Gradual Release: Unifying Declassification, Encryption and Key Release Policies

Aslan Askarov Andrei Sabelfeld
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract

Information security has a challenge to address: enabling information-flow controls with expressive information release (or declassification) policies. Existing approaches tend to address some aspects of information release, exposing the other aspects for possible attacks. It is striking that these approaches fall into two mostly separate categories: revelation-based (as in information purchase, aggregate computation, moves in a game, etc.) and encryption-based declassification (as in sending encrypted secrets over an untrusted network, storing passwords, etc.).

This paper introduces gradual release*, a policy that unifies declassification, encryption, and key release policies. We model an attacker's knowledge by the sets of possible secret inputs as functions of publicly observable outputs. The essence of gradual release is that this knowledge must remain constant between releases. Gradual release turns out to be a powerful foundation for release policies, which we demonstrate by formally connecting revelation-based and encryption-based declassification. Furthermore, we show that gradual release can be provably enforced by security types and effects.*

1 Introduction

Information security [32] has a challenge to address: enabling information flow controls with expressive information release (or declassification) policies [32, 42, 34]. In a scenario of systems that operate on data with different sensitivity levels, the goal is to provide security assurance via restricting the information flow within the system. However, allowing no flow whatsoever from secret (*high*) inputs to public (*low*) outputs (as prescribed by *noninterference* [16]) is too restrictive because many systems deliberately declassify information from high to low.

Characterizing and enforcing declassification policies is the focus of an active area of research [34]. However, existing approaches tend to address selected aspects of information release, exposing the other aspects for possible attacks. It is striking that these approaches fall into two mostly separate categories: revelation-based (as in information purchase, aggregate computation, moves in a game, etc.) and encryption-based declassification (as in sending encrypted secrets over an untrusted network, storing passwords, etc.). It is essential that declassification policies support a combination of these categories: for example, a possibility to release the result of encryption should not be abused to release cleartext through the same declassification mechanism.

This paper introduces *gradual release*, a policy that unifies declassification, encryption, and key release policies. As we explain below, the latter is not only a useful feature, but also a vital component for connecting revelation-based and encryption-based declassification. We model an attacker's knowledge by the sets of possible secret inputs as functions of publicly observable outputs. The essence of gradual release is that this knowledge must remain constant between releases. Gradual release turns out to be a powerful foundation for release policies, which we demonstrate by formally connecting revelation-based and encryption-based declassification.

When it comes to handling encryption, there is a demand for expressing rich policies beyond declassification at the point of encryption. To this end, a desirable ingredient in declassification policies is reasoning about released keys. In bit commitment, premature revelation of the bit should be prevented by not releasing the secret key until necessary. In a media distribution scenario—when large media is distributed in encrypted form, and the key is supplied on the date of media release—early key release should be prevented. In addition, key release policies are important for *mental poker* [35, 9, 4] (for playing poker without a trusted third party), where the participants reveal their keys for each other at the end of the game, in order to prove that they were not cheating during the game. In this protocol too, it should not be possible to release secret keys prematurely or encrypt with a key that has already been released.

Gradual release allows for reasoning about newly generated and released keys. In fact, this combination turns out to

In *Proc. IEEE Symposium on Security and Privacy*, Berkeley/Oakland, CA, May 2007.

be crucial for connecting revelation-based and encryption-based declassification. We show that gradual release for revelation-based declassification can be represented by a rewardingly simple encryption-based declassification: declassifying an expression corresponds to encrypting the expression with a fresh key and immediately releasing the key.

As a result, gradual release is, to the best of our knowledge, the first framework to unify revelation-based and encryption-based declassification policies. Furthermore, we show that gradual release can be provably enforced by security types and effects.

Structure-wise, Section 2 introduces gradual release, illustrates its properties, and shows how to enforce it by a security type system for a simple declassification-enabled language. Section 3 enriches the language with key generation, encryption, and key release primitives. Section 4 applies gradual release to the enriched language. Section 5 presents a type and effect system that enforces gradual release for the enriched language. Section 6 provides useful examples of typed programs. Section 7 discusses related work, and Section 8 concludes.

2 Gradual release

Gradual release is a general notion, defined in terms of events (which are classified into low and high, with some of the low ones classified as release events). Of particular interest are language-based instantiations of this model, where events are generated by program constructs. A direct benefit of such instantiations is the possibility of enforcing gradual release by static analysis. To be concrete (but without loss of generality), we present gradual release for a simple imperative language with declassification. We show that gradual release is a conservative extension of noninterference and demonstrate how to provably enforce gradual release by a security type system.

Language Figure 1 presents the syntax of the language, which contains expressions e and commands c. For simplicity, we assume that variables are assigned one of the two security levels: `L` (low) or `H` (high), forming a simple *security lattice*, where $\texttt{L} \sqsubset \texttt{H}$. These levels are recorded in the *security environment* Γ, a mapping from variable names to security levels. The construct $\texttt{declassify}(e)$ in an assignment is provided for declassifying the level of e to `L`.

$$
\begin{array}{lcl}
e & ::= & n \mid x \mid e \; op \; e \\
c & ::= & \texttt{skip} \mid x := e \mid c; c \mid \texttt{if } e \texttt{ then } c \texttt{ else } c \\
 & \mid & \texttt{while } e \texttt{ do } c \mid x := \texttt{declassify}(e)
\end{array}
$$

Figure 1. Simple imperative language

Low-equivalence Memories are mappings of variable names to values. Two memories M_1, M_2 are *low-equivalent* with respect to a security environment Γ if $\forall x \,.\, \Gamma(x) \sqsubseteq \texttt{L} \implies M_1(x) = M_2(x)$. This is denoted as $M_1 \sim_\Gamma M_2$.

Semantics and low events As is standard, expressions evaluate according to rules of the form $\langle M, e\rangle \Downarrow n$, where $\langle M, e\rangle$ is a configuration consisting of the memory M and the expression to evaluate e, and n is the resulting value.

Semantics for commands are expressed by small-step transitions between configurations. These transitions have either the form $\langle M, c\rangle \xrightarrow{\alpha} \langle M', c'\rangle$, which corresponds to a step from configuration $\langle M, c\rangle$ to configuration $\langle M', c'\rangle$, or the form $\langle M, c\rangle \xrightarrow{\downarrow} M'$, which corresponds to termination in the memory M'.

A transition is *low* if it is due to an assignment to a low variable or termination. The *event* $\alpha \in \{\epsilon, \ell\}$ records whether the transition is low (reflected by the label ℓ, where ℓ is either the projection $M'_{\texttt{L}}$ of the low part of M' or a *termination event* $\downarrow$) or otherwise (reflected by the empty label ϵ). We write $\langle M, c\rangle \xrightarrow{\vec{\ell}}{}^{*} \langle M', c'\rangle$ (resp. $\langle M, c\rangle \xrightarrow{\vec{\ell}}{}^{*} M'$) when $\langle M', c'\rangle$ (resp. M') is reachable from $\langle M, c\rangle$ by a sequence of small steps, where $\vec{\ell}$ represents the sequence of generated low events. We classify the termination event $\downarrow$ as low. If an event in a sequence is a termination event, then no other events may be generated after it.

A particular kind of low events are due to `declassify` commands. We refer to those events as *release* events. Release events record the points of intentional information release by a command. The complete semantics of the language is available in an accompanying technical report [5].

Knowledge We let the attacker observe the low projection $M^0_{\texttt{L}}$ of the initial memory M^0, and all intermediate low events $\ell_1, \ldots, \ell_n$ generated during a run of a command c. Given these observations, the attacker may infer the set of initial memories that could possibly have led to these events. We refer to this set as the attacker's *knowledge* about the initial memory.

Semantically, the knowledge is a set of tuples, where each tuple represents a possible initial memory. For example, consider the program $l := h_1 + h_2$ and a sample run that yields a low event ℓ where $\ell(l) = 10$. The attacker's knowledge in this case is all such memories that the sum of h_1 and h_2 is 10:

$$
\begin{array}{cccc}
 & h_1 & h_2 & \ldots \\
\{ & (1, & 9, & \ldots) \\
 & (2, & 8, & \ldots) \\
 & (3, & 7, & \ldots) \\
 & (4, & 6, & \ldots) \\
 & \ldots & & \}
\end{array}
$$

Note that in our terminology knowledge corresponds to *uncertainty* about the tuples in the knowledge set: any of the tuples is a possible input. The actual knowledge of the attacker is that tuples outside the knowledge set are *not* possible inputs.

As the computation progresses, the uncertainty might decrease because new observations might render some initial inputs impossible. This means that the knowledge set may shrink with time.

Let $L(c, M_{\mathtt{L}}^0)$ be the set of possible low event sequences that a program c may generate along terminating traces that start in some memory whose low projection is $M_{\mathtt{L}}^0$:

$$L(c, M_{\mathtt{L}}^0) \triangleq \{\vec{\ell} \mid \exists M, M' \,.\, M_{\mathtt{L}} = M_{\mathtt{L}}^0 \wedge \langle M, c\rangle \xrightarrow{\vec{\ell}}{}^{*} M'\}$$

Based on a program c, a low projection of the initial memory $M_{\mathtt{L}}^0$, and a (possibly empty) sequence of low events $\vec{\ell} \in \widehat{L}(c, M_{\mathtt{L}}^0)$, where $\widehat{L}(c, M_{\mathtt{L}}^0)$ denotes the prefix closure of $L(c, M_{\mathtt{L}}^0)$, the *knowledge* is defined as:

$$k(c, M_{\mathtt{L}}^0, \vec{\ell}) \triangleq \{M \mid M_{\mathtt{L}} = M_{\mathtt{L}}^0 \wedge \exists M', c' \,. \\ \langle M, c\rangle \xrightarrow{\vec{\ell}}{}^{*} \langle M', c'\rangle \;\vee\; \langle M, c\rangle \xrightarrow{\vec{\ell}}{}^{*} M'\}$$

This set records all possible inputs that lead to observing $\vec{\ell}$ when starting with initial memories that agree with $M_{\mathtt{L}}^0$ on the low variables. This definition of knowledge is *termination-sensitive* (cf. [32]) because observing that program does not enter an infinite loop may lead to refining the knowledge.

Given a command c and the low projection of a memory $M_{\mathtt{L}}^0$, we also define the *initial knowledge* $k(c, M_{\mathtt{L}}^0)$ that corresponds to all possible initial memories that lead to termination:

$$k(c, M_{\mathtt{L}}^0) \triangleq \{M \mid M_{\mathtt{L}} = M_{\mathtt{L}}^0 \wedge \exists M', \vec{\ell} \,.\, \langle M, c\rangle \xrightarrow{\vec{\ell}}{}^{*} M'\}$$

Using this definition, we define a *termination-insensitive* (cf. [32]) version of knowledge as:

$$k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}) \triangleq k(c, M_{\mathtt{L}}^0, \vec{\ell}) \cap k(c, M_{\mathtt{L}}^0)$$

As we expect, the attacker may not "forget" the knowledge about the initial states, i.e., each new observable event may only refine the knowledge:

Proposition 1 (Monotonicity of knowledge)**.** *For a command c, some $M_{\mathtt{L}}^0$, and $\vec{\ell}_n \in L(c, M_{\mathtt{L}}^0)$, where $\vec{\ell}_n = \ell_1, \ldots, \ell_n$, we have*

$$\forall i \,.\, 1 \le i \le n \,.\, k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}_{i-1}) \supseteq k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}_i)$$

where $k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}_0) \triangleq k(c, M_{\mathtt{L}}^0)$. The proofs of this and the following results are available in an accompanying technical report [5].

Noninterference We present a definition of noninterference and demonstrate how to represent it in the knowledge-based setting.

Definition 1 (Noninterference)**.** *A command c satisfies* noninterference *if whenever $M_1 \sim_\Gamma M_2$, $\langle M_1, c\rangle \xrightarrow{\vec{\ell}_1}{}^{*} M_1'$, and $\langle M_2, c\rangle \xrightarrow{\vec{\ell}_2}{}^{*} M_2'$ then $\vec{\ell}_1 = \vec{\ell}_2$ (and $M_1' \sim_\Gamma M_2'$).*

The definition requires that when starting with low-equivalent memories, terminating traces must agree on their low events (and, as a consequence, on the low parts of the resulting memories). This corresponds to the absence of flows from high to low data.

We show that this flavor of noninterference is straightforwardly expressible in the knowledge-based setting:

Proposition 2. *A command c satisfies noninterference if and only if*

$$\forall M_{\mathtt{L}}^0, \vec{\ell} \in L(c, M_{\mathtt{L}}^0) \,.\, k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}) = k(c, M_{\mathtt{L}}^0)$$

The proposition states that the attacker's knowledge does not benefit from observing a run of a noninterfering program: all memories that agree with $M_{\mathtt{L}}^0$ on the low part are possible inputs regardless of the observed low events $\vec{\ell}$.

Gradual release With every new low event produced by a program run, the attacker's knowledge may become more precise, i.e., the set of possible initial memories may become smaller. The only intentional points when this knowledge may be narrowed down are the release events, as specified by declassification primitives. Gradual release accepts changes in the knowledge that are caused by the release events and requires that no other low events may affect the knowledge.

Definition 2 (Gradual release)**.** *A command c satisfies* gradual release *if for all $M_{\mathtt{L}}^0$ and $\vec{\ell} \in L(c, M_{\mathtt{L}}^0)$, where $\vec{\ell}_n = \ell_1, \ldots, \ell_n, n \ge 1$, of which $\ell_{r_1}, \ldots, \ell_{r_m}$ are all release events, we have:*

$$\forall i \,.\, 1 \le i \le n \,.\, (\forall j \,.\, r_j \ne i) \implies \\ k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}_{i-1}) = k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}_i)$$

where, as before, $k_{\downarrow}(c, M_{\mathtt{L}}^0, \vec{\ell}_0) \triangleq k(c, M_{\mathtt{L}}^0)$. The gradual release requirement on the evolution of knowledge is illustrated in Figure 2, where the vertical axis is uncertainty, and the horizontal axis is time.

In the examples throughout the paper, variables $l, l_1, \ldots$ and $h, h_1, \ldots$ are assumed to be low and high, respectively. Examples of programs that are rejected by gradual release are:

```
l := h
```

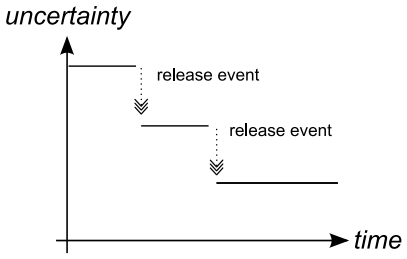

Figure 2. Gradual release

where the knowledge is narrowed down from all possible high inputs to exactly one, and:

```
if h then l := declassify(h1)
```

where the knowledge is narrowed down to memories where h is zero in the case when no low events are observable before termination.

Examples of programs that are accepted by gradual release are:

```
l := declassify(h)
```

where the release event due to declassification justifies the change in the knowledge, and:

```
l:=declassify(h != 0); if l then l1 := declassify(h1)
```

where the non-zero test expression is explicitly released before the choice whether to declassify h_1 is made.

In contrast to some declassification definitions (e.g., [10, 33], but in agreement with others (e.g., [29]), gradual release assumes that the attacker observes more information about traces (namely, effects of assignments to low variables) than visible initial/final values. Moreover, the attacker observes some events produced by partial program executions. This enables natural extensions to languages with input/output: low assignments can be viewed as implicit output on a low channel. Note that breaches in security due to events that are not happening are caught by gradual release thanks to termination events. Recall that in the second insecure example above, if the only observable event by the attacker is termination, then the attacker may deduce that the assignment in the conditional did *not* happen, and that the initial value of h must be zero.

In a recent classification of declassification [34], release policies are classified with respect to the *what*, *who*, *where*, and *when* dimensions of declassification. Under this classification, gradual release is primarily a *where* policy because it emphasizes that release is only allowed via declassification points. This allows us capturing some flows that are not represented by other dimensions. For example, the following program is accepted by pure *what* definitions such as *delimited release* [33] that ignore the *where* aspect:

```
h1 := h2;
h2 := 0;
l1 := declassify (h2);
h2 := h1;
l2 := h2
```

Note that at the point of declassification, the attacker learns nothing about the value of h_2. However, after reaching the assignment $l_2 := h_2$ the attacker's knowledge will allow inferring the value h_2. This is rejected by the release policy because the assignment $l_2 := h_2$ is not a declassification.

Compared to several *where* definitions, gradual release does not rely on state resetting in-between transitions. Consider, for example, the following program:

```
l := declassify (h);
l := h;
```

This innocent program is a false negative for the *intransitive downgrading* [27], *non-disclosure* [2], *flow locks* [8], and $WHERE$ [26] definitions. These definitions reject the program above because they demand security in the presence of state resetting. The program is rejected by them because resetting the secret state after declassification may reintroduce secret into h before it is assigned to l. However, the program is secure according to gradual release because the attacker does not gain any new knowledge from observing the effect of the last assignment.

As a sanity check for our definition, we show that for programs without release, gradual release is equivalent to noninterference. This property is known as the *conservativity* principle of declassification [34]. The principle follows from Proposition 2 and because the attacker's knowledge for declassification-free programs must remain constant over the execution of the program.

Proposition 3. *If a command c is free of declassification then c satisfies gradual release if and only if c satisfies noninterference.*

A final remark on the dimensions: although gradual release is primarily a *where* policy, it can be also viewed as a *relative what* policy. Indeed, *what* attacker learns remains constant with respect to the last release point. Further, it is possible to fully integrate the *what* dimension into gradual release. Assume a knowledge evolution sequence that is provided explicitly as a policy (for example, in terms of *escape hatch expressions* as in delimited release [33]). In addition to the demand on constant knowledge in-between releases, an enhanced policy might require that each refinement of knowledge is in strict accordance with what can be learned from observing the respective escape hatch expression from the sequence in the policy.

Enforcement It is straightforward to enforce gradual release by a security type system. Figure 3 presents the typing rules for expressions and commands. The rules for catching explicit and implicit flows are standard [12, 39]. The

$$\text{(T-INT)}\frac{}{\Gamma \vdash n : \mathtt{L}} \qquad \text{(T-VAR)}\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}$$

$$\text{(T-OP)}\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash e_1 \mathit{\ op\ } e_2 : \sigma_1 \sqcup \sigma_2}$$

$$\text{(T-SKIP)}\frac{}{\Gamma, pc \vdash \mathtt{skip}}$$

$$\text{(T-ASGN)}\frac{\Gamma \vdash e : \sigma \qquad pc \sqcup \sigma \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e}$$

$$\text{(T-SEQ)}\frac{\Gamma, pc \vdash c_1 \qquad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

$$\text{(T-IF)}\frac{\Gamma \vdash e : \sigma \qquad \Gamma, pc \sqcup \sigma \vdash c_i \qquad i = 1, 2}{\Gamma, pc \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2}$$

$$\text{(T-WHILE)}\frac{\Gamma \vdash e : \sigma \qquad \Gamma, pc \sqcup \sigma \vdash c}{\Gamma, pc \vdash \mathtt{while}\ e\ \mathtt{do}\ c}$$

$$\text{(T-DECL)}\frac{pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := \mathtt{declassify}(e)}$$

Figure 3. Type rules for the simple language

only non-standard rule is (T-DECL), which disallows declassification inside of conditionals and loops with sensitive guards. We call such a conditional or loop *high context* and track it with the context type variable pc. By requiring $pc \sqsubseteq \Gamma(x)$ at declassification points, we ensure that if pc is H then we might be inside a high context, and, hence, we may not assign the result of declassification to a variable at level L.

Note that no information about the declassified expression e is used in the rule for declassification (T-DECL). This is sensible because gradual release is a *where* policy, not a *what* policy, and so the value (and even the syntax) of the declassified expression is unimportant.

That the type system enforces gradual release is guaranteed by the following theorem:

Theorem 1. *If $\Gamma, pc \vdash c$ then c satisfies gradual release.*

3 Language with cryptographic primitives and key release

This section presents an enriched language that includes cryptographic primitives and key release features.

Syntax Figure 4 displays the syntax of the language, which is based on the one presented in [3]. Values and keys have corresponding *security levels*. Values are either low (L) or high (H). The key levels declare the maximum value security level the key can safely encrypt. In particular, a key at level *high key* (HK) may safely encrypt low and high values, whereas a key at level *low key* (LK) may only safely encrypt low values. Keys at level RK correspond to the previously secret keys that have been released and may only safely encrypt public values.

Basic types t consist of integers int and ciphertexts $\mathtt{enc}_\gamma\ \tau$ obtained by encrypting data of *primitive type* τ with keys at level γ. Primitive types τ consist of basic types labeled with security levels $t\ \sigma$, key types $\mathtt{key}\ \gamma$, and pairs (τ, τ) of primitive types.

Apart from expressions for encryption and decryption, expressions are standard: integers, variables, total binary operators, pair formation, and projection. Commands include the standard commands of an imperative language, a command for generating a new key at a given security level, and a command for releasing keys.

We assume that all variables x used in program text are typed with primitive types according to a *typing environment* Γ as $x : \Gamma(x)$. We also define the low projection $\Gamma_{\mathtt{L}}$ of the typing environment, which only includes low variables.

Semantics First we define values and environments, which are used in the following definitions of the semantics for expressions and commands. Let $n \in \mathbb{Z}$ range over integers and $k \in \mathit{Key} = \mathit{Key}_{\mathtt{LK}} \cup \mathit{Key}_{\mathtt{HK}}$ range over *keys*, where $\mathit{Key}_{\mathtt{LK}}$ and $\mathit{Key}_{\mathtt{HK}}$ are disjoint. We assume that released keys belong to the set of high keys $\mathit{Key}_{\mathtt{HK}}$. *Values* are built up by *ordinary values*: integers, keys, and pairs of values; together with *encrypted values* $u \in U = U_{\mathtt{LK}} \cup U_{\mathtt{HK}}$.

$$\text{values} \in \mathit{Value} \qquad v \quad ::= \quad n \mid k \mid (v, v) \mid u$$

The system is parameterized over two *symmetric encryption schemes*—one for the low key level $\gamma = \mathtt{LK}$, and one for the high key level $\gamma = \mathtt{HK}$—represented by triples $S\mathcal{E}_\gamma = (\mathcal{K}_\gamma, \mathcal{E}_\gamma, \mathcal{D}_\gamma)$, where

- $\mathcal{K}_\gamma$ is a *key generation* algorithm that on each invocation generates a new key.
- $\mathcal{E}_\gamma$ is a nondeterministic *encryption* algorithm that takes a key $k \in \mathit{Key}_\gamma$, a value $v \in \mathit{Value}$ and returns a ciphertext $u \in U_\gamma$.
- $\mathcal{D}_\gamma$ is a deterministic *decryption* algorithm that takes a key $k \in \mathit{Key}_\gamma$, a ciphertext $u \in U_\gamma$ and returns a value $v \in \mathit{Value}$ or fails. Decryption should satisfy $\mathcal{D}_\gamma(k, \mathcal{E}_\gamma(k, v)) = v$.

The reason for the use of different encryption schemes for different security levels is to lay the ground for extensions to systems with more than two security levels. In such a system we would have one encryption schema at each security level, trusted to encrypt values up to and including the security level.

$$
\begin{array}{llcl}
\text{sec. levels} & \sigma & ::= & \mathtt{L} \mid \mathtt{H} \\
\text{key levels} & \gamma & ::= & \mathtt{LK} \mid \mathtt{HK} \mid \mathtt{RK}
\end{array}
\qquad
\begin{array}{llcl}
\text{basic types} & t & ::= & \mathtt{int} \mid \mathtt{enc}_{\gamma}\ \tau \\
\text{prim. types} & \tau & ::= & t\ \sigma \mid \mathtt{key}\ \gamma \mid (\tau, \tau)
\end{array}
$$

$$
\begin{array}{llcl}
e & ::= & n \mid x \mid e\ op\ e \mid \mathtt{encrypt}_{\gamma}\ (e,e) \mid \mathtt{decrypt}_{\gamma}\ (e,e) \mid (e,e) \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \\
c & ::= & \mathtt{skip} \mid x := e \mid c; c \mid \mathtt{if}\ e\ \mathtt{then}\ c\ \mathtt{else}\ c \mid \mathtt{while}\ e\ \mathtt{do}\ c \mid \mathtt{newkey}(x, \gamma) \mid \mathtt{release}(e)
\end{array}
$$

Figure 4. Enriched language

As stated above, the key sets $Key_{\mathtt{LK}}$ and $Key_{\mathtt{HK}}$ of the two different encryption schemes are distinct. We assume that lk ranges over $Key_{\mathtt{LK}}$ and hk over $Key_{\mathtt{HK}}$.

The *full environment* (or, simply, environment) E is a tuple (M, G, R), where the *memory environment* M is a mapping from variable names to values; the *key-stream environment* G is a mapping from key levels to streams that generate fresh keys; and the *released-key environment* R is a set of released keys.

Semantics for expressions Similarly to the simple language in Section 2, the semantics for expressions have the form $\langle M, e\rangle \Downarrow v$, where v is the result of evaluating expression e in memory M.

Figure 5 presents the rules specific to the treatment of cryptography. Encryption (S-ENC) and decryption (S-DEC) both use the encryption schemes $S\mathcal{E}_{\gamma}$ introduced above. The rest of the rules can be found in the full version [5] of the paper.

Semantics for commands Similarly to Section 2, the semantics for commands have either the form $\langle E, c\rangle \xrightarrow{\alpha} \langle E', c'\rangle$, where E and E' are the initial and resulting environments, c and c' are the initial and resulting commands, and $\alpha \in \{\epsilon, \ell\}$ is an event annotation (where ℓ is the projection $E'_{\mathtt{L}}$ of the environment $E' = (M', G', R')$ defined as $E'_{\mathtt{L}} = (M'_{\mathtt{L}}, G'(\mathtt{L}), R')$); or the form $\langle E, c\rangle \xrightarrow{\downarrow} E'$, which indicates termination in the environment E'. The distinctive primitive of the language is a key release command $\mathtt{release}(k)$ that updates the set of released keys with the value of the key k, in case the key is high. In Figure 6, we display the two most interesting rules for commands. Key generation (S-NEWKEY) takes a variable name and a level of the key to be generated and assigns the topmost element in the key stream associated to that level in the key-stream environment to the variable. The rule (S-KEY-RELEASE) for the key release primitive generates a release event and updates the set of released keys R in the environment. The rest of the rules can be found in the full version [5].

4 Gradual release for the enriched language

This section applies the gradual release approach to the language defined in Section 3. Although the language does not have a declassification construct, we show how to model revelation-based declassification using key generation, encryption, key release, and decryption. We formally connect this model to gradual release in the simple language from Section 2.

Encryption model We adopt the encryption model of [3] and consider nondeterministic encryption schemes with initial vectors. As foreshadowed in Section 3, encryption schemes are represented as triples of the form $(\mathcal{K}, \mathcal{E}, \mathcal{D})$, where $\mathcal{K}$ is a key generation algorithm, $\mathcal{E}$ is an encryption algorithm, and $\mathcal{D}$ is a decryption algorithm. The encryption algorithm is a function of a key, a plaintext, and an *initial vector* (which we sometimes omit when its value is unimportant). For the same key and plaintext it returns different ciphertexts depending on the value of the initial vector. Two ciphertexts are *low-related* if they have been encrypted with the same initial vector iv:

$$\forall k_1, k_2, v_1, v_2\ .\ \mathcal{E}(k_1, v_1, iv) \doteq \mathcal{E}(k_2, v_2, iv)$$

This relation has the following properties: (i) different ciphertexts produced by one plaintext and one key have different initial vectors and are not low-related, and (ii) since every plaintext and key produce ciphertexts using all initial vectors, for each ciphertext produced by one plaintext and key there will be exactly one low-related ciphertext for every other choice of plaintext and key.

This construction prevents occlusion, which would happen if we treated all ciphertexts as equal, as illustrated in an example that follows Proposition 4.

Low-equivalence Let $E_1 \sim_{\Gamma} E_2$ denote that the environments E_1 and E_2 are low-equivalent with respect to the environment type Γ (i.e., their low projections are the same).

The *low-equivalence* relation, presented in Figure 7, draws on one in [3]. In addition, it also reflects that once a key has been released, it is unsafe to encrypt with this key or, more precisely, assign the result of the encryption to a low variable.

Low-equivalence is defined structurally with respect to the type of its arguments and the set of released keys $\hat{r}$. For example, two high keys are indistinguishable (i.e., related by low-equivalence) if none of them has been released (rule (LE-KEY-SK)). On the other hand, only equivalent values of

$$\text{(S-ENC)}\ \frac{\langle M, e_1\rangle \Downarrow k \quad \langle M, e_2\rangle \Downarrow v \quad k \in Key_\gamma \quad u = \mathcal{E}_\gamma(k, v)}{\langle M, \mathtt{encrypt}_\gamma\ (e_1, e_2)\rangle \Downarrow u}$$

$$\text{(S-DEC)}\ \frac{\langle M, e_1\rangle \Downarrow k \quad \langle M, e_2\rangle \Downarrow u \quad k \in Key_\gamma \quad v = \mathcal{D}_\gamma(k, u)}{\langle M, \mathtt{decrypt}_\gamma\ (e_1, e_2)\rangle \Downarrow v}$$

Figure 5. Selected semantic rules for expressions

$$\text{(S-NEWKEY)}\ \frac{G(\gamma) = k \cdot ks}{\langle (M, G, R), \mathtt{newkey}(x, \gamma)\rangle \longrightarrow (M[x \mapsto k], G[\gamma \mapsto ks], R)}$$

$$\text{(S-KEY-RELEASE)}\ \frac{\langle M, e\rangle \Downarrow k \quad k \in Key_\gamma}{\langle (M, G, R), \mathtt{release}(e)\rangle \xrightarrow{r:(M_{\mathtt{L}}, G(\mathtt{L}), S\cup R)} (M, G, S \cup R)} \quad \text{where } S = \begin{cases} \{k\} & \text{if } \gamma = \mathtt{HK}, \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 6. Selected semantic rules for commands

low and released keys are indistinguishable (rules (LE-KEY-SK) and (LE-KEY-RK)).

The rules that define the low-equivalence of encrypted values are worth highlighting. The rules (LE-ENC-L1) and (LE-ENC-L2) both require ciphertexts to be low-related by the relation $\doteq$ on encrypted values. Their additional demands on the ciphertexts depend on whether the encryption key is released and what its type is. The first rule (LE-ENC-L1), when $\gamma = \mathtt{RK}$, requires that low-equivalent ciphertexts obtained with released keys must agree on the value of the key and plaintext (because anyone can decrypt them). This is achieved by demanding that keys and plaintexts are low-equivalent with respect to the types $tolow(\mathtt{key}\ \gamma)$ and $tolow(\tau)$. The function $tolow(\cdot)$, defined in Figure 7, converts high primitive types into low ones.

Similarly to released keys, the demand for low keys (LE-ENC-L1, $\gamma = \mathtt{LK}$) is that plaintexts are low-equivalent with respect to the type $tolow(\tau)$. The only demand for encryption with high keys (LE-ENC-L2) is that the resulting values should be low-equivalent with respect to the primitive type τ of the encryption type.

Uninitialized values (denoted by $\bullet$) are low-equivalent (rule (LE-ENC-L3)). The rule (LE-ENC-H) relates high ciphertexts if one of them is uninitialized or their primitive type structures are the same (an auxiliary predicate $struct(\tau, v)$, which holds whenever v has type structure τ, is given in Figure 7). Note that an uninitialized value is low-equivalent to any other value. This is also reflected in the rule (LE-MEM), where $v^\bullet$ is either a value or $\bullet$.

Gradual release for the enriched language Before we proceed to gradual release for the enriched language, we define low-equivalence on event sequences.

Note that the released-key environments, which are explicitly recorded by low events, provide only a partial view of what keys may be known to the attacker. For example, the value of the key k_2 in the expression $\mathtt{encrypt}_{\mathtt{HK}}\ (k_1, (x, k_2))$ is not contained in the released-key environment after k_1 is released. Nevertheless, using k_2 for encrypting high values after the release of k_1 would be disastrous since the attacker knows the exact value of k_2 and, hence, can decrypt ciphertexts encrypted with it. Therefore, the set of *all released keys* $\hat{r}$ is computed, in a Dolev-Yao style [14], by traversing low events and accumulating the keys that depend on the ones that are already released. We use an accumulator function $r(\tau, v, R)$ defined inductively on a variable type τ, a value of that type v, and an initial set of released keys R:

$$\begin{aligned}
&r(\mathtt{enc}_\gamma\ \tau\ \mathtt{L}, u, R) = \{r(tolow(\tau), v, R) \mid \\
&\qquad\qquad \exists v, k\ .\ v = \mathcal{D}_\gamma(k, u) \wedge (k \in R \vee \gamma = \mathtt{LK})\} \\
&r(\mathtt{enc}_\gamma\ \tau\ \mathtt{H}, u, R) = \emptyset \qquad r(\mathtt{int}\ \sigma, v, R) = \emptyset \\
&r(\mathtt{key}\ \mathtt{LK}, k, R) = \emptyset \qquad r(\mathtt{key}\ \mathtt{HK}, k, R) = \emptyset \\
&r(\mathtt{key}\ \mathtt{RK}, k, R) = \{k\} \\
&r((\tau_1, \tau_2), (v_1, v_2), R) = r(\tau_1, v_1, R) \cup r(\tau_2, v_2, R)
\end{aligned}$$

For a given typing environment Γ and a sequence $\vec{\ell}$ of low events $(\ell_1, \ldots, \ell_n)$, where $\ell_i = (M_i, G_i, R_i)$, we let:

$$\begin{aligned}
r_0 &= R_n, \quad \text{and} \\
r_j &= \textstyle\bigcup_{i=1}^{n} \{r(\Gamma(x), M_i(x), r_{j-1}) \mid x \in \mathtt{dom}\ (\Gamma)\}
\end{aligned}$$

The set $\hat{r}$ is then defined as r_k such that $r_k = r_{k+1}$. Such a k exists because the set of keys that can be extracted from the sequence $\vec{\ell}$ of low events is finite.

Two sequences of low events $\vec{\ell}$ and $\vec{\ell'}$ are low-equivalent if the numbers of events in each of the sequences are the same, the sets of all released keys that are computed from each of them are equivalent, and the sequences agree on every respective element, i.e., memories, key generating streams, and released-key environments are low-equivalent:

$$\text{(LE-KEY-LK)}\ \frac{}{lk \sim^{\hat{r}}_{\texttt{key LK}} lk} \qquad \text{(LE-INT-L)}\ \frac{}{n \sim^{\hat{r}}_{\texttt{int L}} n} \qquad \text{(LE-INT-H)}\ \frac{}{n_1 \sim^{\hat{r}}_{\texttt{int H}} n_2}$$

$$\text{(LE-KEY-RK)}\ \frac{hk \in \hat{r}}{hk \sim^{\hat{r}}_{\texttt{key RK}} hk} \qquad \text{(LE-ENC-L3)}\ \frac{}{\bullet \sim^{\hat{r}}_{\texttt{enc}_\gamma\ \tau\ \texttt{L}} \bullet} \qquad \text{(LE-PAIR)}\ \frac{v_{11} \sim^{\hat{r}}_{\tau_1} v_{21} \quad v_{12} \sim^{\hat{r}}_{\tau_2} v_{22}}{(v_{11}, v_{12}) \sim^{\hat{r}}_{(\tau_1,\tau_2)} (v_{21}, v_{22})}$$

$$\text{(LE-KEY-SK)}\ \frac{hk_i \notin \hat{r} \quad i = 1,2}{hk_1 \sim^{\hat{r}}_{\texttt{key HK}} hk_2} \qquad \text{(LE-MEM)}\ \frac{\forall x \in \texttt{dom}\ (\Gamma).M_i(x) = v_i^\bullet\ (i = 1,2) \implies v_1^\bullet \sim^{\hat{r}}_{\Gamma(x)} v_2^\bullet}{M_1 \sim^{\hat{r}}_\Gamma M_2}$$

$$\text{(LE-KGEN1)}\ \frac{k_1 \sim^{\hat{r}}_{\texttt{key}\ \gamma} k_2 \quad K_1 \sim^{\hat{r}}_\gamma K_2}{k_1 \cdot K_1 \sim^{\hat{r}}_\gamma k_2 \cdot K_2} \qquad \text{(LE-KGEN2)}\ \frac{G_1(\texttt{HK}) \sim^{\hat{r}}_{\texttt{HK}} G_2(\texttt{HK}) \quad G_1(\texttt{LK}) \sim^{\hat{r}}_{\texttt{LK}} G_2(\texttt{LK})}{G_1 \sim^{\hat{r}} G_2}$$

$$\text{(LE-ENC-H)}\ \frac{\exists j\ .\ u_j = \bullet\ \vee\ struct(\texttt{enc}_\gamma\ \tau\ \texttt{H}, u_i)\ (i = 1,2)}{u_1 \sim^{\hat{r}}_{\texttt{enc}_\gamma\ \tau\ \texttt{H}} u_2} \qquad \text{(LE-ENC-L2)}\ \frac{\exists v_i, k_i\ .\ v_i = \mathcal{D}_\gamma(k_i, u_i) \quad i = 1,2 \quad k_1 \sim^{\hat{r}}_{\texttt{key HK}} k_2 \quad v_1 \sim^{\hat{r}}_\tau v_2 \quad u_1 \doteq u_2}{u_1 \sim^{\hat{r}}_{\texttt{enc}_{\texttt{HK}}\ \tau\ \texttt{L}} u_2}$$

$$\text{(LE-ENC-L1)}\ \frac{\exists v_i, k_i\ .\ v_i = \mathcal{D}_\gamma(k_i, u_i) \quad i = 1,2 \quad k_1 \sim^{\hat{r}}_{tolow(\texttt{key}\ \gamma)} k_2 \quad v_1 \sim^{\hat{r}}_{tolow(\tau)} v_2 \quad u_1 \doteq u_2}{u_1 \sim^{\hat{r}}_{\texttt{enc}_\gamma\ \tau\ \texttt{L}} u_2}$$

$$tolow(t\ \sigma) = t\ \texttt{L} \qquad tolow(\texttt{key LK}) = \texttt{key LK}$$
$$tolow(\texttt{key HK}) = \texttt{key RK} \qquad tolow(\texttt{key RK}) = \texttt{key RK}$$
$$tolow((\tau_1, \tau_2)) = (tolow(\tau_1), tolow(\tau_2))$$

$$\frac{}{struct(\texttt{int}\ \sigma, n)} \qquad \frac{}{struct(\texttt{key}\ \gamma, k)}$$

$$\frac{struct(\tau_1, v_1) \quad struct(\tau_2, v_2)}{struct((\tau_1, \tau_2), (v_1, v_2))}$$

$$\frac{\exists v, k\ .\ v = \mathcal{D}_\gamma(k, u) \quad struct(\texttt{key}\ \gamma, k) \quad struct(\tau, v)}{struct(\texttt{enc}_\gamma\ \tau\ \sigma, u)}$$

Figure 7. Low-equivalence

$$\frac{R_i = R'_i \quad \hat{r}(\vec{\ell}_n) = \hat{r}(\vec{\ell'}_n) = \hat{r} \quad M_i \sim^{\hat{r}}_\Gamma M'_i \quad G_i \sim^{\hat{r}} G'_i \quad i = 1 \ldots n}{\vec{\ell}_n \sim_\Gamma \vec{\ell'}_n}$$

Similarly to the simple language, we define the set of possible low-event sequences. We assume that initial environments (which we indicate with a superscript as in E^0) have the form $(M, G, \emptyset)$, i.e., their released-key sets are empty. In such an environment, we let $M(x) = \bullet$ for all such x that $x : \texttt{enc}_\gamma\ \tau\ \sigma$, i.e., all variables of ciphertext type are uninitialized. Given a command c and a projection of an initial environment $E^0_{\texttt{L}}$, the set of possible low events $L(c, E^0_{\texttt{L}})$ is:

$$L(c, E^0_{\texttt{L}}) \triangleq \{\vec{\ell} \mid \exists E, E'\ .\ E_{\texttt{L}} \sim_\Gamma E^0_{\texttt{L}} \wedge \langle E, c\rangle \overset{\vec{\ell}}{\longrightarrow}{}^* E'\}$$

The definition of knowledge is in order, also similar to the one for the simple language. Given the low projection of an initial environment $E^0_{\texttt{L}}$ and a (possibly empty) sequence of low events $\vec{\ell} \in \widehat{L}(c, E^0_{\texttt{L}})$, where $\widehat{L}(c, E^0_{\texttt{L}})$ denotes the prefix closure of $L(c, E^i_{\texttt{L}})$, the *knowledge* $k(c, E^0_{\texttt{L}}, \vec{\ell})$ of the attacker is defined as:

$$k(c, E^0_{\texttt{L}}, \vec{\ell}) \triangleq \{E \mid E_{\texttt{L}} \sim_\Gamma E^0_{\texttt{L}} \wedge \exists E', c', \vec{\ell'}\ .\ (\langle E, c\rangle \overset{\vec{\ell'}}{\longrightarrow}{}^* \langle E', c'\rangle \vee \langle E, c\rangle \overset{\vec{\ell'}}{\longrightarrow}{}^* E') \wedge \vec{\ell} \sim_\Gamma \vec{\ell'}\}$$

A new ingredient—the low-equivalence of low-event sequences $\vec{\ell} \sim_\Gamma \vec{\ell'}$—is used in the definition, which allows for relating ciphertexts that are obtained from different keys and plaintexts. This prevents the knowledge from becoming more precise. Otherwise, publishing a result of an encryption with a high key would narrow down the knowledge about the key and the plaintext behind the ciphertext to their exact values, which is infeasible to infer for the attacker.

Accordingly, the definition of the initial knowledge is:

$$k(c, E^0_{\texttt{L}}) \triangleq \{E \mid E_{\texttt{L}} \sim_\Gamma E^0_{\texttt{L}} \wedge \exists E', \vec{\ell}\ .\ \langle E, c\rangle \overset{\vec{\ell}}{\longrightarrow}{}^* E'\}$$

and the termination-insensitive knowledge is:

$$k_\downarrow(c, E^0_{\texttt{L}}, \vec{\ell}) \triangleq k(c, E^0_{\texttt{L}}, \vec{\ell}) \cap k(c, E^0_{\texttt{L}})$$

The definition of gradual release for the enriched language follows Definition 2 for the simple language:

Definition 3 (Gradual release)**.** *A command c satisfies* gradual release *if for all low projections of initial environments $E^0_{\texttt{L}}$ such that $R^0 = \emptyset$ and sequences of low events $\vec{\ell} \in L(c, E^0_{\texttt{L}})$, where $\vec{\ell_n} = (\ell_1, \ldots, \ell_n), n \geq 1$, of which $\ell_{r_1}, \ldots, \ell_{r_m}$ are all release events, we have:*

$$\forall i\ .\ 1 \leq i \leq n\ .\ (\forall j\ .\ r_j \neq i) \implies k_\downarrow(c, E^0_{\texttt{L}}, \vec{\ell}_{i-1}) = k_\downarrow(c, E^0_{\texttt{L}}, \vec{\ell}_i)$$

where $k_{\downarrow}(c, E^0_{\mathtt{L}}, \vec{\ell}_0) \triangleq k(c, E^0_{\mathtt{L}})$.

We illustrate the definition by simple examples (more interesting examples are deferred to Section 5).

The following program is accepted by gradual release:

```
k := newkey(HK); l := encrypt_HK(k, h);
```

Different low-event sequences that can be produced by this program correspond to different values of the variable l. Since the encryption key is high, every value of l is low-equivalent to ciphertexts produced by all other possible keys and plaintexts. Therefore, for each possible sequence of low events that this program may generate, the knowledge contains all possible keys and plaintexts, i.e., it is equal to the initial knowledge and *remains constant* throughout all runs.

Gradual release rejects the following program:

```
k := newkey(HK); l := encrypt_HK(k, h); l':= h;
```

Similar to an example in Section 2, the last assignment narrows down the set of possible initial values for the variable h to exactly one.

On the other hand, gradual release accepts this program:

```
k := newkey(HK); l:=encrypt_HK(k, h); release (k); l':= h;
```

There are four low events in this program: the encryption, release statement, last assignment, and termination. As in the previous example, publishing the result of encryption does not change the knowledge. The release statement, however, triggers a change in the low-equivalence for the value of l: once the key k is released, it is only related with ciphertexts that agree with l both on the value of the key and the plaintext. This corresponds to refining the knowledge to the exact values of k and h. Thus, the last assignment to l' does not make the knowledge more precise. Gradual release accepts this program since the only point where the knowledge changes is the release event.

Conservativity with respect to cryptographically-masked flows As a sanity check, we demonstrate that Definition 3 is a conservative extension of *possibilistic* noninterference.

Our *possibilistic security* definition is based on the one of cryptographically-masked flows [3], although it represents an attacker that observes event sequences rather than final environments. Assume $T(c, E)$ is the set of possible event sequences generated by a command c in an environment E. A command c satisfies possibilistic noninterference if:

$$\forall E_1, E_2 \,.\, E_1 \sim_\Gamma E_2 \wedge T(c, E_j) \neq \emptyset, j = 1, 2 \Longrightarrow \\ \forall \vec{\ell}_1 \in T(c, E_1)\ \exists \vec{\ell}_2 \in T(c, E_2) \,.\, \vec{\ell}_1 \sim_\Gamma \vec{\ell}_2$$

That is, for every pair of low-equivalent environments and configurations that terminate in these environments, and for each event sequence generated by running the first configuration there is a low-equivalent event sequence generated by running the second one (and vice versa by symmetry).

Proposition 4. *If a command c is free of key release primitives, then c satisfies gradual release if and only if c satisfies possibilistic noninterference.*

This proposition parallels Proposition 3. If there is no key release, then the attacker's knowledge must stay unchanged throughout execution. Therefore, this proposition reduces to showing an analogue of Proposition 2, which is straightforward.

Consider the following code, which is inspired by an occlusion example from [3]. The program below is intuitively insecure: depending on the value of h the value of y is either a new encryption, or the copy of x. An attacker observing that x and y are different can derive that the first branch has been taken.

```
x := encrypt_HK(k1, h1);
if h then y := encrypt_HK(k2, h2);
     else y := x;
```

Thanks to the initial-vector mechanism (described in the beginning of the section), the program is rejected by both gradual release and possibilistic noninterference. An initial memory M, where the value for h is 1, produces a sequence of low events where the final values v_x and v_y of x and y, respectively, are such that $v_x \neq v_y$. On the other hand, a memory M', where the value of h is 0 is not a part of the knowledge. Indeed, under M', it is possible to produce a low-event sequence where $v = v_x$ and v is the final value of both x and y under M', but then, since the second branch is taken, it implies $v \neq v_y$. This means that we cannot reach the final memory with the value v_y for y from the memory M', and, therefore, M' is not a part of the knowledge. Because the knowledge is refined, and there are no release events to justify the refinement, the above program is rejected.

Relation to gradual release for the simple language As another sanity check, we establish a relation to the definition of gradual release (Definition 2) for the simple language (Section 2). The combination of the key generation, encryption/decryption, and key release features in the enriched language turns out to be crucial for connecting encryption-based declassification to revelation-based information release. We can translate the general revelation-based declassification command in the following way:

$$\begin{aligned} l := \ & \mathtt{declassify}(e) \hookrightarrow \\ & k := \mathtt{newkey}(\mathtt{HK}); t := \mathtt{encrypt}_{\mathtt{HK}}(k, e); \\ & \mathtt{release}(k); l := \mathtt{decrypt}_{\mathtt{RK}}(k, t); \end{aligned}$$

for some fresh temporary variable t. Assuming that the transformation does not change other commands than de-

classifications, we arrive at the following formal connection between the two definitions.

Proposition 5. *A command c in the simple language satisfies gradual release (Definition 2) if and only if command c' obtained from c by transformation ($c \hookrightarrow c'$) in the enriched language satisfies gradual release (Definition 3).*

As an example, consider the following command in the simple language:

```
if h then l := declassify(h1);
     else l := declassify(h2);
l' := h;
```

The transformation produces the following result in the enriched language:

```
if h then k := newkey(HK);
          t := encrypt_HK(k, h1);
          release (k);
          l := decrypt_RK(k, t);
     else
          k := newkey(HK);
          t := encrypt_HK(k, h2);
          release (k);
          l := decrypt_RK(k, t);
l' := h;
```

This command is semantically equivalent to the following:

```
k := newkey (S);
if h then h' := h1 else h' := h2;
t := encrypt_HK(k, h');
release (k);
l := decrypt_RK(k, t);
l':= h;
```

In the context with no information about the values of the variables of h_1 and h_2, this program (as well as the source of the transformation) is rejected by the gradual release since the last assignment leaks the exact value of the variable h that is unknown to the attacker otherwise.

5 Enforcement

This section presents a type and effect system that enforces gradual release for the enriched language. The type system uses an extended typing environment in which type annotations for keys are lifted to contain unique names $\kappa \in \mathit{KeyNames}$:

$$\begin{array}{lrcl} \text{ext. basic types} & \tilde{t} & ::= & \mathtt{int} \mid \mathtt{enc}_{\kappa}\ \tilde{\tau} \\ \text{ext. prim. types} & \tilde{\tau} & ::= & \tilde{t}\ \sigma \mid \mathtt{key}\ \kappa \mid (\tilde{\tau}, \tilde{\tau}) \end{array}$$

Such an environment can be easily obtained by enumerating all occurrences of key types in the variable environment Γ; the resulting extended environment is denoted as $\tilde{\Gamma}$.

The type system is also parameterized over a dependency analysis that tracks possible and definite key dependencies in a program. A dependency relation is maintained throughout the typing rules, which makes the type system aware of the key names that *may* or *must* have been released at every program point. Since key names may appear inside tuples and encryption types, dependency relations track dependencies between extended types. The typing rules for commands thus have the form $\mathcal{A}, \tilde{\Gamma}, pc \vdash c \Rightarrow \mathcal{A}'$, where $\mathcal{A}$ and $\mathcal{A}'$ are the initial and final dependency relations.

To access the dependency relation, the typing rules employ an interface which we describe below. An accompanying technical report [5] describes a graph transformation-based implementation that matches this interface.

Analysis interface and requirements Let $\mathcal{A}$ denote a key dependency relation. The interface for accessing $\mathcal{A}$ is then defined by the following syntax:

$$\begin{array}{lrcl} \text{type connectors} & \mathit{conn} & ::= & \tilde{\tau}\ \nearrow \mid \tilde{\tau}_1 \rightarrow \tilde{\tau}_2 \mid \kappa \searrow \\ \text{analysis transf.} & & ::= & \mathtt{upd}\ (\mathcal{A}, \mathit{conn}) \\ \text{analysis comb.} & & ::= & \mathtt{join}\ (\mathcal{A}_1, \mathcal{A}_2) \\ \text{fresh type} & & ::= & \mathtt{fresh}\ (\mathcal{A}, \tau) \\ \text{may-type} & & ::= & \mathtt{type}_{\mathtt{May}}\ (\mathcal{A}, \tilde{\tau}) \\ \text{must-type} & & ::= & \mathtt{type}_{\mathtt{Must}}\ (\mathcal{A}, \tilde{\tau}) \end{array}$$

Type connectors together with the operator $\mathtt{upd}\ (\mathcal{A}, \mathit{conn})$ are used for updating the dependency relation. The $\tilde{\tau}\ \nearrow$ connector marks the extended type $\tilde{\tau}$ as unrelated to any other type. The $\tilde{\tau}_1 \rightarrow \tilde{\tau}_2$ connector indicates that if $\tilde{\tau}_1$ is released then $\tilde{\tau}_2$ is released as well. The $\kappa \searrow$ connector specifies that the key name κ is released.

Joining relations is done by the operator $\mathtt{join}\ (\mathcal{A}_1, \mathcal{A}_2)$ which combines the information from $\mathcal{A}_1$ and $\mathcal{A}_2$. The operator $\mathtt{fresh}\ (\mathcal{A}, \tau)$ takes a dependency relation $\mathcal{A}$ and a primitive type τ and returns an extended type $\tilde{\tau}$ that has the same structure as τ, but the key names in it are fresh. This operator may be used in the extended environment construction and in the typing rule for variable lookup.

Obtaining the results of the analysis is done via the functions $\mathtt{type}_{\mathtt{May}}\ (\mathcal{A}, \tilde{\tau})$ and $\mathtt{type}_{\mathtt{Must}}\ (\mathcal{A}, \tilde{\tau})$. These functions cast the extended type $\tilde{\tau}$ to a primitive type taking into account all possible (resp. definite) key dependencies in $\mathcal{A}$.

Given an environment $E = (M, G, R)$, a dependency relation $\mathcal{A}$, and an extended typing environment $\tilde{\Gamma}$, we say that $\mathcal{A}$ *enforces* E, denoted by $\mathcal{A}, \tilde{\Gamma} \models E$, when for all key names that occur in $\tilde{\Gamma}$ the following holds:

1. If the analysis returns that a key name may not have been released, then key values in the memory environment associated with that name are not in the set of all released keys

2. If the analysis returns that a key name must have been released, then key values in the memory environment associated with that name are in the set of all released keys.

Our demands on the analysis implementation can be expressed as follows:

$$\text{(T-VAR)}\ \frac{\tilde{\Gamma}(x) = \tilde{\tau}_1 \qquad \tilde{\tau}' : \texttt{fresh}\ (\mathcal{A}, \Gamma(x)) \qquad \mathcal{A}' = \texttt{upd}\ (\mathcal{A}, \tilde{\tau}' \nearrow, \tilde{\tau}' \to \tilde{\tau}_1, \tilde{\tau}_1 \to \tilde{\tau}')}{\mathcal{A}, \tilde{\Gamma} \vdash x : \tilde{\tau}', \mathcal{A}'}$$

$$\text{(T-ENC1)}\ \frac{\mathcal{A}, \tilde{\Gamma} \vdash e_1 : \texttt{key}\ \kappa, \mathcal{A}_1 \qquad \mathcal{A}_1, \tilde{\Gamma} \vdash e_2 : \tilde{\tau}, \mathcal{A}_2 \qquad \texttt{type}_{\texttt{May}}\ (\mathcal{A}_1, \texttt{key}\ \kappa) = \texttt{key HK} \qquad \mathcal{A}_3 = \texttt{upd}\ (\mathcal{A}_2, \texttt{key}\ \kappa \to \tilde{\tau})}{\mathcal{A}, \tilde{\Gamma} \vdash \texttt{encrypt}_{\texttt{HK}}\ (e_1, e_2) : \texttt{enc}_{\kappa}\ \tilde{\tau}\ \texttt{L}, \mathcal{A}_3}$$

$$\text{(T-ENC2)}\ \frac{\mathcal{A}, \tilde{\Gamma} \vdash e_1 : \texttt{key}\ \kappa, \mathcal{A}_1 \qquad \mathcal{A}_1, \tilde{\Gamma} \vdash e_2 : \tilde{\tau}, \mathcal{A}_2 \qquad \texttt{type}_{\texttt{May}}\ (\mathcal{A}_1, \texttt{key}\ \kappa) = \texttt{key}\ \gamma \qquad \gamma \neq \texttt{HK} \qquad lvl(\tilde{\tau}) = \sigma \qquad \mathcal{A}_3 = \texttt{upd}\ (\mathcal{A}_2, \texttt{key}\ \kappa \to \tilde{\tau})}{\mathcal{A}, \tilde{\Gamma} \vdash \texttt{encrypt}_{\gamma}\ (e_1, e_2) : \texttt{enc}_{\kappa}\ \tilde{\tau}\ \sigma, \mathcal{A}_3}$$

$$\text{(T-DEC1)}\ \frac{\mathcal{A}, \tilde{\Gamma} \vdash e_1 : \texttt{key}\ \kappa_1, \mathcal{A}_1 \qquad \mathcal{A}_1, \tilde{\Gamma} \vdash e_2 : \texttt{enc}_{\kappa_2}\ \tilde{\tau}\ \sigma, \mathcal{A}_2 \qquad \texttt{type}_{\texttt{Must}}\ (\mathcal{A}_1, \texttt{key}\ \kappa_1) = \texttt{key}\ \gamma}{\mathcal{A}, \tilde{\Gamma} \vdash \texttt{decrypt}_{\gamma}\ (e_1, e_2) : \tilde{\tau}^{\sigma}, \mathcal{A}_2}$$

$$\text{(T-DEC2)}\ \frac{\mathcal{A}, \tilde{\Gamma} \vdash e_1 : \texttt{key}\ \kappa_1, \mathcal{A}_1 \qquad \mathcal{A}_1, \tilde{\Gamma} \vdash e_2 : \texttt{enc}_{\kappa_2}\ \tilde{\tau}\ \sigma, \mathcal{A}_2 \qquad \texttt{type}_{\texttt{Must}}\ (\mathcal{A}_1, \texttt{key}\ \kappa_1) = \texttt{key RK} \qquad \texttt{type}_{\texttt{Must}}\ (\mathcal{A}_2, \texttt{key}\ \kappa_2) = \texttt{key RK}}{\mathcal{A}, \tilde{\Gamma} \vdash \texttt{decrypt}_{\texttt{RK}}\ (e_1, e_2) : tolow(\tilde{\tau})^{\sigma}, \mathcal{A}_2}$$

Figure 8. Selected type rules for expressions

$$\text{(T-SKIP)}\ \frac{}{\mathcal{A}, \tilde{\Gamma}, pc \vdash \texttt{skip} \Rightarrow \mathcal{A}}$$

$$\text{(T-ASGN)}\ \frac{\tilde{\Gamma}(x) = \tilde{\tau} \qquad \mathcal{A}, \tilde{\Gamma} \vdash e : \tilde{\tau}', \mathcal{A}' \qquad pc \sqsubseteq lvl(\Gamma(x)) \qquad \texttt{type}_{\texttt{May}}\ (\mathcal{A}', \tilde{\tau}') <: \Gamma(x) \qquad \texttt{type}_{\texttt{Must}}\ (\mathcal{A}', \tilde{\tau}') <: \Gamma(x) \qquad \mathcal{A}'' = \texttt{upd}\ (\mathcal{A}', \tilde{\tau} \nearrow, \tilde{\tau} \to \tilde{\tau}', \tilde{\tau}' \to \tilde{\tau})}{\mathcal{A}, \tilde{\Gamma}, pc \vdash x := e \Rightarrow \mathcal{A}''}$$

$$\text{(T-SEQ)}\ \frac{\mathcal{A}, \tilde{\Gamma}, pc \vdash c_1 \Rightarrow \mathcal{A}' \qquad \mathcal{A}', \tilde{\Gamma}, pc \vdash c_2 \Rightarrow \mathcal{A}'''}{\mathcal{A}, \tilde{\Gamma}, pc \vdash c_1; c_2 \Rightarrow \mathcal{A}''}$$

$$\text{(T-IF)}\ \frac{\mathcal{A}, \tilde{\Gamma} \vdash e : \texttt{int}\ \sigma, \mathcal{A}' \qquad \mathcal{A}', \tilde{\Gamma}, pc \sqcup \sigma \vdash c_i \Rightarrow \mathcal{A}'_i \qquad i = 1, 2}{\mathcal{A}, \tilde{\Gamma}, pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Rightarrow \texttt{join}\ (\mathcal{A}'_1, \mathcal{A}'_2)}$$

$$\text{(T-WHILE)}\ \frac{\texttt{join}\ (\mathcal{A}, \mathcal{A}''), \tilde{\Gamma} \vdash e : \texttt{int}\ \sigma, \mathcal{A}' \qquad \mathcal{A}', \tilde{\Gamma}, pc \sqcup \sigma \vdash c \Rightarrow \mathcal{A}''}{\mathcal{A}, \tilde{\Gamma}, pc \vdash \texttt{while}\ e\ \texttt{do}\ c \Rightarrow \texttt{join}\ (\mathcal{A}', \mathcal{A}'')}$$

$$\text{(T-NEWKEY)}\ \frac{\Gamma(x) = \texttt{key}\ \gamma \qquad \gamma \neq \texttt{RK} \qquad \tilde{\Gamma}(x) = \tilde{\tau} \qquad \mathcal{A}' = \texttt{upd}\ (\mathcal{A}, \tilde{\tau} \nearrow) \qquad pc \sqsubseteq lvl(\Gamma(x))}{\mathcal{A}, \tilde{\Gamma}, pc \vdash \texttt{newkey}(x, \gamma) \Rightarrow \mathcal{A}'}$$

$$\text{(T-RELEASE)}\ \frac{\mathcal{A}, \tilde{\Gamma} \vdash e : \texttt{key}\ \kappa, \mathcal{A}' \qquad pc = \texttt{L} \qquad \mathcal{A}'' = \texttt{upd}\ (\mathcal{A}', \kappa \searrow)}{\mathcal{A}, \tilde{\Gamma}, pc \vdash \texttt{release}(e) \Rightarrow \mathcal{A}''}$$

Figure 9. Type rules for commands

$$\mathcal{A}, \tilde{\Gamma}, pc \vdash c \Rightarrow \mathcal{A}' \ \wedge\ \mathcal{A}, \tilde{\Gamma} \models E \ \wedge\ \langle E, c\rangle \xrightarrow{\vec{\ell}}{}^{*} E' \implies \mathcal{A}', \tilde{\Gamma} \models E'$$

Expression typing rules The rules for expressions have the form $\mathcal{A}, \tilde{\Gamma} \vdash e : \tau, \mathcal{A}'$, where $\mathcal{A}'$ tracks dependencies created by the expression e. Figure 8 presents the rules for non-standard expressions, while the other rules can be found in [5].

The rule for variable lookup (T-VAR) looks up the type of the variable in the extended environment and creates a fresh type in the dependency relation that is connected to the original variable. The two rules for encryption correspond to encryption with high non-released keys, and to encryption with keys that have low or released levels. In the first case, the rule (T-ENC1) allows the resulting type of the expression to have the low type L, if it is known that the key name used for encryption is definitely not released. The second rule (T-ENC2) preserves the original level of the plaintext σ; it uses the function $lvl(\cdot)$ that returns the security level of its argument:

$$lvl(t\ \sigma) = \sigma \quad lvl((\tau_1, \tau_2)) = lvl(\tau_1) \sqcup lvl(\tau_2)$$
$$lvl(\texttt{key HK}) = \texttt{H} \quad lvl(\texttt{key LK}) = \texttt{L} \quad lvl(\texttt{key RK}) = \texttt{L}$$

Both rules update the dependency relations with information that the release of the ciphertext is now dependent on the release of the encryption key. The rules for decryption require that the level of the key used for decryption matches the key level of the encrypted value. The result of decryption is tainted by the security level of the encrypted value, where the taint function is defined as follows:

$$(t\ \sigma)^{\sigma'} = t\ (\sigma \sqcup \sigma') \quad (\tau_1, \tau_2)^{\sigma} = (\tau_1^{\sigma}, \tau_2^{\sigma})$$
$$(\texttt{key LK})^{\texttt{L}} = \texttt{key LK} \quad (\texttt{key RK})^{\texttt{L}} = \texttt{key RK}$$
$$(\texttt{key HK})^{\sigma} = \texttt{key HK}$$

The interesting case, which uses the must analysis, is the rule (T-DEC2) that relaxes the returning value if both encryption and decryption keys are known to be released.

Command typing rules Figure 9 presents the command type rules. The rules for a command c have the form $\mathcal{A}, \tilde{\Gamma}, pc \vdash c \Rightarrow \mathcal{A}'$ where $\mathcal{A}$ and $\mathcal{A}'$ are the dependency relations before and after executing c.

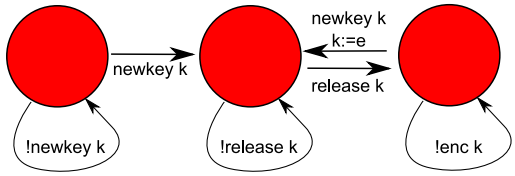

Figure 10. Enforcement for high keys

The rule for assignment (T-ASGN) looks up the type of the variable in both normal and extended environments. It evaluates the type of the expression to assign in the extended environment and its may and must types, as recorded in the dependency relation. We require both of these types to be a subtype of the variable type. Next, we update the dependency relation with the new dependency between extended types. As is standard, the rule also checks that the pc label is bounded by the security level of the assigned variable.

The rule (T-SEQ) propagates the updated dependencies through sequential composition. The rule for conditionals (T-IF) evaluates the security level of the guard to check both branches; the resulting dependency relation is obtained by joining the resulting relations for c_1 and c_2. In a similar flavor, the rule (T-WHILE) demands that there exists a pair of dependency relations $\mathcal{A}'$ and $\mathcal{A}''$ such that we can type the loop body in an environment starting from $\mathcal{A}'$ and producing $\mathcal{A}''$. In addition, $\mathcal{A}'$ is a result of evaluating loop guard in an environment obtained by joining $\mathcal{A}$ and $\mathcal{A}''$.

The rule (T-NEWKEY) for new key generation checks that the pc label is no greater than the level of the key to generate. It looks up the extended type of the key variable in the extended environment and removes dependencies of this type in the dependency relation. The rule (T-RELEASE) is the one where key names are marked as released. It also enforces that release only happens in low context (under low pc).

Figure 10 demonstrates the key-related part of enforcement of the type system for high keys. The type system tracks the state in a simple automaton for each key name and rejects the program if its (statically approximated) execution paths might lead to a transition that is not prescribed by the automaton.

Soundness We denote by $\mathcal{A}^0$ dependency relations that contain no released-key names. The type system provably enforces gradual release for the enriched language:

Theorem 2. *If $\mathcal{A}^0, \tilde{\Gamma}, pc \vdash c \Rightarrow \mathcal{A}'$ then c satisfies gradual release.*

6 Examples

This section provides examples of secure programming in the context of media distribution, mental poker, and bit commitment.

Media distribution In this example we consider the scenario of media distribution. Large media is distributed in encrypted form prior to the official release date. On the date of release, secret keys are supplied to the consumers.

This protocol can be implemented in our language as follows:

```
1  // local var declarations
2  key HK           k;
3  int H            media;
4  enc HK (int H) L outMedia;
5  key RK           outK;
6
7  // generating new key
8  k := newkey (HK);
9  media := ...;
10 // publishing the low value of the media
11 outMedia := encrypt_HK(k,media);
12 ...
13 release(k); // releasing the key
14 outK := k;
```

The media is distributed via variable `outMedia`, whose type `enc HK (int H) L` says that it stores low data which results from encrypting high data with high keys. At the media release time, the high key is released and published in the variable `outK`, whose type says that it stores released keys. This program is typable by the type system from Section 5 and, thus, is secure.

Note that premature key release is prevented by the type system. For example, if the lines

```
release(k);
outMedia := encrypt_HK(k,media);
```

are moved ahead of the encryption, then the program will be rejected as insecure.

Mental poker A pattern similar to media distribution is used in *mental poker* [35, 9, 4] protocols. Encryptions during the game correspond to card shuffling without trusted third party. At the end of the game, there is a verification phase to verify that no player was cheating. As with the fragment above, the type system guarantees that keys may not be released before the game phase is over.

Bit commitment using symmetric cryptography Bit commitment is a common building block in security protocols. We are primarily concerned with the confidentiality of the committed bit for our purposes. In a typical run between two principals, one of the principals commits its bid by encrypting a tuple consisting of the bid and a random value obtained from the other principal. The tuple is encrypted using a fresh key, which is later released in the revelation

phase. The following listing shows an example implementation of the protocol for the committing principal.

```
1  // local var declarations
2  int L rnd;
3  key HK k;
4  enc HK <int L, int S> L  outCommit;
5  int H  bit;
6  key RK outK;
7
8  // step 1: receive random value
9  rnd := ...;
10
11 k  := newkey(HK); // step 2: commit and send
12 outCommit := encrypt_HK(k, <rnd, bit>);
13 ...
14 release (k);    // step 3: release and send the key
15 outK := k;
```

The comments in the program connect the protocol steps to the code. This program, too, is typable by the type system from Section 5 and, thus, is secure.

To see how premature key release attacks are stopped, observe that the type system prevents from confusing steps 2 and 3 in the protocol implementation. The type system guarantees that no encryption with a released key may take place.

7 Related work

Much recent and ongoing work on language-based information security concerns policies for declassification. However, many policies tend to emphasize only some of the *what*, *who*, *where*, and *when* dimensions of declassification, leaving the other dimensions vulnerable for attacks [34]. As mentioned in Section 1, it is striking that these approaches (an overview of which can be found in [34]) fall into two mostly separate categories: revelation-based and encryption-based declassification.

Compared to other approaches to declassification, the distinctive features of gradual release include a transparent underlying semantic guarantee (the intuition of what is assured is clear), its scalability to rich policies (which include encryption and key release), and its practical enforcement.

The need for key release policies is motivated by a mental poker case study [4], where an important phase of the protocol is based on key revelation and verification that the participants were not cheating during the game.

Gradual release is inspired by work on *deducible information flow* [13] (which builds on earlier work [41, 25] on possibilistic security) that characterizes noninterference-like policies for abstract event systems in terms of what secret events can be deducible from public output events. Deducible information flow, however, has not been investigated in the context of information release (apart from simple partial flows) or language-based security.

A logic-based approach to representing attackers' knowledge about system events has been investigated in [19]. This approach is based on Sutherland's nondeducibility [36] Similarly to these approaches, we do not consider user strategies [40, 30] that can infer additional knowledge.

Our starting point in modeling information flow in the presence of encryption is *cryptographically-masked flows* [3]. In the present work, we recast cryptographically-masked flows in terms of small-step semantics, which is needed for sensitivity as to when keys are released.

Secrecy by typing [1] offers a type system for enforcing secrecy for a calculus that models cryptographic protocols. The assumption is that the attacker may not decrypt ciphertexts encrypted with secret keys. Key release and general declassification policies are, however, not considered.

Cryptographic types [15, 11] facilitate access-control enforcement in a distributed programming language. However, they provide no information-flow guarantees.

Noninterference modulo trusted functions [20] is based on the indistinguishability of program segments that are free of trusted cryptographic functions. In the spirit of *intransitive noninterference* [31], noninterference modulo trusted functions is a *where* declassification policy that does not provide guarantees for traces with declassification events.

Examples of *when* and *where* definitions that do not address cryptographic primitives are *noninterference "until"* [10], *intransitive downgrading* [27], *non-disclosure* [2], *flow locks* [8], and *WHERE* [26]. Noninterference "until" ignores the remainder of a trace once a declassification has happened. The other definitions rely on attacker models that are stronger than necessary [34]: wrapping release statements by downgrading commands [27, 26], flow declarations [2] or flow locks [8] would pose stronger requirements on security than gradual release because they demand security in the presence of state resetting (cf. Section 2).

Secrecy despite compromise [18] explores the problem of compromised principals within π-calculi, where secure channels can be modeled via shared secret keys, and compromised principals correspond to released keys. This work, however, only tracks a limited form of information flow: explicit flows.

Relative secrecy discusses release policies associated with particular primitives: releasing the result of matching a query to a secret password [38] and releasing the result of computing a one-way function [37]. The underlying guarantee is that the attacker cannot learn the secret in polynomial time in the size of the secret by running a program that satisfies relative secrecy.

A notable line of work [22, 21, 23] deals with computational guarantees for languages with statically distinguished keys. Computational attacker models have been also investigated in the context of π calculus [24, 28]. No general declassification policies are supported by these approaches.

Declassification policies based on intransitive noninterference have been explored for reactive systems [6]. A dis-

tinguishing feature of this work is the possibility of computational characterization of data declassification. While our goal (of having a practically enforceable security condition at the programming-language level) is initially different, we share the ultimate goal of considering computational adversaries with [6] (see the future work).

8 Conclusion

We have presented gradual release, a framework for unifying declassification, encryption, and key release policies.

Contributions The benefits of the framework include the following:

- *Policy-perimeter defense* The framework is the first to provide assurance for policies that include all of declassification, encryption, and key release;
- *Connection between revelation-based and encryption-based declassification* Not only does the framework combine revelation-based and encryption-based policies, but also formally connects the two kinds of declassification: gradual release for revelation-based declassification can be represented by a reassuringly simple encryption-based declassification: declassifying an expression corresponds to encrypting the expression with a fresh key and immediately releasing the key. In this light, the framework is the first to unify revelation-based and encryption-based declassification policies.
- *Conservativity* We have shown that gradual release for programs with no declassification or encryption is equivalent to a form of noninterference; in addition, we have shown that gradual release with no key release (but possibly with encryption) is equivalent to a form of possibilistic noninterference; and
- *Type-based enforcement* We have demonstrated that gradual release can be enforced by type and effect systems.

Future work While gradual release emphasizes the *where* dimension of information release it only offers a relative (to the previous point of release) assurance as to *what* data is released. As sketched in Section 2, it is possible to fully integrate the *what* dimension by connecting each release point to a policy that regulates what can be leaked.

The benefit of cryptographic and key-release primitives is truly realized in a distributed setting. A natural extension of our language is one with *actors* that run concurrently and interact with each other by sending and receiving messages on declared channels. While we can build on the message-passing enabled language [3] for which cryptographically-masked flows were introduced, we cannot directly reuse its security model because it considers actors in isolation. In presence of key release, it is not sufficient to view the actors independently because a key release by one actor may affect other actors. Thus, a goal for future work in this direction is to reason about whole-system security. In general, this involves reasoning about information flows due to blocking, scheduling, races, and other flows that may arise in concurrent systems (cf. [32]).

Although not in the scope of this paper, a high-priority direction for work on cryptographically-masked flows (with and without key release) is showing that *semantic security under chosen plaintext attack (SEM-CPA)* [17] (i.e., whatever is efficiently computable about the cleartext given the ciphertext, is also efficiently computable without the ciphertext) with an appropriate message authentication code (e.g., *INT-PTXT* [7]) for the underlying cryptographic primitives is sufficient for the semantic security of programs (with ample restrictions on key cycles) that satisfy the condition of cryptographically-masked flows.

Acknowledgments

We wish to thank Daniel Hedin and David Sands for helpful discussions on an earlier draft of this paper. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905, in part by VINNOVA, and in part by the Swedish Research Council.

References

[1] M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, Sept. 1999.

[2] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240, June 2005.

[3] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. In *Proc. Symp. on Static Analysis*, LNCS, pages 353–369. Springer-Verlag, Aug. 2006.

[4] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, pages 197–221. Springer-Verlag, Sept. 2005.

[5] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. Technical report, Chalmers University of Technology, 2007. Located at `http://www.cs.chalmers.se/~aaskarov/sp07full.pdf`.

[6] M. Backes and B. Pfitzmann. Intransitive non-interference for cryptographic purposes. In *Proc. IEEE Symp. on Security and Privacy*, pages 140–153, May 2003.

[7] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt 2000*, volume 1976 of *LNCS*, pages 531–545, Jan. 2000.

[8] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proc. European Symp. on Programming*, volume 3924 of *LNCS*, pages 180–196. Springer-Verlag, 2006.

[9] J. Castellà-Roca, J. Domingo-Ferrer, A. Riera, and J. Borrell. Practical mental poker without a TTP based on homomorphic encryption. In *Progress in Cryptology-Indocrypt*, volume 2904 of *LNCS*, pages 280–294. Springer-Verlag, Dec. 2003.

[10] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, Oct. 2004.

[11] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *Proc. IEEE Computer Security Foundations Workshop*, pages 170–186, 2003.

[12] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[13] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

[14] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, Aug. 1983.

[15] D. Duggan. Cryptographic types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 238–252, June 2002.

[16] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[17] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[18] A. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. CONCUR'05*, number 3653 in LNCS, pages 186–201. Springer-Verlag, Aug. 2005.

[19] J. Halpern and K. O'Neill. Secrecy in multi-agent systems. In *Proc. IEEE Computer Security Foundations Workshop*, pages 32–46, June 2002.

[20] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: High-level policy for a security-typed language. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, June 2006.

[21] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 77–91. Springer-Verlag, Apr. 2001.

[22] P. Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 159–173. Springer-Verlag, Apr. 2003.

[23] P. Laud and V. Vene. A type system for computationally secure information flow. In *Proc. Fundamentals of Computation Theory*, volume 3623 of *LNCS*, pages 365–377, Aug. 2005.

[24] P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 112–121, Nov. 1998.

[25] H. Mantel. Possibilistic definitions of security – An assembly kit –. In *Proc. IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.

[26] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, LNCS. Springer-Verlag, 2007.

[27] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, Nov. 2004.

[28] J. C. Mitchell. Probabilistic polynomial-time process calculus and security protocol analysis. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 23–29. Springer-Verlag, Apr. 2001.

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

[30] K. O'Neill, M. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201, July 2006.

[31] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

[32] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[33] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.

[34] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2007. To appear.

[35] A. Shamir, R. Rivest, and L. Adleman. Mental poker. *Mathematical Gardner*, pages 37–43, 1981.

[36] D. Sutherland. A model of information. In *Proc. National Computer Security Conference*, pages 175–183, Sept. 1986.

[37] D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.

[38] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, Jan. 2000.

[39] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[40] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 144–161, 1990.

[41] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *Proc. IEEE Symp. on Security and Privacy*, pages 94–102, May 1997.

[42] S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, Aug. 2004.