(); k'_d := k_d; y_{LH} := \mathcal{E}(1, k_e); \_;$$
$$\mathsf{if}\ x_{HH}\ \mathsf{then}\ k''_d := k_d\ \mathsf{else}\ k''_d := k'_d; x_{LL} := \mathcal{D}(y_{LH}, k''_d)$$

We let k'_d and k''_d be low-integrity copies of the decryption key, that is $L(k'_d) = L(k''_d) = HL$. The command is not typable, and is actually unsafe, since we may fill the hole with the command $k'_d := 0$; thus, k''_d to contain the correct key if and only if $x_{HH} \neq 0$, and the adversary can finally compare x_{LL} with 1, the correct decryption, and infer the confidential value x_{HH}.

$$\textsc{GenE}\quad \frac{(\mathcal{G}_e,\mathcal{E},\mathcal{D}) \text{ is CPA} \quad \Gamma(k_e)=\mathsf{Ke}\,\mathsf{E}\,K(\ell_e) \quad \Gamma(k_d)=\mathsf{Kd}\,\mathsf{E}\,K(\ell_d) \quad \forall \tau\in\mathsf{E}, \tau\le_C \ell_d}{\vdash k_e,k_d:=\mathcal{G}_e()\ :\ell_e\sqcap\ell_d}$$

$$\textsc{Encrypt}\quad \frac{\Gamma(y)=\mathsf{Enc}\,\tau K(\ell_y) \quad \vdash e:\tau \quad \tau\in\mathsf{E} \quad \vdash k_e:\mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad k_e\le_I\alpha \ \textbf{or}\ \tau\le_C\alpha}{\vdash y:=\mathcal{E}(e,k_e)\ :\ell_y}$$

$$\textsc{Decrypt}\quad \frac{\tau\le\Gamma(x) \quad \tau\in\mathsf{E} \quad \vdash y:\mathsf{Enc}\,\tau K(L(x)) \quad \vdash k_d:\mathsf{Kd}\,\mathsf{E}\,K(\ell_d) \quad \ell_d\le_I x \quad (y\le_I\alpha \textbf{ and } \ell_d\le_I\alpha) \textbf{ or } \ell_d\le_C x \textbf{ or } \ell_d\le_C\alpha}{\vdash x:=\mathcal{D}(y,k_d)\ :L(x)}$$

Figure 3: Typing rules for CPA encryption with policy Γ.

Similarly, since we rely on CPA (for chosen plaintext) and not CCA (for chosen ciphertext), decrypting a low-integrity ciphertext may also be problematic, as illustrated below:

EXAMPLE 3 (CHOSEN-CIPHERTEXT ATTACKS). *When decrypted, low-integrity ciphertexts may leak information about their decryption keys. For a given CPA encryption scheme $(\mathcal{G}_e,\mathcal{E},\mathcal{D})$, we derive a new scheme $(\mathcal{G}'_e,\mathcal{E}',\mathcal{D}')$ as follows:*

$$\mathcal{G}'_e() \doteq \mathcal{G}_e() \qquad \mathcal{E}'(x,k_e) \doteq 1|\mathcal{E}(x,k_e)$$
$$\mathcal{D}'(y,k_d) \doteq b|y' := y; \mathsf{if}\ b=0\ \mathsf{then}\ k_d\ \mathsf{else}\ \mathcal{D}(y',k_d)$$

(where $b|y' := y$ abbreviates decomposing y into b and y'). In the new scheme, decryption leaks its key when called on an ill-formed ciphertext (prefixed with a 0 instead of a 1).

Although this scheme is CPA, decryption of a low-integrity ciphertext may cause a confidentiality flow from k_d to the decrypted plaintext, letting the adversary decrypt any other value encrypted under k_e.

CPA versus chosen ciphertext attacks As illustrated in Sections 7 and 8, our typing rules for CPA allow the decryption of low-integrity ciphertexts when the plaintext is secret, as long as the resulting plaintext never flows to any cryptographic declassification.

We define a notion of key dependencies, used to express the absence of key cycles in Section 6, and we give a counter-example showing an implicit information flow for one such cycle.

DEFINITION 3 (KEY DEPENDENCIES). *For given policy, adversary, and typed command context P, K* depends on *K' when*

1. *K occurs in the set E' of plaintext types for K'; or*
2. *K depends on c, $c\le_C c'$, and K' declassifies c'.*

EXAMPLE 4. *For a given CPA encryption scheme $(\mathcal{G}_e,\mathcal{E},\mathcal{D})$, we define a modified scheme $(\mathcal{G}_e,\mathcal{E}',\mathcal{D}')$ as follows:*

$$\mathcal{E}'(x,k_e) \doteq r := random\_plaintext();\ \mathsf{if}\ \mathcal{D}(\mathcal{E}(r,k_e),x)=r\ \mathsf{then}\ 0|x\ \mathsf{else}\ 1|\mathcal{E}(x,k_e)$$
$$\mathcal{D}'(b|y',k_d) \doteq \mathsf{if}\ b=0\ \mathsf{then}\ k_d\ \mathsf{else}\ \mathcal{D}(y',k_d)$$

In the modified scheme, the encryption of a decryption key with the corresponding encryption key leaks the decryption key. This scheme is still CPA since an adversary would have to guess the decryption key for a successful attack. However, this scheme leaks k_d when the decryption of a compromised plaintext appears in an encryption:

$$k_e,k_d := \mathcal{G}_e(); y_{LL} := \mathcal{E}(0,k_e); \_;$$
$$x_{HL} := \mathcal{D}(y_{LL},k_d); y'_{LL} := \mathcal{E}(x_{HL},k_e);$$

This command has a key-dependency cycle, but it is otherwise typable, and it is unsafe: we may fill the hole with a command injecting an ill-formed ciphertext; thus $y'_{LL} = x_{HL} = k_d$ and the adversary can obtain the private key by reading y'_{LL}.

Typing CCA2 encryptions We recall the definition of CCA2, obtained from CPA by adding a decryption oracle:

DEFINITION 4 (CCA2 SECURITY). *The scheme $(\mathcal{G}_e,\mathcal{E},\mathcal{D})$ provides indistinguishability against adaptive chosen-plaintext and chosen-ciphertext attacks when, for the commands*

$$E \doteq \mathsf{if}\ b=0\ \mathsf{then}\ m:=\mathcal{E}(x_0,k_e)\ \mathsf{else}\ m:=\mathcal{E}(x_1,k_e);\ log := log+m$$
$$D \doteq \mathsf{if}\ m\in log\ \mathsf{then}\ x:=0\ \mathsf{else}\ x:=\mathcal{D}(m,k_d)$$
$$CCA \doteq b:=\{0,1\}; log:=\mathsf{nil}; k_e,k_d:=\mathcal{G}_e(); A[E,D]$$

and for any polynomial command context A such that $b,k_d\notin rv(A)$ and $b,k_d,k_e,\eta,log\notin wv(A)$, the advantage of the adversary $|\Pr[CCA; b=_0 g]-\frac{1}{2}|$ is negligible.

If the scheme is CCA2 secure, we may use an additional rule for typing decryptions, given below.

$$\textsc{Decrypt CCA2}\quad \frac{\tau\le\Gamma(x) \quad \tau\in\mathsf{E} \quad (\mathcal{G}_e,\mathcal{E},\mathcal{D}) \text{ is CCA2} \quad \vdash y:\mathsf{Enc}\,\tau K(L(x)) \quad \vdash k_d:\mathsf{Kd}\,\mathsf{E}\,K(\ell_d) \quad \ell_d\le_I x \quad \ell_d\le_I\alpha \textbf{ or } \ell_d\le_C\alpha \quad \forall\tau'\in\mathsf{E}, \tau'\le\tau\sqcup(\bot,\top)}{\vdash x:=\mathcal{D}(y,k_d)\ :L(x)}$$

Except for the CCA2 cryptographic assumption, the rule differs from rule DECRYPT only on the last line of hypotheses, so DECRYPT CCA2 effectively adds a fourth case for decryptions of low-integrity ciphertexts. In this new case, the adversary may be able to mix encryptions for values of type τ with those of values of any other type $\tau'\in\mathsf{E}$, so we must statically exclude some of the resulting flows between plaintexts: τ and τ' must have the same datatypes at the same levels of confidentiality.

EXAMPLE 5 (CIPHERTEXT REWRITING). *Consider a command context using the same keypair for payloads at levels HH and LL:*

$$k_e,k_d := \mathcal{G}_e();$$
$$y_{LL} := \mathcal{E}(x_{LL},k_e); y'_{LL} := \mathcal{E}(x_{HH},k_e); \_; x'_{LL} := \mathcal{D}(y_{LL},k_d)$$

The command is not typable, and is unsafe: indeed, we may fill the hole with an adversary command $y_{LL} := y'_{LL}$, thereby causing the program to leak a copy of the secret x_{HH} into x'_{LL}.

4. BLINDING SCHEMES: SECURITY AND TYPING

Blinding schemes, also known as reencryption schemes, have been introduced by Blaze et al. [3] and can be seen as special cases of homomorphic encryption schemes. To precisely keep track of their information flows, we separate encryption into two stages, each with its own primitive and typing rule:

- Pre-encryption $\mathcal{P}()$ inputs a plaintext and outputs its representation as a ciphertext, but does not in itself provides confidentiality; it can be deterministic; it is typed as an ordinary operation and does not involve declassification.
- Blinding $\mathcal{B}()$ operates on ciphertexts; it hides the correlation between its input and its output, by randomly sampling another ciphertext that decrypts to the same plaintext; it is typed with a declassification, similarly to rule ENCRYPT.

As shown below, some standard encryption schemes can easily be decomposed into pre-encryption and blinding. This enables us for instance to blind a ciphertext without knowing its plaintext, and to perform multiple operations on ciphertexts before blinding. For conciseness, we may still write $\mathcal{E}$ instead of $\mathcal{P};\mathcal{B}$ when the two operations are executed together. We define the functional properties of blinding encryption schemes:

DEFINITION 5 (BLINDING SCHEME). *A blinding encryption scheme is a tuple $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D})$ such that $(\mathcal{G}_e, \mathcal{P}; \mathcal{B}, \mathcal{D})$ is an encryption scheme and $\mathcal{B}$ is a probabilistic function such that, for all $k_e, k_d := \mathcal{G}_e()$, if v encrypts m, then $\mathcal{D}(v, k_d) = m$, where 'encrypts' is defined by*

1. *v encrypts m when $v := \mathcal{B}(\mathcal{P}(m, k_e))$ with $m \in$ plaintexts;*
2. *v' encrypts m when $v' := \mathcal{B}(v, k_e)$ and v encrypts m.*

Blinding hides whether an encrypted value is a copy of an other, as shown in the following example.

EXAMPLE 6. *Consider a service that, depending on a secret, either forwards or overwrites an encrypted message. We distinguish a third confidentiality level S such that $L \leq_C S \leq_C H$.*

$$k_e, k_d := \mathcal{G}_e(); y_{LH} := \mathcal{E}(m_{HH}, k_e);$$
$$\text{if } s_{SH} \text{ then } y_{SH} := \mathcal{P}(m_{SH}, k_e) \text{ else } y_{SH} := y_{LH};$$
$$y'_{LH} := \mathcal{B}(y_{SH}, k_e)$$

The resulting ciphertext y_{SH} is itself secret, as an adversary may otherwise compare it with y_{LH} and infer the value of s_{HH}. After blinding, however, y'_{LH} still protects the same message but can be safely treated as public, as an adversary reading y_{LH} and y'_{LH} learns nothing about m_{HH} or s_{SH}.

CPA for Blinding A blinding scheme $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D})$ is secure enough to be used with our typing rules when $(\mathcal{G}_e, \mathcal{B}, \mathcal{D})$ is CPA. We have shown this property for the blinding schemes based on ElGamal [8] $(\mathcal{G}_e^{\mathsf{E}}, \mathcal{P}^{\mathsf{E}}, \mathcal{B}^{\mathsf{E}}, \mathcal{D}^{\mathsf{E}})$, and Paillier [22] $(\mathcal{G}_e^{\mathsf{P}}, \mathcal{P}^{\mathsf{P}}, \mathcal{B}^{\mathsf{P}}, \mathcal{D}^{\mathsf{P}})$.

THEOREM 1. *$(\mathcal{G}_e^{\mathsf{E}}, \mathcal{B}^{\mathsf{E}}, \mathcal{D}^{\mathsf{E}})$ and $(\mathcal{G}_e^{\mathsf{P}}, \mathcal{B}^{\mathsf{P}}, \mathcal{D}^{\mathsf{P}})$ are CPA.*

Typing rules for Pre-Encryption and Blinding The typing rules for pre-encryption and blinding appear in Figure 4.

Rule PRE-ENCRYPT is similar to ENCRYPT, but constrains the confidentiality flow from e to y, which forbids any declassification by typing.

The first three hypotheses of the Rule BLIND bind types to the variables k_e, z, and y. These types are related by $\tau, \tau' \in \mathsf{E}$ and K, which are also typing assumptions for encryptions with key k_e. The label ℓ_y in the typing of k_e records the flow from k_e to y (by subtyping, we must have $k_e \leq \ell_y$). The label $L(\tau)$ in the typing of z records the flow from z to the encrypted value in y. The hypothesis $\tau \leq \tau'$ ensure the correctness of the flow from the encrypted value in z to the encrypted value in y. Similarly to ENCRYPT, there is no constraint on the flow from z to y, so our typing rule may be sound only with cryptographic assumptions. When $z \not\leq_C y$, we say that K *declassifies* $C(z)$.

5. HOMOMORPHIC ENCRYPTIONS

We now consider encryption schemes with homomorphic properties: some functions on plaintexts can instead be computed on their ciphertexts, so that the command that performs the computation may run at a lower level of confidentiality. These schemes enable private remote evaluation: supposing that f_K is a function that homomorphically compute a function f, a client may delegate its evaluation to a server as follows:

1. the client encrypts the secret plaintext x into z_1;
2. the server applies f_K to the encrypted value (possibly encrypting its own secret inputs);
3. the client decrypts the result.

Programmatically, to implement $x' := f(x)$, we use a sequence of three commands sharing the variables z and z' and the encryption key k_e:

$$z := \mathcal{E}(x, k_e); z' := f_K(z, k_e); x' := \mathcal{D}(z', k_d)$$

For simplicity, we do not consider probabilistic homomorphic functions, or homomorphic functions that take non-ciphertext arguments.

DEFINITION 6 (HOMOMORPHIC ENCRYPTION SCHEME). *An homomorphic encryption scheme is a tuple $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D}, \mathcal{F})$ such that $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D})$ is a blinding encryption scheme and $\mathcal{F}$ is a partial map on polynomial deterministic functions such that for all $k_e, k_d := \mathcal{G}_e()$, if v encrypts$_q$ m, then $\mathcal{D}(v, k_d) = m$, where 'encrypts' is defined by*

1. *v encrypts$_0$ m when $v := \mathcal{B}(\mathcal{P}(m, k_e))$ with $m \in$ plaintexts;*
2. *v' encrypts$_q$ m when $v' := \mathcal{B}(v, k_e)$ and v encrypts$_q$ m.*
3. *v encrypts$_{q'}$ $f(m_1, \ldots, m_n)$ when $f \mapsto f_K : q \to q' \in \mathcal{F}$, $v := f_K(v_1, \ldots, v_n)$, and each v_i encrypts$_q$ m_i for $i \in 1..n$.*

Homomorphic properties are usually incompatible with CCA2 security. Also, practical encryption schemes usually support a few fixed functions f, and often put limits on the number of consecutive applications of f_K. (This is the purpose of indexes in our model.) Intuitively, these homomorphic application are not perfect; they also produce noise, which may eventually leads to incorrect decryptions. For example, [4] provides a scheme with an unlimited number of additions but only one homomorphic multiplication. In our model, this translate to: $\mathcal{F} = \{+ \mapsto +_K : 0 \to 0; * \mapsto *_K : 0 \to 1; + \mapsto +_K : 1 \to 1\}$.

Figure 4 includes our additional rule for homomorphic computation, HOM-FUN. This rule is parametric on the homomorphic encryption scheme, and relies on a typing assumption on the corresponding 'virtual' computation on plaintexts, using a security policy extended with fresh plaintext variables: $\Gamma, x : \tau \sqcup \ell_y, \vec{x} : \vec{\tau} \sqcup \ell_y \vdash x := f(\vec{x}) : L(\tau) \sqcup \ell_y$.

The first hypotheses bind types to the variables k_e, y, and $\vec{z}$ involved in the homomorphic operation; these types are related by $\vec{\tau}, \tau \in \mathsf{E}$ and K, which describe the typing assumptions for encryptions with key k_e. They also ensure that the function f is homomorphically implementable in this encryption scheme with the corresponding indexes. The last hypothesis ensure that f would be typable if applied on the decrypted ciphertext. The label ℓ_y in the typing of k_e, y, and $\vec{z}$ records the flow from k_e and $\vec{z}$ to y.

Our typing rule ensures that if an homomorphic function types, its equivalent non-homomorphic version would also type, as illustrated in Example 7.

$$\textsc{Pre-Encrypt}\quad \frac{\Gamma(y) = \mathsf{Enc}\,\tau K q(\ell_y) \quad \vdash e : \tau \quad \tau \leq_C y \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \tau \in \mathsf{E}}{\vdash y := \mathcal{P}(e, k_e)\ :\ell_y}$$

$$\textsc{Blind}\quad \frac{(\mathcal{G}_e, \mathcal{B}, \mathcal{D}) \text{ is CPA} \quad \Gamma(y) = \mathsf{Enc}\,\tau\, K q(\ell_y) \quad \tau' \leq \tau \quad \vdash z : \mathsf{Enc}\,\tau'\, K q(L(\tau)) \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \tau, \tau' \in \mathsf{E} \quad k_e \leq_I \alpha \ \textbf{or}\ \tau \leq_C \alpha}{\vdash y := \mathcal{B}(z, k_e)\ :\ell_y}$$

$$\textsc{Hom-Fun}\quad \frac{\mathcal{F}_K(f) = f_K : \vec{q} \to q' \quad \Gamma(y) = \mathsf{Enc}\,\tau\, K q'(\ell_y) \quad \vdash z_i : \mathsf{Enc}\,\tau_i\, K q_i(\ell_y) \quad \text{for } z_i \in \vec{z} \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \vec{\tau}, \tau \in \mathsf{E} \quad \Gamma, x : \tau \sqcup \ell_y, \vec{x} : \vec{\tau} \sqcup \ell_y \vdash x := f(\vec{x}) : L(\tau) \sqcup \ell_y}{\vdash y := f_K(\vec{z}, k_e)\ :\ell_y}$$

Figure 4: Additional typing rules for pre-encryption, blinding, and encryption homomorphisms with policy Γ.

Examples The examples below use Paillier encryption, which enables us to add plaintexts by multiplying their ciphertexts. Thus, we use $f = +$ and $f_K = *$, and for $\mathcal{F}(+) = * : 0 \mapsto 0$ we obtain a single instance of rule HOM-FUN:

$$\textsc{Hom-Paillier}\quad \frac{\Gamma(y) = \mathsf{Enc}\,(\mathsf{Data}\,(\ell))\,K\,(\ell_y) \quad \vdash z_i : \mathsf{Enc}\,(\mathsf{Data}\,(\ell_i))\,K\,(\ell_y) \text{ for } i = 0, 1 \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \mathsf{Data}\,(\ell), \mathsf{Data}\,(\ell_0), \mathsf{Data}\,(\ell_1) \in \mathsf{E} \quad \ell_0 \sqcup \ell_1 \sqcup \ell_y \leq \ell \sqcup \ell_y}{\vdash y := z_0 * z_1\ :\ell_y}$$

EXAMPLE 7. *To illustrate our typing rule, we compare two programs that perform the same addition, on plaintexts (on the left) and homomorphically (on the right):*

$$\begin{array}{lll} P \doteq & k_e, k_d := \mathcal{G}_e(); & P' \doteq \quad k_e, k_d := \mathcal{G}_e(); \\ & z_1 := \mathcal{E}(x_1, k_e); & \quad z_1 := \mathcal{E}(x_1, k_e); \\ & x_1' := \mathcal{D}(z_1, k_d) & \\ & x := x_1' + x_1'; & \quad y := z_1 * z_1; \\ & y := \mathcal{E}(x, k_e); & \\ & r := \mathcal{D}(y, k_d); & \quad r := \mathcal{D}(y, k_d); \end{array}$$

Suppose that we type P with a policy Γ, then, relying on the plaintext-typing assumption in rule HOM-FUN*, we can also type P' with the same policy.*

EXAMPLE 8. *We show how to homomorphically multiply an encrypted value by a small integer factor.*

$$\begin{array}{l} k_e, k_d := \mathcal{G}_e(); y_{LH} := \mathcal{E}(x_{HH}, k_e); z_{SH} := \mathcal{E}(0, k_e); \\ \mathsf{for}\ i_{SH} := 1\ \mathsf{to}\ n_{SH}\ \mathsf{do}\ z_{SH} := z_{SH} * y_{LH}; \\ z_{LH} := \mathcal{B}(z_{SH}, k_e); \_; x_{HH} := \mathcal{D}(z_{LH}, k_d) \end{array}$$

By typing, we have that an honest-but-curious adversary ($\alpha = SH$) does not learn x_{HH}, and that a network adversary ($\alpha = LH$) learns neither x_{HH} nor n_{SH}. The latter guarantee crucially relies on blinding the result at the end of the loop, since otherwise the adversary may also iterate multiplications of y_{LH} and compare them to the result to guess n_{SH}.

Another classic example of homomorphic scheme is the ElGamal encryption, which enables us to multiply plaintexts by multiplying their ciphertexts; we omit similar, typable programming examples.

6. COMPUTATIONAL SOUNDNESS

We define security properties for probabilistic command contexts as computational variants of noninterference, expressed as games [9].

DEFINITION 7 (COMPUTATIONAL NON-INTERFERENCE). *Let Γ be a policy, α an adversary level, and P a command context.*

P is computationally non-interferent against α-adversaries when, for both $V = V_\alpha^C$ and $V = V_\alpha^I$, and for all polynomial commands J, B_0, B_1, T containing no cryptographic variables, such that $wv(B_b) \cap V = \emptyset$ and $rv(T) \subseteq V$, $\vec{A}$ α-adversaries such that $P[\vec{A}]$ is a polynomial command, $g \notin v(J, B_0, B_1, P, \vec{A})$, and some variable $b \notin v(J, B_0, B_1, P, \vec{A}, T)$ in the command

$$\mathit{CNI} \doteq b := \{0, 1\}; J; \mathsf{if}\ b = 0\ \mathsf{then}\ B_0\ \mathsf{else}\ B_1; P[\vec{A}]$$

if we have $\Pr[\mathit{CNI}; \bigwedge_{x \in rv(T)} x \neq \bot] = 1$, then the advantage $|\Pr[\mathit{CNI}; T; b =_0 g] - \frac{1}{2}|$ is negligible.

In the game, the command J; if $b = 0$ then B_0 else B_1 probabilistically initializes the memory, depending on b. Definition 7, then runs the command context applied to adversaries $P[\vec{A}]$. Finally, T attempts to guess the value of b and set g accordingly. T cannot read cryptographic variables, which are not an observable outcome of the program. The condition $\bigwedge_{x \in rv(T)} x \neq \bot$ rules out commands T that may read uninitialized memory. Hence, the property states that the two memory distributions for $b = 0$ and $b = 1$ after running *CNI* cannot be separated by an adversary that reads V.

Besides the cryptographic assumptions, we state additional safety conditions for soundness.

DEFINITION 8. *Let $\alpha \in \mathcal{L}$ be security label. A command context P is safe against α-adversaries when $\Gamma \vdash P$ and*

1. *Each key label is used in at most one key generation.*
2. *Each key variable read in P is first initialized by P.*
3. *There is no dependency cycle between key labels.*

We rely on these conditions to apply cryptographic games in the proof of type soundness, for instance to guarantee the integrity of decrypted values. They can be enforced by static analysis, for instance by collecting all relevant static occurrences of variables and forbidding encryption-key generation within loops.

Condition 1 prevents decryption-key mismatches. Condition 2 recalls our assumption on uninitialized variables for keys. Condition 3 prevents leaks with low integrity ciphertext when using CPA.

We also assume that each static key label is associated to a fixed scheme for encryption that meets Definitions 1, 5, or 6 and we impose constraints on the length of ciphertexts in order to prevent information leakage via the length of encrypted messages (see appendix for a formal definition).

Relying on these conditions, we obtain our main security theorem: well-typed programs are computationally non-interferent.

THEOREM 2. *Let $\alpha \in \mathcal{L}$ be a security label. Let Γ be a policy. Let P be a polynomial-time command context, safe against α-adversaries. P satisfies computational non-interference against α-adversaries.*

The proof relies on a series of typability-preserving program transformations that match the structure of the games used in the

cryptographic security assumptions (Definition 2). These transformations replace, one static key label at a time, couples of encryptions and decryption algorithms by an ideal implementation that maintains a global table for all values encrypted so far and encrypts 0s instead of the actual plaintexts.

7. PRIVATE SEARCH ON DATA STREAMS

We illustrate the use of Paillier encryption on a simplified version of a practical protocol developed by Ostrovsky and Skeith III [21] for privately searching for keywords in data streams (without the Bloom filter). The protocol has two roles: an agency P and a service S. Assume that the service issues, or processes, confidential documents such as mail orders or airline tickets, and that the agency wishes to retrieve any such document whose content matches some keywords on a secret black list. The two roles communicate using a public network. The black list is too sensitive to be given to the service. Conversely, the service may be processing a large number of documents, possibly at many different sites, and may be unwilling to pass all those documents to the agency.

We formally assume that the agency is more trusted than the service. We suppose that the documents are arrays of words, and that all words (including the keywords on the black list) appear in a public, trusted dictionary. We model the public network using variables shared between P and S. We rely on the additive property of Paillier encryption, detailed in Section 5. We code the protocol using three commands, explained below.

Initially, the agency generates a keypair and encodes the list of keywords as an array ($mask$) of encryptions indexed by the public dictionary that contains, for each word, either an encryption of 1 if the word appears in the black list or an encryption of 0 otherwise. The agency can distribute this array to the service without revealing the black list.

$$\begin{array}{l} P_0 \doteq\ k_e, k_d := \mathcal{G}_e(); \\ \quad \mathsf{for}\ i := 0\ \mathsf{to}\ size(words) - 1\ \mathsf{do} \\ \quad \mathsf{if}\ words[i] \in keywords\ \mathsf{then}\ wb := 1\ \mathsf{else}\ wb := 0; \\ \quad mask[i] := \mathcal{E}(wb, k_e) \end{array}$$

Then, for each document d, the service homomorphically computes the number of matching keywords, as the sum of the 0 and 1s encrypted in $mask$ for all indexes of the words ($d[j]$) present in the document, by multiplying those encryptions in e. Moreover, the service homomorphically multiplies e and the value of the document d (seen as a large integer) in z—the loop computes e^d, and may be efficiently replaced with a fast exponentiation. In particular, if they were no matches, z is just an encryption of 0. Finally, the serviceblinds both e and z before sending them to the agency—this step is necessary to declassify these encryptions without leaking information on the document.

$$\begin{array}{l} S \doteq\ e := 1; \\ \quad \mathsf{for}\ j := 0\ \mathsf{to}\ size(d) - 1\ \mathsf{do} \\ \quad\quad e := e * mask[words^{-1}(d[j])]; \\ \quad e' := \mathcal{B}(e, k_e); z := 1; \\ \quad \mathsf{for}\ j := 0\ \mathsf{to}\ d - 1\ \mathsf{do}\ z := z * e; \\ \quad z' := \mathcal{B}(z, k_e) \end{array}$$

Last, for each pair e' and z', the agency decrypts the value containing the number of matches (in n) and, if n is different from 0, retrieves the value of the document by decrypting the product and dividing by n.

$$\begin{array}{l} P \doteq\ n := \mathcal{D}(e', k_d); \\ \quad \mathsf{if}\ n > 0\ \mathsf{then} \\ \quad\quad nd := \mathcal{D}(z', k_d); log := log + (nd/n) \end{array}$$

Adding an outer loop for the documents processed by the service, the whole protocol is modelled as the command

$$\begin{array}{l} Q \doteq\ P_0; \\ \quad \mathsf{for}\ l := 0\ \mathsf{to}\ nb\_docs\ \mathsf{do} \\ \quad\quad d := docs[l]; S; P; \_ \end{array}$$

As discussed by Ostrovsky and Skeith III, this algorithm is reasonably efficient for the service, as it requires just a multiplication and a table lookup for each word in each document. Relying on Bloom filters, the authors of the protocol and Danezis and Diaz [6] develop more advanced variants, requiring less bandwidth and guaranteeing additional privacy properties for the service, but the overall structure of the protocol remains unchanged.

We now specify the security of the protocol as a policy Γ that maps the variable of Q to labels. We distinguish three levels of integrity: H for the agency, S for the service, and L for untrusted data, with $H <_I S <_I L$. k_e and $keywords$ have high integrity (H); all other variables have service integrity (S).

We also distinguish four levels of confidentiality: H and P for the agency, S for the service, and L for public data. With $L <_C P <_C H$ and $L <_C S <_C H$.

- $words$, $mask$, e', z', i, nb_docs, k_e are public (at level L);
- $docs$, d, e, z, j, l are readable by the service (S) and the agency;
- $keywords$, kd, bw are readable only by the agency (P);
- log, n, nd are also readable by the agency, but they depend on low-integrity decryptions, and thus should not flow to any further encryption with key k_e (H).

We prove the security properties of the protocol as instances of computational non-interference, established by typing our code with different choices of adversary level α.

THEOREM 3. *$\Gamma \vdash Q$ when α ranges over SH, SS, and LS, hence Q is computationally non interferent against these adversaries.*

The case $\alpha =$ *SH* corresponds to a powerful adversary that entirely controls the service but still learns nothing about the blacklist: it cannot distinguish between any two runs of the protocol with different values of $keywords$. If the service is honest but curious ($\alpha =$ *SS*), the protocol also completes with the correct result. If the adversary controls only the network ($\alpha =$ *LS*), it learns nothing either about the documents contents ($docs$).

8. BOOTSTRAPPING HOMOMORPHIC ENCRYPTIONS

Gentry [11] proposes a first fully homomorphic encryption (FHE) scheme, that is, a scheme that supports arbitrary computations on encrypted data, thereby solving a long-standing cryptographic problem [13]. Others, e.g. Smart and Vercauteren [23], develop more efficient constructions towards practical schemes.

These constructions are based on *bootstrappable encryption schemes*, equipped with homomorphic functions for the scheme's own decryption function as well as some more basic functions; these basic operations may represent e.g. logical gates. (Designing a bootstrappable encryption scheme is in itself very challenging, as it severely constrains the decryption algorithm, but is not our concern here.)

The bootstrap relies on a series of keys. Whenever the noise accumulated as the result of homomorphic evaluation needs to be

canceled, the intermediate result (encrypted under the current key) is encrypted using the next encryption key, then homomorphically decrypted using an encryption of the current decryption key under the next encryption key. This bootstrapping yields an homomorphic scheme for any function as long as the computation can progress using other homomorphic functions between encryptions and decryptions to the next key. (Following Gentry's terminology, the resulting scheme is a *leveled* FHE: the key length still depends on the max size of the circuit for evaluating f; some additional key-cycle assumption is required to get a fixed-length key.)

Next, we assume the properties of the base algorithms given by Gentry—being CPA and homomorphic for its own decryption plus basic operations f_i that suffice to evaluate some arbitrary function f. We then show that these properties suffice to program and verify by typing the bootstrapping part of Gentry's construction, leading to homomorphic encryption for this arbitrary function f. Typing of Gentry's bootstrapping relies on typing rules for CPA with multiple keys and types of encrypted values, and on instances of rule HOM-FUN with $f = \mathcal{D}$.

We assume given a bootstrappable scheme using a set of homomorphic functions for decrypting and for computing: $\mathcal{F} = \{\mathcal{D} \mapsto \mathcal{D}_K : 0 \to 1; f^i \mapsto f^i_K : 1 \to 2 \text{ for } i = 1..n\}$.

We are now ready to program the bootstrapping for f. Both parties agree on the function to compute, f, and its decomposition into successive computation steps $(f_i)_{i=1..n}$ such that $f = f_1; \ldots; f_n$. This may be realized by producing a circuit to evaluate f and then letting the f_i be the successive gates of the circuit. For simplicity, we assume that each f_i operates on all the bits of the encrypted values; Gentry instead uses a tuple of functions $f_i^{j=1..w}$ for each step, each f_i^j computing one bit of the encrypted values. This more detailed bootstrapping can be typed similarly.

For the plaintexts, we write x_0 for the input, $x_1, \ldots x_{n-1}$ for the intermediate results, and x_n for the final result, so the high-level computation is just $x_i := f_i(x_{i-1})$ for $i = 1..n$.

The client generates all keypairs; encrypts the input using the first encryption key; encrypts each decryption key using the next encryption key; calls the server; and decrypts the result using the final key:

$$\begin{array}{lll} P_c \doteq ke_i, kd_i := \mathcal{G}_e(); & & \text{for } i = 1..n \\ \quad skd_i & := \mathcal{E}(kd_i, ke_{i+1}); & \text{for } i = 1..n-1 \\ \quad y_0 & := \mathcal{E}(x_0, ke_1); & \\ \quad \_; & & \text{calling the server} \\ \quad x_n & := \mathcal{D}(z_n, sk_n) & \end{array}$$

The message consists of the content of the shared variables y_0, ke_1, skd_1, …, ke_{n-1}, skd_{n-1}, ke_n. The server performs the homomorphic computation for each f_i, interleaved with the re-keying.

$$\begin{array}{l} P_s \doteq \big(z_i := f^i_{K_i}(y_{i-1}, ke_i); \\ \qquad t_i := \mathcal{E}(z_i, ke_{i+1}); \\ \qquad y_i := \mathcal{D}_{K_{i+1}}(t_i, skd_i, ke_{i+1}); \big)_{\text{ for } i=1..n-1} \\ \qquad z_n := f^n_{K_n}(y_{n-1}, ke_n); \_ \end{array}$$

We specify the security of the protocol as a policy Γ that maps the variable of P_c and P_s to labels. We distinguish three levels of integrity: H for the agency, S for the service, and L for untrusted data, with $H <_I S <_I L$. ke_i and kd_i have high integrity (H) for $i = 1..n$; all other variables have service integrity (S).

Since our scheme is only CPA, we need $n+1$ levels of confidentiality to separate the results of low integrity decryptions: $H_0..H_n$ for the agency and L for public data, such that $L <_C H_0 <_C \cdots <_C H_n$.

- $ke_i, skd_i, z_i, t_i, y_i$ are public for $i = 1..n$ (at level L);
- x_0 is readable only by the agency (H_0);
- kd_i is readable only by the agency for $i = 1..n$ (H_i);
- x_n is also readable by the agency, but moreover it depends on low-integrity decryptions, and thus should not flow to any further encryption with any of the key ke_i for $i = 1..n$ (H_n).

Let $\tau_i = \mathsf{Data}\,(H_iS)$. We type our variables as follows, for $i = 1..n-1$.

$$\begin{array}{ll} \Gamma(ke_i) & = \mathsf{Ke}\,\mathsf{E}_i\; K_i(LH) \\ \Gamma(kd_i) & = \mathsf{Kd}\,\mathsf{E}_i\; K_i(H_iH) \\ \Gamma(x_0) & = \tau_0 \\ \Gamma(skd_i) & = \mathsf{Enc}\,(\mathsf{Kd}\,\mathsf{E}_i\; K_i(H_{i-1}S))K_{i+1}0(LS) \\ \Gamma(y_0) & = \mathsf{Enc}\,\tau_0\; K_1 0(LS) \\ \Gamma(z_i) & = \mathsf{Enc}\,\tau_{i-1}\; K_i 2(LS) \\ \Gamma(t_i) & = \mathsf{Enc}\,(\mathsf{Enc}\,\tau_{i-1}\; K_i 2(LS))\; K_{i+1}0(LS) \\ \Gamma(y_i) & = \mathsf{Enc}\,\tau_i\; K_{i+1}1(LS) \\ \Gamma(z_n) & = \mathsf{Enc}\,\tau_{n-1}\; K_n 2(LS) \\ \Gamma(x_n) & = \tau_n \end{array}$$

with the encrypted sets below, for $i = 2..n$:

$$\begin{array}{l} \mathsf{E}_1 = \{\tau_0\} \\ \mathsf{E}_i = \{\tau_{i-1}, \mathsf{Kd}\,\mathsf{E}_{i-1}\; K_{i-1}(H_{i-2}S), \mathsf{Enc}\,\tau_{i-1}\; K_{i-1}(HS)\} \end{array}$$

We verify that the commands above are typable with the resulting policy Γ:

THEOREM 4. *$\Gamma \vdash P_c[P_s]$ when α ranges over LS and LH, hence $P_c[P_s]$ is computationally non-interferent against these adversaries.*

The theorem yields strong indistinguishability guarantees. The case $\alpha = LS$ corresponds to an "honest but curious" adversary, who can observe all server operations on ciphertexts (that is, read all intermediate values) but still learn nothing about the plaintexts. The case $\alpha = LH$ corresponds to an active adversary in full control of the service, who can disrupt its operations, and still learns nothing about the plaintexts. To clarify these confidentiality guarantees, we verify by typing that our construction is in particular CPA. We let $\mathcal{S}$ abbreviate the scheme $(\mathcal{G}^h_e, \mathcal{E}^h, \mathcal{D}^h, \{f \mapsto f_K : 0 \to 1\})$ defined as follows:

$$\begin{array}{ll} ke, kd := \mathcal{G}^h_e() \doteq & \big(ke_i, kd_i := \mathcal{G}_e();\big)_{\text{ for } i=1..n} \\ & \big(skd_i := \mathcal{E}(kd_i, ke_{i+1});\big)_{\text{ for } i=1..n-1} \\ & ke := (ke_1, ..., ke_n, skd_1, skd_{n-1}); \\ & kd := (kd_1, kd_n) \\ y := \mathcal{E}^h(e, ke) \doteq & y := \mathcal{E}(e, (ke)_1); \\ y := f^h_K(z, ke) \doteq & y_0 := z; \\ & \big(z_i := f^i_{K_i}(y_{i-1}, (ke)_i); \\ & \quad t_i := \mathcal{E}(z_i, (ke)_{i+1}); \\ & \quad y_i := \mathcal{D}_{K_{i+1}}(t_i, (ke)_{i+n}, (ke)_{i+1});\big)_{\text{ for } i=1..n-1} \\ & y := f^n_{K_n}(y_{n-1}, (ke)_n) \\ x := \mathcal{D}^h(y, kd) \doteq & \mathsf{if}\ is\_enc(y, (kd)_1) \\ & \mathsf{then}\ x := \mathcal{D}(y, (kd)_1) \\ & \mathsf{else}\ x := \mathcal{D}(y, (kd)_2) \end{array}$$

THEOREM 5. *$\mathcal{S}$ is a CPA homomorphic encryption scheme.*

9. RELATED WORK

Information flow policies and types Non-interference was introduced by Goguen and Meseguer [12]. Laud [17] introduces computational non-interference in a model with passive adversaries (CNIP). Laud and Vene [19] propose a type system (relying on CPA) to verify CNIP in a language with symmetric encryption but where decrypted values cannot be treated as keys. Fournet and Rezk [9] generalize CNIP to the active case, with adversaries that may interfere with the normal execution of programs, and also covers integrity properties. We present a type system (relying on CCA2) that enables typing of key establishment protocols. In a similar line of work, Smith and Alpízar [24] present a type system (relying on CCA2) with encryption and decryption but no explicit keys. Askarov et al. [1] propose a type system for cryptographically masked flows. Their system also handles key reuse for plaintexts at different levels, via subtyping, but their adversary model is weaker than ours, and excludes in particular chosen-ciphertext attacks. Laud [18] investigates conditions such that cryptographically masked flow imply secrecy and proposes a simpler computationally-sound execution model for passive adversaries.

Homomorphic encryptions The first CPA homomorphic encryption scheme is due to Goldwasser and Micali [13]. Additive homomorphic encryption schemes proved to be CPA include Benaloh [2] and Paillier [22]. ElGamal [8] is a CPA multiplicative homomorphic encryption scheme. Other schemes [4, 11] allow both addition and multiplication.

Protocol analysis featuring homomorphic encryptions Delaune et al. [7] and Lafourcade [16] analyse protocols featuring homomorphic encryptions using equational theories (see Cortier et al. [5]) to model homomorphisms.

Acknowledgments We thank Martín Abadi, Anupam Datta, Markulf Kohlweiss, Nikhil Swamy, and the anonymous reviewers for their comments.

References

[1] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. In *Proceedings of the 13th International Static Analysis Symposium*, LNCS, 2006. Springer-Verlag.

[2] J. C. Benaloh. Secret sharing homomorphisms: Keeping shares of a secret sharing. In A. M. Odlyzko, editor, *CRYPTO*, volume 263 of *LNCS*, pages 251–260. Springer, 1986.

[3] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT*, pages 127–144, 1998.

[4] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Theory of Cryptography (TCC)*, number 3378 in LNCS, pages 325–341. Springer, Feb. 2005.

[5] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[6] G. Danezis and C. Diaz. Space-efficient private search with applications to rateless codes. In *Financial cryptography and data security: 11th international conference, FC 2007, and 1st International Workshop on Usable Security, USEC 2007*, 2007.

[7] S. Delaune, P. Lafourcade, D. Lugiez, and R. Treinen. Symbolic protocol analysis in presence of a homomorphism operator and exclusive or. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *ICALP (2)*, volume 4052 of *LNCS*, pages 132–143. Springer, 2006.

[8] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18, 1984.

[9] C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335, Jan. 2008.

[10] C. Fournet, G. le Guernic, and T. Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *ACM Conference on Computer and Communications Security*, pages 432–441, Nov. 2009.

[11] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM symposium on Theory of computing (STOC)*, pages 169–178, 2009.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[13] S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *ACM symposium on Theory of computing (STOC)*, pages 365–377, 1982.

[14] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *17th ACM Conference on Computer and Communications Security*, pages 451–462, Oct. 2010.

[15] J. Katz and L. Malka. Secure text processing with applications to private DNA matching. In *17th ACM Conference on Computer and Communications Security*, pages 485–492, Oct. 2010.

[16] P. Lafourcade. *Vérification des protocoles cryptographiques en présence de théories équationnelles*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, Sept. 2006.

[17] P. Laud. Semantics and program analysis of computationally secure information flow. In *10th European Symposium on Programming (ESOP)*, volume 2028 of *LNCS*. Springer, Apr. 2001.

[18] P. Laud. On the computational soundness of cryptographically-masked flows. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 337–348, Jan. 2008.

[19] P. Laud and V. Vene. A type system for computationally secure information flow. In *Fundamentals of Computation Theory*, LNCS, pages 365–377. Springer, 2005.

[20] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[21] R. Ostrovsky and W. E. Skeith III. Private searching on streaming data. In V. Shoup, editor, *Advances in Cryptology–CRYPTO 2005*, volume 3621 of *LNCS*, pages 223–240, 2005.

[22] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[23] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. *Public Key Cryptography–PKC 2010*, pages 420–443, 2010.

[24] G. Smith and R. Alpízar. Secure information flow with random assignment and encryption. In *FMSE '06: fourth ACM workshop on Formal methods in security*, pages 33–44, 2006.