Dolev-Yao Information Flow

Simon Lund
DTU Compute
Kgs. Lyngby, Denmark
0009-0005-2957-3472

Sebastian Mödersheim
DTU Compute
Kgs. Lyngby, Denmark
0000-0002-6901-8319

***Abstract*—We propose a variant of classic information flow analysis that permits transmission of secrets over a public network, provided that secrets are suitably encrypted. In the style of Dolev and Yao, the intruder controls the network, observing all messages sent, but can only decrypt messages for which they know the decryption key, i.e., those keys which correspond to the security level of the intruder. In contrast to similar previous works we allow the intruder to send arbitrary bit strings as input to the program without any assumption that these inputs are in some sense well-typed. This means that cryptographic messages can enter program variables that were not meant to hold cryptographic messages and become part of computations and conditions. Despite this strong intruder model, we show that a program that satisfies our information-flow analysis also enjoys Dolev-Yao noninterference, a variant of standard noninterference where the intruder cannot break cryptography. The underlying model, which combines operating on actual bit strings with a symbolic intruder model, and the entire result are formalized and proved in Isabelle/HOL.**

Acknowledgments: Partly funded by EU Horizon Europe under Grant Agreement no. 101093006 (TaRDIS).

I. Introduction

A limitation of classical information flow analysis is that we cannot directly support distributed applications that communicate over an insecure network, even when using encryption to prevent leakage of messages. Simply put, encrypting a secret means that information is flowing from the secret into the encrypted message, and thus also the encryption must be classified as secret. This makes sense if the property we want to establish is noninterference: executing an information-flow-secure program in two states that only differ in secret variables leads to two states that also only differ in secret variables; thus an attacker who cannot see secret variables cannot distinguish these resulting states—no matter what computational resources they have. If we consider messages on the insecure network like being in public variables, then obviously no encryption can achieve this notion of noninterference.

For this reason, Askarov et al. [2] have introduced the concept of cryptographically-masked flows, a symbolic model of nondeterministic encryption and a special notion of possibilistic noninterference that essentially prevents the attacker from distinguishing two states that differ only in high-variables, even if they can observe encryptions of the high-variables. Laud [9] shows that a corresponding result can also be shown in the computational model.

While this is a strong result on what observations an intruder can make about the output of a program, there is a limitation of this approach in how *input* by the intruder to the program is handled. At first sight this is easy if we only consider confidentiality: one can allow the intruder to send arbitrary messages to the program, even encryptions where the intruder does not know the encryption key. Even if this influences secret variables, this does not help the intruder observe anything about them. However, with cryptography in messages we have opened Pandora's box, so to speak: at a position where the program expects basic data, the intruder may send an encrypted message, and vice-versa. This means that the program unintentionally works with cryptographic messages in arithmetic computations or conditions that were meant to work on basic data, or vice-versa attempts to decrypt basic data because it was expected to be an encrypted message. While most of these executions probably never lead to anything useful to the intruder, it is not obvious how to handle this formally. [2] therefore simply rules out from the model that the intruder can send such ill-typed messages.

A core contribution of this work is to define a model that does not need this restriction: in our model the intruder can send arbitrary bit strings to the program. This requires our model to go a rather different route than [2], so that ill-typed executions are semantically defined and that we can still achieve a noninterference result for programs that satisfy the information-flow analysis. As usual, this is a static analysis, checking that there are no explicit or implicit flows from higher to lower security labels. To account for the ill-typed intruder inputs, this includes type checking in a type system that formalizes the *intended* types of every program variable (basic data versus cryptographic messages where the content again has a particular intended type). A well-typed program in this type-system does not prevent an intruder sending ill-typed inputs, but it ensures that each message is used with a consistent type: if a piece of data is encrypted and sent, and later received and decrypted, then it will be interpreted with the same type. We demonstrate in Example VI.2 that an ill-typed program in this sense can leak information in the model where the intruder can send arbitrary bit strings as input. The approach of [2] is blind to this attack, because the intruder is restricted to sending messages that are (in our parlance) *well-typed*, while our model also allows the intruder to send any bit string. This is not a mistake of [2], but rather our model is simply more sensitive. It is in our opinion relevant, because an intruder can take messages out of context as our example demonstrates, and it is valuable to have a verification

method that covers also this kind of problem. We note that this is not an issue of symbolic versus computational models of cryptography: the intruder may not be able to decrypt the message that they use out of context. We also believe that the fix we suggest for this problem is reasonable: we define a type system that consistently distinguishes basic from cryptographic data; this does not type any program that may be vulnerable to the intruder using messages out of context. This type system also does not impose unrealistic burdens to the programmer; one should simply use formats to clearly indicate the meaning of a message.

In order to achieve a result in such a general model, we need to adopt a combination of the black-box model of cryptography and the arithmetic on actual bit strings. To that end, we use the concept of *crypto algebras* introduced by Mödersheim and Katsoris [13]: the universe is the set of all bit strings and all arithmetic operations are interpreted literally as what they do on given bit strings; the cryptographic operations are interpreted as *some* functions on bit strings that satisfy a number of conditions, namely that decryption of encryption obtains the original message. The space of all crypto algebras thus includes all implementations (reasonable or not) of the cryptographic algorithms that satisfy the functional properties of that algorithm. The idea is that the intruder cannot exploit any properties of a specific cryptographic implementation, but only properties that hold in every crypto algebra and thus also in the black-box model.

This is reflected in our notion of *Dolev-Yao noninterference* that we prove for any program c that satisfies our information-flow analysis: Consider two initial states S and T that only differ in variables that the intruder cannot see and any crypto algebra $\mathfrak{C}_1$. If executing c on S under $\mathfrak{C}_1$ results in a state S', then there *exists* a crypto algebra $\mathfrak{C}_2$ under which executing c on T results in a state T' such that the intruder cannot distinguish S' and T'. Here, distinguishing S' and T' means that the intruder can make an experiment (any sequence of operations on the variables that are accessible to them) that yields different bit strings in S' under $\mathfrak{C}_1$ and in T' under $\mathfrak{C}_2$.

The existential quantifier on $\mathfrak{C}_2$ here means that it is sufficient to find *any* crypto algebra such that from T, program c reaches a state T' that is indistinguishable from S'. This prevents the intruder from ever distinguishing the reached states S' and T' due to the concrete behavior of a given crypto algebra.

For instance, suppose a secret variable x has different values in S and T, and the program c encrypts x and sends the encrypted value, and suppose we would use the same crypto algebra $\mathfrak{C}_1$ for the successor states S' and T', then the encryption of x would necessarily be different in S' and T' (otherwise $\mathfrak{C}_1$ would violate the functional requirement that one can decrypt with the proper key). The principle of our model is that we show there is a crypto algebra $\mathfrak{C}_2$ such that the encrypted message in T' is the same bit string as the encrypted message in S' under $\mathfrak{C}_1$.

One may wonder what happens if the program c also encrypts another secret variable y and $x = y$ in S while $x \neq y$ in T. Under a naïve use of cryptography, the intruder could indeed see that the two encrypted messages are equal in S' and different in T'—and this would in fact be independent of the crypto algebra. We solve this using randomized encryption and our program semantics automatically chooses a new random value every time an encryption operator is applied, so that a program never produces the same bit string twice, even when encrypting the same data. Also in this case, our result shows that there is a $\mathfrak{C}_2$ that interprets the encrypted terms in T' as the same bit strings as $\mathfrak{C}_1$ in S'.

The fact that the concrete bit strings are identical in every two corresponding states (even though they represent encryption of different data) also makes the following case straightforward: if the program c receives an intruder-crafted message from the network, decrypts it and stores the result in a (public level) variable z, then c may treat z as an integer while it is actually part of an encrypted message. If c now contains a condition on z (e.g., whether it is even) then the truth value of the condition is the same in both considered executions.

This is, to our knowledge, the first model to achieve all the following properties at once:

- we model all arithmetic operations as what they truly are,
- we model without restriction that the intruder can interact with the program (using keys that are at or below the intruder's security level) using any available message as content (e.g. cryptographic messages where the program expects integers),
- and we still have a black-box model of cryptography, avoiding the intricacies of computational models with negligible probabilities and polynomial bounds on the steps of the intruder.

Our model and the proof of our main result (that a well-typed program satisfies Dolev-Yao noninterference) are quite involved. In pen-and-paper proofs, one might easily make a mistake without realizing it. Therefore, we have chosen to formalize the entire model and prove the result in the proof assistant Isabelle/HOL.

The formalization is over 7000 lines of code, and deeply embeds our model in the higher-order logic of Isabelle.
The Isabelle code may be found at the following URL:
https://www.imm.dtu.dk/~samo/DYIF-Isabelle.zip

The rest of this paper is organized as follows: Section II describes our term model and notion of crypto algebras. In Section III, we give our notion of DY noninterference. Then, in Section IV, we introduce our typing scheme for terms. Section V defines our program semantics and Section VI defines the typing scheme for programs. In Section VII, we formalize our intruder model. Section VIII contains our main result: that a program that types according to the scheme from Section VI will fulfill the DY noninterference of Section III. We finish with conclusions and related work in Section IX.

II. Term Model

A central piece of our model is manipulation of terms; the programs perform encryptions and computations on terms, and the intruder can construct terms to attempt to influence the programs. In this section, we briefly present the syntax and semantics of terms.

We build terms from a set of operators $\Sigma = \mathcal{B} \uplus \mathcal{E} \uplus \mathcal{F}$ where we distinguish between the following kinds:

- Non-cryptographic operators $\mathcal{B}$ like arithmetic operators. This includes the constants (as operators with 0 arguments) for all the basic data values.
- Encryption operators $\mathcal{E}$. (Decryption is handled separately.)
- Formatting operators $\mathcal{F}$. These are a specific combination of non-cryptographic operations to format messages before encryption, e.g., concatenation, tags, length fields.

Each $f \in \mathcal{B} \uplus \mathcal{E} \uplus \mathcal{F}$ has an associated arity, denoted by $arity(f)$. We call a term where the outermost operator is f an *f-term*. We write $\mathcal{T}(\Sigma, A)$ for terms built over Σ and a set A where in this paper A will be a set of variables or bit strings.

We let $\mathtt{B}$ denote the set of all bit strings, and, in certain contexts, we also use this set for constants in terms. In the following b_1, b_2 etc. will denote such bit strings. In this work we will consider Σ-algebras with $\mathtt{B}$ as carriers, i.e., for a Σ-algebra $\mathfrak{A}$, we have $\mathfrak{A}(f) : \mathtt{B}^n \to \mathtt{B}$ for every $f \in \Sigma$ of arity n. Given a set of variables $\mathcal{V}$ and an interpretation $\sigma : \mathcal{V} \to \mathtt{B} \cup \{undefined\}$, we define the interpretation $t^{\sigma,\mathfrak{A}}$ of a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ as follows:

$$t^{\sigma,\mathfrak{A}} = \begin{cases} \sigma(t) & \mid \text{if } t \in \mathcal{V} \\ \mathfrak{A}(f)({t_1}^{\sigma,\mathfrak{A}}, \ldots, {t_n}^{\sigma,\mathfrak{A}}) & \mid \text{if } t = f(t_1, \ldots, t_n) \\ & \quad \text{and } ({t_1}^{\sigma,\mathfrak{A}}, \ldots, {t_n}^{\sigma,\mathfrak{A}}) \in \mathtt{B}^n \\ undefined & \mid \text{otherwise} \end{cases}$$

We also define $t^{\mathfrak{A}} = t^{(\lambda x.\ undefined),\mathfrak{A}}$, which we exclusively use when t does not contain variables.

Let the *base algebra* $\mathfrak{B}$ be a $\mathcal{B} \uplus \mathcal{F}$-algebra with the property that $\mathfrak{B}(f)$ is injective for every $f \in \mathcal{F}$. This fixes the interpretation of the non-cryptographic operations and formats to their real-world implementation, e.g., integer addition. Our result is independent of the precise behavior of these functions, and we only require that formats are injective so that a given bit string can be parsed for a given format $f \in \mathcal{F}$ in at most one way, and thus formats are unambiguous to parse.

We now define the concept of a crypto algebra, which interprets the remaining function symbols in $\mathcal{E}$ that the base algebra does not cover.

Definition II.1. *A* crypto algebra $\mathfrak{C}$ *is a Σ-algebra that has the fixed base algebra $\mathfrak{B}$ as a subalgebra and such that $\mathfrak{C}(f)$ is injective for every function $f \in \mathcal{E}$, i.e., from $\mathfrak{C}(f)(b_1, \ldots, b_n) = \mathfrak{C}(f)(b'_1, \ldots, b'_n)$ follows $b_i = b'_i$ for every $i \in \{1, \ldots, n\}$. We associate with a crypto algebra a random-number generator $\mathfrak{C}_R : (\mathcal{E} \times \mathbb{N}) \to \mathtt{B}$.*

Each encryption function in $\mathcal{E}$ models encryption with a fixed key, and we will also often write $e_k \in \mathcal{E}$ to make explicit the encryption key k. This key k can be either a public key for asymmetric encryption or a shared secret key for symmetric encryption. With each key k we associate a security level, and we model an intruder who can only decrypt e_k-messages where k is at or below the intruder's security level. Note that in case of asymmetric encryption, the intruder may thus be able to encrypt with e_k, but not decrypt e_k-messages if k is at a higher security level.

Our only requirement for encryption functions is that they are injective. This requirement is of course necessary, since otherwise one could not define decryption in a reasonable way. However, it may be surprising that we have no security requirements on the functions. This is because we will not make any computational arguments in this work. Rather, we model an intruder who can only use the cryptography as a black box, in the style of the Dolev-Yao model, i.e., encrypt and decrypt with known keys. The crypto algebras are the key idea to integrate Dolev-Yao with a bit string model. To see that, consider the question of what operations the intruder can perform on an encrypted term that give the same result in *every* crypto algebra. The answer is: only decryption with the correct key and the comparison of two encrypted messages with the same key; $e_k(b_1, \ldots, b_n) = e_k(b'_1, \ldots, b'_n)$ holds iff $b_i = b'_i$ for all $i \in \{1, \ldots, n\}$, independent of the crypto algebra. This reflects two problems of non-randomized encryption: if the intruder knows that two messages are encrypted with the same key, even not knowing the key, the comparison reveals if their content is that same; moreover, for public key encryption the intruder can also attempt to guess the content of a message, encrypt with the public key and compare to the observed message, which gives the same result iff the guess was correct. While the general model of Dolev-Yao noninterference in the next section allows for non-randomized encryption, our main result uses only randomized encryption: the first argument of an encryption will be a fresh random value in every encryption that the program produces, thus preventing guessing attacks and comparison attacks by construction. Thus, the intruder can only learn anything about the content of an encryption, if they know the decryption key. As a preparation for that construction, our notion of crypto algebra includes a random number generator $\mathfrak{C}_R$, which given an encryption symbol and sequence number provides a unique bit string for randomization.

Our model of cryptographic operators with fixed keys limits our approach to a fixed infrastructure of long-term keys, and we thus cannot model, for instance, programs where group keys are changed whenever group membership changes.[1] Protocols that derive fresh session keys from long-term keys and then use the session keys to transmit payload messages seem to offer at least as good security guarantees as encryption

[1] Such a change of group keys is desirable to ensure that group members only have access to group messages that were sent while being part of the group; however this change does not give strong information-flow control guarantees anyway, e.g., a message while a is a member may depend on a message before a was a member.

with the long-term keys, but this requires also reasoning about authentication (not only secrecy) and we leave a formal treatment of this general link to protocol verification to future work.

Example II.1. *We consider $b_1, b_2 \in \mathtt{B}$, $f \in \mathcal{F}$ and $e_k \in \mathcal{E}$. We set $t = f(b_1, b_2)$.*

Let $\mathfrak{C}$ and $\mathfrak{C}'$ be crypto algebras. Then $t^{\mathfrak{C}} = t^{\mathfrak{C}'} = t^{\mathfrak{B}}$ because t contains no encryption operator, while in general $e_k(r,t)^{\mathfrak{C}} \neq e_k(r,t)^{\mathfrak{C}'}$, even for the same randomization $r \in \mathtt{B}$.

During execution, a program will use different random values for the encryption, e.g., $b_3 = \mathfrak{C}_R(e_k, 0)$ and $b_4 = \mathfrak{C}_R(e_k, 1)$ for the first two encryptions with k, and since $b_3 \neq b_4$ by construction, also $\mathfrak{C}(e_k)(b_3, t^{\mathfrak{C}}) \neq \mathfrak{C}(e_k)(b_4, t^{\mathfrak{C}})$, leaving the intruder unable to tell whether the encryptions contain the same payload (in any crypto algebra). Note that $\mathfrak{C}_R$ itself is not an operator of the crypto algebra, and the programmer will simply write $x = e_k(t)$ whenever a new encryption should be executed, and it is part of the program semantics to use the random number generator at this point in the execution.

III. DY NONINTERFERENCE

The basis of the standard notion of noninterference is that an intruder cannot distinguish two program states that are equal on all program variables that are at or below the intruder's security level. One then shows for a program c that satisfies information flow: running c on arbitrary indistinguishable states S and T leads to states S' and T', respectively, that are also indistinguishable. Extending these concepts with cryptographic messages that programs can send and receive on a public channel presents the challenge that cryptographic messages can "creep" into the normal program variables (that were not supposed to hold cryptographic messages) as the following example illustrates:

Example III.1. *Let x and z be secret, i and y be public. Recall that all encryptions are implicitly randomized, so every sent message in this program is distinct from the previous ones, even if the content is the same. Let further e_k be symmetric encryption and e_{pk} a public-key encryption that are both at level secret, and the intruder is at level public (and thus can only encrypt with e_{pk}).*

```
i ← 15
while(i > 0) {
   i ← i − 1
   try_rcv(e_pk(y))
   if(x ≥ y) {
      z ← x
   } {
      z ← y
   }
   send(e_k(z))
}
```

All instructions are standard, except `try_rcv`$(e_{pk}(y))$ *which means that the program receives a message and tries to decrypt it with the private key corresponding to public key pk. If that decryption fails, the program ignores the incoming message and continues (we discuss error handling a bit later), otherwise y gets bound to the content of the encrypted message. The program compares x and y and encrypts the larger one with e_k and sends it.*

An example of how the intruder can interact with this program is as follows: They pick an initial value y_0 and send $e_{pk}(y_0)$. The program will compare $x_0 \geq y_0$ (where x_0 is the initial value of x) and either send $t = e_k(x_0)$ or $t = e_k(y_0)$. The intruder cannot decrypt t or tell whether it contains x_0 or y_0 (since the encryption is randomized), but they can produce $e_{pk}(t)$. This term will actually be received by the program, taking $y_1 = t$ as the new value of y. The comparison now depends on the concrete crypto algebra, i.e., whether the bit string corresponding to t, interpreted as an integer, is smaller than x_0. Either way, the program will answer with $e_k(x_0)$ or $e_k(t)$.

This example shows that the execution of the program may depend on the crypto algebra, also in a way that is not transparent to the intruder (in the example, the intruder does not know whether $x_0 \geq t$). To our knowledge, all previous works that consider information flow with cryptography in a computational or probabilistic model like [2, 9] limit the interaction with the intruder so much that cryptographic messages cannot influence normal variables and thus the control flow or information flow of a program as in the previous example. In fact, one of our contributions is to define a semantics that fully allows this intruder interaction; at the same time we keep the spirit of Dolev and Yao: our model reflects that the cryptography is a black box to the intruder.

We now define Dolev-Yao noninterference (DYNI) parameterized over several basic notions like program states, program execution, and intruder recipes. There are no constraints on these notions, we only give an informal idea of how we intend to use them, so for now this is only a generic framework. In the following sections, we will instantiate these with a concrete notion of program states and so on, and prove DYNI for programs that satisfy the information flow analysis. The first reason for this framework is the ease of presentation: we can give an outline of the approach without immediately diving into the technical details. The second reason is generality. Suppose for instance one wants to augment the approach with a notion of timing (that we do not consider in this paper); this would require integrating time into all the parameters of the framework, but the notion of DYNI would not necessarily be changed. The notions of our DYNI framework are as follows:

1) A notion $\mathbf{S}$ of (program) states; as compared to classic notions, states will also contain encrypted messages sent by the program;
2) A notion $\sim_{classic}$ of classic indistinguishability between states. We do not make any formal requirements to this relation (see however Definition III.1). The typical use is that $S \sim_{classic} T$ iff everything visible to the intruder is identical in S and T, e.g. program variables at and below the intruder's security level, and the encrypted

messages sent by the program. The difference to Dolev-Yao indistinguishability is that $\sim_{classic}$ does not consider encryption and decryption operations the intruder may perform and it is also independent of an interpretation of cryptography in a crypto algebra.

3) A notion $init(S)$ of initial state, where in particular no encryptions have been performed yet;
4) A notion of executing program c in state S under crypto algebra $\mathfrak{C}$, yielding a state S', written $(S, c) \underset{\mathfrak{C}}{\rightsquigarrow} S'$.
5) A notion $\mathbf{R}$ of recipes, i.e., operations and evaluations the intruder can perform in a state; this could include the intruder encrypting or decrypting messages with available keys; and
6) A notion of evaluating a recipe $\mathbf{r} \in \mathbf{R}$ on a state S under crypto algebra $\mathfrak{C}$, denoted $S(\mathbf{r})^{\mathfrak{C}}$ which yields a bit string.

Definition III.1. *We say two states S and T are* DY indistinguishable (with regard to crypto algebras $\mathfrak{C}$ and $\mathfrak{C}'$)*, and write $S \ {}^{\mathfrak{C}}\!\sim^{\mathfrak{C}'}_{DY} T$, if for any $\mathbf{r} \in \mathbf{R}$ we have $S(\mathbf{r})^{\mathfrak{C}} = T(\mathbf{r})^{\mathfrak{C}'}$.*

We call $\sim_{DY}$ well-formed with regard to the classic indistinguishability $\sim_{classic}$ *if ${}^{\mathfrak{C}}\!\sim^{\mathfrak{C}'}_{DY} \subseteq \sim_{classic}$ for all $\mathfrak{C}, \mathfrak{C}'$.*

A program c satisfies DY noninterference*, if for all S and T with $init(S)$, $init(T)$, and $S \sim_{classic} T$, and all $\mathfrak{C}$ and S' with $(S, c) \underset{\mathfrak{C}}{\rightsquigarrow} S'$, there exist $\mathfrak{C}'$ and T' such that $(T, c) \underset{\mathfrak{C}'}{\rightsquigarrow} T'$ and $S' \ {}^{\mathfrak{C}}\!\sim^{\mathfrak{C}'}_{DY} T'$.*

DY indistinguishability means that no recipe $\mathbf{r} \in \mathbf{R}$ allows the intruder to distinguish the given states. Note that we allow different algebras $\mathfrak{C}$ and $\mathfrak{C}'$ to interpret the states in: this allows us to interpret every encrypted message as exactly the same bit string in both states, even if their content differs.

The well-formedness means that $\sim_{DY}$ is a refinement of the classical indistinguishability; a model in which this is violated would be unreasonable (giving the intruder the ability to look at cryptographic messages should only improve their ability to distinguish states).

That a program c satisfies Dolev-Yao noninterference thus means: given any two initial program states S and T (i.e., where no cryptographic operations have been performed) that are classically indistinguishable (i.e., they agree on all program variables that the intruder can see) and given any crypto algebra $\mathfrak{C}$, if the execution of c leads from state S to state S' in $\mathfrak{C}$, then we can find a crypto algebra $\mathfrak{C}'$ under which executing c leads from T to a state T' and T' is DY-indistinguishable from S'. Given well-formedness, S' and T' are also classically indistinguishable. The fact that $S' \ {}^{\mathfrak{C}}\!\sim^{\mathfrak{C}'}_{DY} T'$ is a witness that there is no way to distinguish S' and T' that is *independent of the crypto algebra*. Intuitively, this is the idea to model the cryptography as a black-box: the intruder can only exploit properties that hold true in *any* crypto algebra, but cannot make cryptanalysis on a specific cryptographic implementation. This also sidesteps all computational arguments that would require limiting the intruder to polynomial time and negligible probabilities to guess keys and random values.

Example III.2. *Continuing Example III.1, let us for instance consider a state S where $x = 42$ and a state T where $x = 52$. The intruder strategy (what messages to send to the program) must be the same in both execution from S and T; in our notion of states, it will be fixed initially.*

Suppose, starting from S in $\mathfrak{C}$, the intruder sends two guesses $b_1 = e_{pk}(r_1, 25)^{\mathfrak{C}}$ and $b_2 = e_{pk}(r_2, 50)^{\mathfrak{C}}$ to the program and obtains the answers $b'_1 = e_k(r'_1, 42)^{\mathfrak{C}}$ and $b'_2 = e_k(r'_2, 50)^{\mathfrak{C}}$. For clarity, we have made explicit the randomization. Now we can find a crypto algebra $\mathfrak{C}'$ such that we have a corresponding execution in T: $b_1 = e_{pk}(r_1, 25)^{\mathfrak{C}'}$, $b'_1 = e_k(r'_1, 52)^{\mathfrak{C}'}$, $b_2 = e_{pk}(r_2, 50)^{\mathfrak{C}'}$, $b'_2 = e_k(r'_2, 52)^{\mathfrak{C}'}$. In particular, in the T-execution the two messages from the program have the same content, but due to randomization these are still different bit strings $b'_1 \neq b'_2$, like in the S-execution. Thus, the resulting states are indistinguishable.

In contrast, without the randomization, we would have necessarily two identical answers $e_k(52)^{\mathfrak{C}'}$ in the T-execution, while $e_k(52)$ and $e_k(50)$ in the S-execution would necessarily be different (as $\mathfrak{C}(e_k)$ must be injective). Thus, without randomization the intruder can violate secrecy and find out that the initial value of x must be between 25 and 50 in S', respectively larger than 50 in T'. [2]

IV. Term Typing and Labeling

Central to our information flow analysis are two kinds of typing judgment relations. For distinction, we call the first one *typing* as it is concerned with data types, and the second one *labeling* as it is concerned with security labels.

As is standard in information flow, *labeling* is done with regard to a complete lattice $(\mathcal{L}, <, \bigsqcup)$, where $\mathcal{L}$ is a set of *security labels*, partially ordered by $<$, so that every subset of $\mathcal{L}$ has a supremum $\bigsqcup$ (and we denote with $\bigsqcap$ the infimum). The simplest example is $\{\bot, \top\}$ with $\bot < \top$, where $\bot$ represents public information, and $\top$ secret information. In general, the security policy specified by the labeling is that information at label l_1 may be read only by an actor with security level l_2 iff $l_1 \leq l_2$.

The *typing* distinguishes the information that classical information flow is concerned with, which we call *basic*, from messages where encryption and format operators have been applied. The intruder may well send an encrypted message in a position where a program expects a payload/basic message; the types thus specify only the *intended* type of a variable. This is important for the noninterference result, however, since a *well-typed* program will for, instance, never try to decrypt a message that was intended to be of type basic, and thus the intruder will never benefit from ill-typed messages.

Example IV.1. *We consider a service where the key k is used to encrypt medium confidentiality information to a large number of participants. Suppose further that top-secret information*

[2] This also illustrates that DY noninterference is not susceptible to *occlusion*, as defined by Askarov, Hedin, and Sabelfeld [2]. They here refer to the phenomenon where the simplest way of changing the definition of noninterference to accept encryptions, by considering all ciphertexts equal, will mask unintended implicit flows, like the one we get if we omit randomization from the example above.

meant only for a handful of privileged users is stored in the variable x. We would then specify the typing and labeling environments so that the term $e_k(x)$ is ill-labeled, to avoid accidentally sending the top-secret information in a message most participants can decrypt.

Example IV.2. *Consider a program that acts as a server that can receive, process, and reply to two kinds of requests. For the first kind, the answer is a message of the form $e_k(b_{id}, b_s)$, where b_{id} is some identifier and b_s is a secret. The second request is a message of the form $e_k(b_{id}, b_p)$ where b_p is some public information, and the server then publishes b_p. Because there of the similarity of these messages, an intruder can take any message $e_k(b_{id}, b_s)$ and send it to the server as the second kind of request and thus get the secret b_s published. To avoid this problem, one should use formats to distinguish the meaning of the two messages, so that it is clear which fields are public and which are secret.*

More precisely, we consider a distinct type $\bullet$, called *basic*, for all variables that are supposed to contain only basic data (not composed with encryption or formats). A well-typed program applies non-cryptographic operators from $\mathcal{B}$ only to values of type $\bullet$ and the result is again of type $\bullet$.

For every encryption and format operator we fix the intended types of their arguments; e.g., we cannot use the same encryption operator for two different types of messages. We demonstrate in Appendix A that this is usually not a restriction, since using formats one can structure different kinds of information in an unambiguous way before encrypting it, and it is good engineering practice not to encrypt raw data in the first place. As a benefit, the type system is rather simple because: if the root symbol of a term is a function symbol from $\mathcal{E} \uplus \mathcal{F}$, then this determines for each subterm what type it must have. We can thus identify the type of such a term with its top-level function symbol and define the set of types to be $T = \mathcal{E} \uplus \mathcal{F} \uplus \{\bullet\}$. Let the types and labels of all encryption and format operators be specified in the following two environments:

Definition IV.1. *We consider a typing environment Δ and labeling environment Γ, where for every function $f \in \mathcal{E} \uplus \mathcal{F}$ of arity n, there is specified $\Delta(f) = ((\tau_1, \ldots, \tau_n) \to f)$ for some types $\tau_1, \ldots, \tau_n \in T$, and $\Gamma(f) = ((l_1, \ldots, l_n) \to l)$ for some labels $l_1, \ldots, l_n, l$ from the security lattice $\mathcal{L}$. For every $f \in \mathcal{B}$ we set $\Delta(f) = ((\bullet, \ldots, \bullet) \to \bullet)$. We require that Δ is* well-founded *in the following sense: there is no infinite chain $f_1, f_2, \ldots \in \mathcal{E} \uplus \mathcal{F}$ such that, for all i, $\Delta(f) = ((\tau_1, \ldots, \tau_n) \to f_i)$ and f_{i+1} is one of the τ_j.*

The typing environment Δ and labeling environment Γ must be well-formed *in the following sense: for any $f \in \mathcal{E} \uplus \mathcal{F}$, $\tau_1, \ldots, \tau_n \in T$, and $l_1, \ldots, l_n, l \in \mathcal{L}$ such that $\Delta(f) = ((\tau_1, \ldots, \tau_n) \to f)$ and $\Gamma(f) = ((l_1, \ldots, l_n) \to l)$ the following holds:*

(1) For any $1 \le i \le n$ such that $\tau_i \neq \bullet$ and $\Gamma(\tau_i) = ((l'_1, \ldots, l'_m) \to l')$ we have $l' \le l_i$.

(2) If $f \in \mathcal{F}$ then $l \ge \bigsqcup\{l_1, \ldots, l_n\}$.

$$(1) \quad \frac{}{\mathcal{X} \vdash t : \Delta(t)}\ t \in \mathcal{X} \cup \mathtt{B}$$

$$(2) \quad \frac{\Delta(f) = ((\tau_1, \ldots, \tau_n) \to \tau) \quad \mathcal{X} \vdash t_1 : \tau_1 \ \ldots\ \mathcal{X} \vdash t_n : \tau_n}{\mathcal{X} \vdash f(t_1, \ldots, t_n) : \tau}$$

Fig. 1. Syntactic typing of terms in $\mathcal{T}(\mathcal{E} \uplus \mathcal{F} \uplus \mathcal{B}, \mathtt{B} \uplus \mathcal{V})$.

(3) If $f \in \mathcal{E}$ and $n > 0$ then $\tau_1 = \bullet$ and $l_1 = \bigsqcup\{l_2, \ldots, l_n\}$.

Additionally, we use the typing and labeling environments for variables. For each variable $x \in \mathcal{V}$ we associate a type $\Delta(x) = \tau \in T$ and label $\Gamma(x) = l \in \mathcal{L}$. Finally, for every bit string $b \in \mathtt{B}$ we set $\Delta(b) = \bullet$. We require the following additional well-formedness condition:

(4) for any $x \in \mathcal{V}$ such that $\Delta(x) \neq \bullet$ with $\Gamma(\Delta(x)) = ((l_1, \ldots, l_n) \to l)$ we have $l \le \Gamma(x)$.

$\Delta(f) = ((\tau_1, \ldots, \tau_n) \to \tau)$ and $\Gamma(f) = ((l_1, \ldots, l_n) \to l)$ thus specifies that f can only be applied to arguments $t_1, \ldots, t_n$ such that t_i has type τ_i and a security level $l'_i \le l_i$. Moreover, the result will be of type τ and security level l. For $f \in \mathcal{B}$ we do not define $\Gamma(f)$; rather, as is standard in information flow, the label of $f(t_1, \ldots, t_n)$ is the supremum of the labels of $t_1, \ldots, t_n$.

Requirement (1) ensures that a well-typed ground term will also be well-labeled. For example, if $\Delta(f) = ((g) \to f)$ and $\Gamma(f) = ((\bot) \to l)$, then we cannot have $\Gamma(g) = ((l'_1, \ldots, l'_m) \to \top)$. Requirement (2) reflects that formats do not protect their inputs; the output of a format must be at least as secret as each of its inputs. Requirement (3) ensures that the first input of each encryption can be used for the randomization. This value should be a bit string, and it must be at least as secret as all the other inputs. Requirement (4) represents that any variable is labeled such that it is permitted to contain values of the type it is typed as.

We note that, in general, labeling depends on the typing environment, but typing is independent of labeling. For the rest of the paper, we assume we are given a fixed well-formed Δ and Γ.

Example IV.3. *For the term $e(f(x), add(y, z))$ to be well-typed, where $e \in \mathcal{E}$, $f \in \mathcal{F}$, and $add \in \mathcal{B}$, we must have $\Delta(e) = ((f, \bullet) \to e)$, $\Delta(f) = ((\Delta(x)) \to f)$, $\Delta(y) = \bullet$ and $\Delta(z) = \bullet$, where any $\Delta(x)$ except e or f would work.*

If $\Gamma(e) = ((l_{e1}, l_{e2}) \to l_e)$ and $\Gamma(f) = ((l_{f1}) \to l_f)$ then we must have $\Gamma(x) \le l_{f1}$, $l_f \le l_{e1}$, and $\Gamma(y) \sqcup \Gamma(z) \le l_{e2}$ for the above term to be well-labeled.

Fig. 1 defines the *syntactic typing* relation we use for terms. As a preparation for later, we have as a parameter to the typing relation a set $\mathcal{X}$ of variables. Later, $\mathcal{X}$ will be the set of initialized variables and thus the typing can include the check that only initialized variables are used; if this is not needed, we simply set $\mathcal{X} = \mathcal{V}$. We use $\vdash t : \tau$ as shorthand for $\emptyset \vdash t : \tau$ to type ground terms. Rule (1) describes how we type variables and bit strings and rule (2) describes how we type

function applications. When applied to typing and labeling environments as defined in Definition IV.1, a bit string can only be typed as $\bullet$. The semantic typing below will also allow that bit strings can be interpreted as cryptographic messages.

In Section VI, we use the syntactic typing relation to type program expressions, which can contain bit strings only as constants in basic expressions (using operators from $\mathcal{B}$). Later, we also need a typing for arbitrary bit strings that can represent cryptographic messages or formats, e.g., messages received from the network. This typing thus depends on the considered crypto algebra and hence we call it *semantic* typing. Note that in general a bit string can be typed in more than one way, e.g., it could be an encrypted message, but also be interpreted as basic. We thus define semantics typing as a relation between a crypto algebra, a bit string, and a type we want to interpret it as:

Definition IV.2. *We say that the bit string b* can be semantically typed as *τ under crypto algebra $\mathfrak{C}$, written $\mathfrak{C} \vdash b : \tau$, if and only if there exists a term $t \in \mathcal{T}(\mathcal{E} \uplus \mathcal{F}, \mathtt{B})$ such that $t^{\mathfrak{C}} = b$ and $\vdash t : \tau$.*

Example IV.4. *Consider $f, g \in \mathcal{E} \uplus \mathcal{F}$ with $\Delta(f) = ((g, \bullet) \to f)$ and $\Delta(g) = ((\bullet, \bullet) \to g)$. For $b_1, b_2, b_3 \in \mathtt{B}$ we then have $\vdash f(g(b_1, b_2), b_3) : f$. For $b \in \mathtt{B}$ and a crypto algebra $\mathfrak{C}$ such that $f(g(b_1, b_2), b_3)^{\mathfrak{C}} = b$, we have $\mathfrak{C} \vdash b : f$.*

Thus, to type an arbitrary bit string (w.r.t. $\mathfrak{C}$) we ask for the type of any (well-typed) term that can produce that bit string in $\mathfrak{C}$. While in general there are several terms that can produce a given bit string in $\mathfrak{C}$, there is at most one from $\mathcal{T}(\mathcal{E} \uplus \mathcal{F}, \mathtt{B})$ of a given type τ:

Lemma IV.3. *For a crypto algebra, $\mathfrak{C}$, bit string b, and type, τ, there is at most one ground term, $t \in \mathcal{T}(\mathcal{E} \uplus \mathcal{F}, \mathtt{B})$, such that $t^{\mathfrak{C}} = b$ and $\vdash t : \tau$.*

As we make no assumption about there being no collisions between ciphertexts encrypted with different encryption symbols, there may of course be different τ_1 and τ_2 such that $\mathfrak{C} \vdash b : \tau_1$ and $\mathfrak{C} \vdash b : \tau_2$ for any given bit string b. In fact, we know that in addition to any other type b might type as, we will have $\mathfrak{C} \vdash b : \bullet$.

Later, we will need to treat bit strings as terms, for example when they are received from the network to be decrypted and parsed. We give a convenient way to refer to the unique representation we decode bit strings to.

Definition IV.4. *If $\mathfrak{C} \vdash b : \tau$ then by Lemma IV.3 there exists a unique term $t \in \mathcal{T}(\mathcal{E} \uplus \mathcal{F}, \mathtt{B})$ such that $t^{\mathfrak{C}} = b$ and $\vdash t : \tau$. We denote this term as $b{\downarrow}^{\mathfrak{C}}_{\tau}$.*

Similar to well-typedness, we can now define that a term is *well-labeled* by a labeling judgment $\Vdash t : l$ as defined in Fig. 2. For terms without encryption and format operators, it is just the supremum of the contained variables. For formats recall that the result label as at least as high as all the argument labels. The specialty of our work is thus the encryption operator as the only operator that can have a lower security label than its arguments.

$$(1) \quad \frac{}{\Vdash x : \Gamma(x)}\ x \in \mathcal{V}$$

$$(2) \quad \frac{\Vdash t_1 : l_1 \quad \ldots \quad \Vdash t_n : l_n}{\Vdash f(t_1, \ldots, t_n) : l_1 \sqcup \ldots \sqcup l_n}\ f \in \mathcal{B}$$

$$(3) \quad \frac{\Gamma(f) = ((l_1, \ldots, l_n) \to l) \qquad \Vdash t_1 : l_1 \ \ldots \ \Vdash t_n : l_n}{\Vdash f(t_1, \ldots, t_n) : l}\ f \in \mathcal{F} \uplus \mathcal{E}$$

Fig. 2. Well-labeling of terms in $\mathcal{T}(\mathcal{B} \uplus \mathcal{E} \uplus \mathcal{F}, \mathcal{V})$

Example IV.5. *We set $\Delta(x) = \bullet$, $\Delta(y) = \bullet$, $\Gamma(x) = \top$, $\Gamma(y) = \bot$, $\Delta(f) = ((\bullet, \bullet) \to f)$, and $\Gamma(f) = ((\top, \bot) \to \top)$.*

We now have that $f(x, y)$ is well-typed and well-labeled. If we switch around the variables to $f(y, x)$, we get a term that is still well-typed but not well-labeled, since the secret x flows into the public second position of f.

V. Program Semantics

A. Syntax and Program States

Our programming language contains constructs for the assignment of expressions (without encryptions) to variables, encryption, sending to the network, receiving from the network, conditional branching, and looping. They are explained in more detail along with the semantics below. The syntax of our language is given by the following grammar:

$$\begin{array}{rcl} c & ::= & x \leftarrow t \mid x \leftarrow e(y) \mid \mathtt{send}(x) \mid \mathtt{try\_rcv}(t_{rcv})\{c'\} \\ & \mid & c_1; c_2 \mid \mathtt{if}(p)\{c_1\}\{c_2\} \mid \mathtt{while}^m(p)\{c'\} \\ p & ::= & \mathtt{ff} \mid p_1 \longrightarrow p_2 \mid t_a = t_b \end{array}$$

Here, x and y are variables, t is a *program expression* from $\mathcal{T}(\mathcal{F} \uplus \mathcal{B}, \mathcal{V})$, e is an encryption operator from $\mathcal{E}$, t_{rcv} is a *receive pattern* from $\mathcal{T}(\mathcal{E} \uplus \mathcal{F}, \{x \mid x \in \mathcal{V} \wedge \Delta(x) = \bullet\})$, and p is a formula built from $\mathtt{ff}$ (falsity), $\longrightarrow$ (logical implication), and $=$ (equality of terms). The other common logical operators, like $\wedge$, are syntactic sugar, and logical predicates can be expressed as Boolean-valued functions in $\mathcal{B}$. Finally, m is a *loop bound* that can either be a program expression or a bit string.

Decryption is done implicitly as part of the receive patterns as explained below. Encryption can only happen as an isolated step; this is only for convenience of the development, and obviously programs with arbitrary use of encryption can be rewritten into this form.

A *program state* is a 4-tuple (σ, ib, ob, C_R). The *internal program state* is $\sigma : \mathcal{V} \to \mathtt{B} \cup \{undefined\}$, representing the value contained in each variable. The *inbox* of the program is ib, an infinite stream of bit strings, representing the messages that will be received by the program during execution. The *outbox* of the program is ob, a list of bit strings, representing messages sent to the network. From the point of view of the intruder, the inbox contains the messages sent to the program and the outbox contains the messages received from

the program. $C_R : \mathcal{E} \to \mathbb{N}$ is a *counter* for each encryption symbol; the program will use $\mathfrak{C}_R(e, C_R(e))$ as the random value when an encryption with $e \in \mathcal{E}$ is performed and increase the counter.

The fact that we allow for arbitrary bit strings in the inbox of a program subsumes anything an intruder strategy can ever achieve. Note that this is in contrast to other Dolev-Yao-style models where the intruder chooses a recipe that determines the message the program receives. Using the over-approximation of arbitrary bit strings simplifies our arguments, but also prevents the application to integrity goals; we will get back to this question in the conclusions.

B. Initial States and Classical Noninterference

We call a state *an initial state*, written $init(\sigma, ib, ob, C_R)$, iff $ob = []$, $C_R = (\lambda e.\ 0)$, and for all $x \in \mathcal{V}$ such that $\delta(x) \neq \bullet$ we have $\sigma(x) = undefined$. Thus, in an initial state the outbox is empty and cryptographic operations have not yet been performed.

In the following let $l_{ref} \in \mathcal{L}$ be the *intruder level*, i.e., the intruder is allowed to see information labeled $l \leq l_{ref}$. As is standard in information flow, the main result will show that, in a program that satisfies our information flow policy, an intruder with security level l_{ref} will only be able to see information at level $l \leq l_{ref}$ and will not learn anything about information with any other labels. This result will hold for any $l_{ref} \in \mathcal{L}$, i.e., no matter which security level the intruder has, they cannot learn anything about information they are not cleared for. For convenience of presentation, let thus in the following l_{ref} be an arbitrary security level. We call two states S and T *classically indistinguishable (wrt. l_{ref})*, written $S \sim_{classic} T$, if they agree in all variables $l \leq l_{ref}$ and have equal inboxes and outboxes:
$(\sigma_S, ib, ob, C_{RS}) \sim_{classic} (\sigma_T, jb, ub, C_{RT})$ iff $ib = jb$, $ob = ub$, and $\sigma_S(x) = \sigma_T(x)$ for all $x \in \mathcal{V}$ with $\Gamma(x) \leq l_{ref}$. Thus, indistinguishable initial states only differ in $\bullet$-type variables that the intruder cannot see.

C. Big-Step Semantics

Fig. 3 gives the semantics of our programming language and we discuss them here in the presented order. All these rules are evaluated in a given crypto algebra $\mathfrak{C}$. First, in an assignment $x \leftarrow t$, we evaluate t in $\mathfrak{C}$ and update the program state for x. The `send`-statement will simply put the bit string contained in the variable into the outbox. For an encryption, $x \leftarrow e(y)$, we use the random value number $C_R(e)$ from random number generator $\mathfrak{C}_R$ to perform the encryption of the bit string $\sigma(y)$; the result is stored in variable x. We then increment $C_R(e)$ so a new random value will be used for the next encryption with e. Sequential composition and conditional branching are straightforward, where $\sigma \models_{\mathfrak{C}} p$ denotes that the predicate p evaluates to true under σ and $\mathfrak{C}$. (Here σ denotes to the program state in the state S.)

The `while` keyword has an additional argument m as a superscript, which we call the *loop bound*. The loop bound is initially a term t that is evaluated to a bit string $t^{\sigma,\mathfrak{C}}$, which is then interpreted as a natural number n in the other rules. This behaves like a standard while loop except that it terminates if number of iterations exceeds the loop bound. We thus prevent non-termination leakage: in general, it would be observable if a process stops communicating due to non-termination of a loop and this can leak information about the secret variables. Standard information flow is not sensitive for that if one defines noninterference between final states (when the program has terminated) but due to the nondeterminism in the choice of the crypto algebra, we cannot make DYNI insensitive to non-termination. Bounding execution by these loop bounds is conceptually the simplest way to ensure termination. While this limits programs to primitive recursion, it is not a limitation of our approach: the same result would hold without loop bounds, if we are given a proof of termination for the program for every input.

The $\texttt{try\_rcv}(t)\{c\}$ statement is the most involved in the semantics. Recall that the pattern t is a term in $\mathcal{T}(\mathcal{E} \uplus \mathcal{F}, \mathcal{V})$ and can only contain $\bullet$-typed variables. This means that the program will only accept a term that is encrypted and formatted in the specified way, but it can receive arbitrary values for subterms that are variables (and these will be treated as basic values of type $\bullet$). For any term t there exists at most one type τ such that $\mathcal{V} \vdash t : \tau$. For any such term t that types we denote this τ by t^τ. Since the two rules for the receive statement use t^τ, it must be the case that t is well-typed, i.e. we have $\mathcal{V} \vdash t : t^\tau$.[3] Let now b be the bit string (from inbox) that the program is trying to receive. We check if $\mathfrak{C} \vdash b : t^\tau$, i.e., if there is a term of type t^τ that produces b in $\mathfrak{C}$. This is actually the question of whether b can be decrypted and parsed according to pattern t. If no, we take the rule on the right to execute the catch block c (e.g. sending an error message). If yes, then by Lemma IV.3, there is a unique one that we denote with $b{\downarrow}^{\mathfrak{C}}_{t^\tau}$. That term has the same shape as t, but has bit strings instead of variables. In this case, we use the parse function ps (and its extension to lists ps_l) that will parse the given input against the pattern step by step, and updating the state whenever parsing a bit string b against a variable x. Note that, if a variable occurs several times in the pattern, then it will simply be overwritten in the order specified in the ps-rules. Note also that ps will not fail on the given inputs by construction.

The only remaining detail is the bp function to update the counters C_R. This is necessary in our model where the intruder can send arbitrary bit strings; this could include an encryption with a randomization that the random generator of the program can also produce at some point. The function bp advances all counters so far that all random values in the given message will not occur anymore:

Definition V.1. *We define an ordering on the counter functions, such that $C_{R1} \leq C_{R2}$ if and only if $C_{R1}(e) \leq C_{R2}(e)$*

[3] In the Isabelle model we slightly generalize the semantics. If an ill-typed receive pattern is used then the execution is still defined, unlike in Fig. 3, but it will continue with an undefined internal program state (i.e. one we cannot prove anything about).

$$\frac{}{((\sigma, ib, ob, C_R), x \leftarrow t) \underset{\mathfrak{C}}{\rightsquigarrow} (\sigma[x \mapsto t^{\sigma,\mathfrak{C}}], ib, ob, C_R)} \quad \frac{}{((\sigma, ib, ob, C_R), \mathtt{send}(x)) \underset{\mathfrak{C}}{\rightsquigarrow} (\sigma, ib, \sigma(x) :: ob, C_R)}$$

$$\frac{}{((\sigma, ib, ob, C_R), x \leftarrow e(y)) \underset{\mathfrak{C}}{\rightsquigarrow} (\sigma[x \mapsto e(\mathfrak{C}_R(e, C_R(e)), \sigma(y))^{\mathfrak{C}}], ib, ob, C_R[e \mapsto C_R(e)+1])}$$

$$\frac{(S, c_1) \underset{\mathfrak{C}}{\rightsquigarrow} S' \quad (S', c_2) \underset{\mathfrak{C}}{\rightsquigarrow} S''}{(S, c_1; c_2) \underset{\mathfrak{C}}{\rightsquigarrow} S''} \quad \frac{(S, c_1) \underset{\mathfrak{C}}{\rightsquigarrow} S'}{(S, \mathtt{if}(p)\{c_1\}\{c_2\}) \underset{\mathfrak{C}}{\rightsquigarrow} S'} \; \sigma \models_{\mathfrak{C}} p \quad \frac{(S, c_2) \underset{\mathfrak{C}}{\rightsquigarrow} S'}{(S, \mathtt{if}(p)\{c_1\}\{c_2\}) \underset{\mathfrak{C}}{\rightsquigarrow} S'} \; \sigma \not\models_{\mathfrak{C}} p$$

$$\frac{\left(S, \mathtt{while}^{t^{\sigma,\mathfrak{C}}}(p)\{c\}\right) \underset{\mathfrak{C}}{\rightsquigarrow} S'}{(S, \mathtt{while}^{t}(p)\{c\}) \underset{\mathfrak{C}}{\rightsquigarrow} S'} \quad \frac{(S, c) \underset{\mathfrak{C}}{\rightsquigarrow} S' \quad \left(S', \mathtt{while}^{n-1}(p)\{c\}\right) \underset{\mathfrak{C}}{\rightsquigarrow} S''}{(S, \mathtt{while}^{n}(p)\{c\}) \underset{\mathfrak{C}}{\rightsquigarrow} S''} \; \begin{array}{l}\sigma \models_{\mathfrak{C}} p \\ \wedge\, n > 0 \\ \wedge\, n \in \mathtt{B}\end{array} \quad \frac{}{(S, \mathtt{while}^{n}(p)\{c\}) \underset{\mathfrak{C}}{\rightsquigarrow} S} \; \begin{array}{l}\sigma \not\models_{\mathfrak{C}} p \\ \vee\, n = 0\end{array}$$

$$\frac{\mathsf{ps}(\sigma, t, b{\downarrow}^{\mathfrak{C}}_{t^\tau}) = \sigma'}{((\sigma, b :: ib, ob, C_R), \mathtt{try\_rcv}(t)\{c\}) \underset{\mathfrak{C}}{\rightsquigarrow} (\sigma', ib, ob, \mathsf{bp}(C_R, b{\downarrow}^{\mathfrak{C}}_{t^\tau}))} \; \mathfrak{C} \vdash b : t^\tau$$

$$\frac{((\sigma, ib, ob, C_R), c) \underset{\mathfrak{C}}{\rightsquigarrow} S'}{((\sigma, b :: ib, ob, C_R), \mathtt{try\_rcv}(t)\{c\}) \underset{\mathfrak{C}}{\rightsquigarrow} S'} \; \mathfrak{C} \nvdash b : t^\tau$$

$$\frac{}{\mathsf{ps}(\sigma, x, b) = \sigma[x \mapsto b]} \quad \frac{\mathsf{ps}_l(\sigma, [t_1, \ldots, t_n], [u_1, \ldots, u_n]) = \sigma'}{\mathsf{ps}(\sigma, f(t_1, \ldots, t_n), f(u_1, \ldots, u_n)) = \sigma'} \quad \frac{}{\mathsf{ps}_l(\sigma, [], []) = \sigma} \quad \frac{\mathsf{ps}(\sigma, t, u) = \sigma' \quad \mathsf{ps}_l(\sigma', ts, us) = \sigma''}{\mathsf{ps}_l(\sigma, t :: ts, u :: us) = \sigma''}$$

Fig. 3. Big-step semantics of the programming language. bp and t^τ are defined in the explaining text, [] and :: are list notation from functional programming.

$$\frac{\mathcal{X}_{in} \vdash t : \Delta(x)}{\mathcal{X}_{in} \vdash x \leftarrow t : \{x\}}$$

$$\frac{\Delta(x) = e \quad \Delta(e) = ((\bullet, \Delta(y)) \to e)}{\mathcal{X}_{in}, y \vdash x \leftarrow e(y) : \{x\}}$$

$$\frac{}{\mathcal{X}_{in}, x \vdash \mathtt{send}(x) : \emptyset}$$

$$\frac{\mathcal{V} \vdash t : t^\tau \quad \mathcal{X}_{in} \vdash c : \mathcal{X}_{out}}{\mathcal{X}_{in} \vdash \mathtt{try\_rcv}(t)\{c\} : fv(t) \cap \mathcal{X}_{out}}$$

$$\frac{\mathcal{X}_{in} \vdash c_1 : \mathcal{X}_{out1} \quad \mathcal{X}_{in} \cup \mathcal{X}_{out1} \vdash c_2 : \mathcal{X}_{out2}}{\mathcal{X}_{in} \vdash c_1; c_2 : \mathcal{X}_{out1} \cup \mathcal{X}_{out2}}$$

$$\frac{\mathcal{X}_{in} \vdash p \quad \mathcal{X}_{in} \vdash c_1 : \mathcal{X}_{out1} \quad \mathcal{X}_{in} \vdash c_2 : \mathcal{X}_{out2}}{\mathcal{X}_{in} \vdash \mathtt{if}(p)\{c_1\}\{c_2\} : \mathcal{X}_{out1} \cap \mathcal{X}_{out2}}$$

$$\frac{\mathcal{X}_{in} \vdash t : \bullet \quad \mathcal{X}_{in} \vdash p \quad \mathcal{X}_{in} \vdash c : \mathcal{X}_{out}}{\mathcal{X}_{in} \vdash \mathtt{while}^t(p)\{c\} : \emptyset}$$

Fig. 4. Well-typed programs

$$\frac{}{\mathcal{X} \vdash \mathtt{ff}} \quad \frac{\mathcal{X} \vdash p_1 \quad \mathcal{X} \vdash p_2}{\mathcal{X} \vdash p_1 \longrightarrow p_2} \quad \frac{\mathcal{X} \vdash t_1 : \bullet \quad \mathcal{X} \vdash t_2 : \bullet}{\mathcal{X} \vdash t_1 = t_2}$$

Fig. 5. Well-typed formulas

for every $e \in \mathcal{E}$.

We let $\mathsf{bp}(C_R, t)$, for $C_R : \mathcal{E} \to \mathbb{N}$ and term t, be the least $C'_R : \mathcal{E} \to \mathbb{N}$ such that $C_R \leq C'_R$, and for any $e(b, \ldots)$ occurring in t and any $n \in \mathbb{N}$, if $\mathfrak{C}_R(e, n) = b$ then we have $n < C'_R(e)$.

VI. Typing and Labeling Programs

We extend the notion of well-typedness and well-labeledness to programs (Figs. 4,6). Our main result will be that a well-typed and well-labeled program enjoys the property of Dolev-Yao noninterference.

A. Typing rules

Our typing rules ensure, among other things, that we do not read from uninitialized variables. Given a set $\mathcal{X}_{in}$ of variables that are already initialized, we call the program c *well-typed* if $\mathcal{X}_{in} \vdash c : \mathcal{X}_{out}$ holds according to the typing relation defined in Fig. 4. At the end of the program the variables in $\mathcal{X}_{out}$ will also be initialized. Initially $\mathcal{X}_{in}$ contains the variables that are defined in the initial state that the program is started from, and this set is augmented in the type check for the part c_2 of a composition $c_1; c_2$ by the variables that c_1 initializes.

Most typing rules are straightforward. Assignment $x \leftarrow t$ requires that x has the same type as t. Similarly, an encryption with operator e must be to a variable of type e, and the argument types of e must be $\bullet$ (for the randomization) and the type $\Delta(y)$ of the term being encrypted. On a receive, we must check that the receive pattern t is well-typed. Moreover, the loop bound of a while must be of type basic: it can either be a bit string or a basic program expression.

B. Labeling rules

The rules in Fig. 6 check that a program does not contain illegal flows. The relation $\Vdash c : l$ means that c can be labeled l, where l is at most the infimum of the security levels of variables affected by c. This label l is then used for checking implicit flows: this command can only occur under a condition with a security label $l' \leq l$. Before we explain the rules, let us first illustrate how the formats can give rise to implicit flows that do not occur in classical information flow:

$$\frac{\Vdash t : l_t}{\Vdash x \leftarrow t : l} \; l_t \leq \Gamma(x) \;\wedge\; l \leq \Gamma(x) \sqcap \underline{\Delta(x)}$$

$$\frac{\Gamma(e) = ((l_r, l_1) \to l_2)}{\Vdash x \leftarrow e(y) : l} \quad \Gamma(y) \leq l_1 \wedge l_2 \leq \Gamma(x) \wedge l \leq \underline{e} \sqcap l_2$$

$$\frac{}{\Vdash \texttt{send}(x) : \bot} \; \Gamma(x) = \bot \quad \frac{\Vdash t : l_t \quad \Vdash c : l'}{\Vdash \texttt{try\_rcv}(t)\{c\} : \bot}$$

$$\frac{\Vdash c_1 : l_1 \quad \Vdash c_2 : l_2}{\Vdash c_1; c_2 : l} \; l \leq l_1 \sqcap l_2$$

$$\frac{\Vdash c_1 : l_1 \qquad \Vdash c_2 : l_2}{\Vdash \texttt{if}(p)\{c_1\}\{c_2\} : l} \; sup_\Gamma(p) \sqcup l \leq l_1 \sqcap l_2$$

$$\frac{\Vdash c : l'}{\Vdash \texttt{while}^t(p)\{c\} : l} \; sup_\Gamma(t) \sqcup sup_\Gamma(p) \sqcup l \leq l'$$

Fig. 6. Well-labeled programs. We use $sup_\Gamma(t) = \bigsqcup\{\Gamma(x) \mid x \in fv(t)\}$ for terms and formulas.

Example VI.1. *Let $x, y : \top$, $z : \bot$, and $f \in \mathcal{F}$.*

$y \leftarrow f(0,0);$
$\texttt{if}\,(x > 0)\;\{$
$\quad y \leftarrow f(x, z);$
$\}$

One may now encrypt y; a recipient may then decrypt and parse the content and treat the second value in the format as $\bot$. This is because formats allow us to transmit in a single message several values of different security levels and the recipient should be able to treat each item at is original security level. However, this now gives rise to an implicit flow from x into the second argument of f: if that value is later (by a recipient) stored in a $\bot$ variable z', then $z' \neq 0$ reveals that $x > 0$. To correct the example, one could either require the second argument of f be labeled $\top$ (so the recipient cannot treat it as low) or slightly change the behavior of the program:

$x' \leftarrow 0$
$\texttt{if}\,(x > 0)\;\; x' \leftarrow x$
$y \leftarrow f(x', z)$

Here, the second argument of f has no more relation to x (in contrast to the previous version) and thus the assignment can be taken out of the conditional, eliminating the implicit flow.

To formalize all the implicit flows here, we define the notion of the *infimum label of a type*: $\underline{\bullet} = \top$ and $\underline{f} = \bigsqcap\{l_1, \ldots, l_n, \underline{\tau_1}, \ldots, \underline{\tau_n}\}$ for $f \in \mathcal{E} \uplus \mathcal{F}$ with $\Gamma(f) = ((l_1, \ldots, l_n) \to l)$ and $\Delta(f) = ((\tau_1, \ldots, \tau_n) \to f)$. The label $\underline{\tau}$ will be at least as low as any field affected by writing to a variable of type τ.

With that, we can define the labeling rules. For an assignment $c = x \leftarrow t$, it must be legal to construct t according to labels of the format fields and the variables used, t must be permitted to flow into x, and the label l of the program can be at most the label of x and the infimum type $\underline{\Delta(x)}$. This reflects that c entails implicit flows into the fields of formats and encryptions contained in $\Delta(x)$ as illustrated in the previous example, and thus if c is part of a conditional statement or loop, then the guard can be at most of label l.

Similarly, for an encryption $c = x \leftarrow e(y)$, y can be at most the level l_1 of the content, and the output level l_2 of the encryption can be at most the level of x. As before, this can induce implicit flows into the fields of formats, and also the encrypted term itself. Thus, the label of c can be at most the infimum type of e and at most the encryption output level l_2.

For sending and receiving, we require the explicit and implicit flows to be of the label $\bot$, as we assume that the network is public. We note that the label l' in the `try_rcv` rule is totally unconstrained. There is actually an implicit flow from the network to this label, but as the network has label $\bot$, this flow is always legal.

The remaining rules are like classical information flow (where we treat the loop bound t in while loops like a second condition): we have to keep track of the infimum of variables that are affected by each piece of code, and ensure that this label is not lower than the label of any condition it depends on to prevent illegal implicit flows.

For the ease of the presentation, we have made the labeling regime a bit stricter than necessary. Consider the program

$z \leftarrow f(x)$
$\texttt{try\_rcv}(f(y))$

where the security lattice is $\bot < + < \top$ and $\Delta(x) = \bullet$, $\Delta(y) = \bullet$, $\Delta(z) = f$, $\Delta(f) = ((\bullet) \to f)$, $\Gamma(x) = \bot$, $\Gamma(y) = \top$, $\Gamma(z) = +$, and $\Gamma(f) = ((+) \to +)$.

This program satisfies DYNI, however it does not label according to our definition. The Isabelle formalization of the labeling actually allows this program, the result is thus more general, but requires two labeling systems, namely one for program expressions (that allow lower variables to flow into higher fields of formats) and one for receive patterns (that allow lower fields of formats to flow into higher variables). We have chosen to present this more strict labeling system for the sake of simplicity, and since this does actually not restrict expressiveness as the following rewrite demonstrates:

$x' \leftarrow x$
$z \leftarrow f(x')$
$\texttt{try\_rcv}(f(y'))$
$y \leftarrow y'$

with $\Delta(x') = \Delta(y') = \bullet$ and $\Gamma(x') = \Gamma(y') = +$.

Example VI.2. *The typing and labeling systems ensure not just that information flow is adhered to in the classical sense, but also that type confusions introduced by the intruder cannot be exploited to break confidentiality.*

Consider the following ill-typed program, where s is a secret variable and e_{k1} and e_{k2} have secret inputs, so the intruder cannot decrypt such messages:

$\texttt{try\_rcv}(x)$
$\texttt{if}\,(s = 1)\;\{$
$\quad x \leftarrow 0$
$\}$
$y \leftarrow e_{k1}(x)$
$\texttt{send}(y)$

```
try_rcv(e_k1(e_k2(z))) {
   send(error)
}
```

Let x be of type $\bullet$. Based on the secret s, x is either kept or overwritten with 0, encrypted, and then sent. The second part of the program tries to decrypt a received term first with $k1$ then with $k2$. It then answers with error if the decryption fails.

There is an attack on this program, where the intruder first sends an e_{k2}-term (e.g. a previously observed one) and then sends the received answer right back to the program. The program answers with `error` *iff $s = 1$. The program is ill-typed in our system: from $e_{k1}(x)$ and $x : basic$, follows, that $\delta(e_{k1}) = (\bullet)$ and thus $e_{k1}(e_{k2}(\cdot))$ cannot type.*

This example thus demonstrates that a requirement that an intruder can only send well-typed messages in general can exclude attacks that our model can detect. A situation like this can realistically occur in larger programs when the programmer(s) did not realize how messages sent and received at different points in the program could be abused by an intruder in unexpected ways.

VII. Recipes

As the last step, we define the actual intruder model, namely what the intruder can do to distinguish two given states. Recall that in Definition III.1 we have referred to an abstract concept of *recipes*, i.e., steps that the intruder can execute in a given state to produce bit strings, and two states are indistinguishable if every recipe gives the same result in both states. As before, let $l_{ref} \in \mathcal{L}$ be the security level of the intruder, representing that the intruder has legitimate access to all information labeled $l \leq l_{ref}$, and the goal is to protect all other information.

A. Destructors

The encryption and format operators we have considered so far are all constructors; operators for decryption and parsing have only been implicitly represented by the receive patterns of programs so far. For the intruder, we now make explicit a set $\mathcal{D}$ of *destructor symbols*. Each destructor $d \in \mathcal{D}$ has a unique *target* $target(d) \in \mathcal{F} \uplus \mathcal{E}$ and argument $\mathbf{dc}(d) \in \{1, \ldots, arity(target(d))\}$. Here $\mathbf{dc}(d) = i$ means that d extracts the ith argument of f.

The meaning of a destructor d w.r.t. a given crypto algebra $\mathfrak{C}$ is defined as follows: $d(b)^{\mathfrak{C}} = b_i$ if $\mathbf{dc}(d) = i$ and $b = target(d)(b_1, \ldots, b_n)^{\mathfrak{C}}$; and $d(b)^{\mathfrak{C}}$ is undefined otherwise. This is well-defined thanks to the injectivity property in Definition II.1 and the fact that a destructor symbol has a unique target.

B. Intruder Accessible Subterms

We allow the intruder to access those subterms of an encrypted message that have a security label $l \leq l_{ref}$, even if the intruder does not actually have the necessary decryption key and thus cannot see other parts of the encrypted message. Since all $l \leq l_{ref}$ information can actually flow into variables that the intruder can see, this over-approximation is unlikely to lead to false positives, but it simplifies the formal model. The intruder can provide a term and a path (a sequence of indices) into that term and obtains either a bit string corresponding to the subterm pointed to by that path, or *undefined* if the path is not valid or leads to a classified subterm. Let thus $l \in \mathcal{L}$, $t \in \mathcal{T}(\mathcal{F} \uplus \mathcal{E}, \mathtt{B})$, and ψ be a list of natural numbers:

$$access(l, t, \psi) = \begin{cases} t \mid \text{if } \psi = [] \text{ and } l \leq l_{ref} \\ access(l_i, t_i, \psi') \\ \quad \mid \text{if } \psi = i :: \psi' \text{ and } t = f(t_1, \ldots, t_n) \\ \quad \text{and } 1 \leq i \leq n \\ \quad \text{and } \Gamma(f) = ((l_1, \ldots, l_n) \to l') \\ undefined \mid \text{otherwise} \end{cases}$$

The label l here represents the label of the given term; this will be $\Gamma(x)$ when used on a program variable x, or $\bot$ when used for a message on the network, as will be defined in the recipes below. During the recursive calls l is the security label of the respective subterm. The intruder finally gets the chosen subterm if its label is at or below l_{ref}.

C. Blacklist

We call an encryption operator e *critical* and write $crit(e)$ if $\Gamma(e) = ((l_1, \ldots, l_n) \to l)$ and $l \leq l_{ref}$ and $\bigsqcup\{l_1, \ldots, l_n\} \not\leq l_{ref}$. This means that the intruder can actually see such encryptions ($l \leq l_{ref}$) while they are not transparent to the intruder, i.e., at least one l_i is not accessible.

Below, we allow the intruder to freely use encryption symbols and in fact choose any natural number as randomization, however, we have to exclude that the intruder can here choose a random number that has already been used by the program, i.e., the intruder can never guess a random number that the program has used. For that, we extend the program states with a log file Ω that contains the entry (rand, e, b_1) if b_1 is a bit string that the program has already used as randomization for encryption e and we define:

$$blacklist(\mathfrak{C}, \Omega) = \{(e, b_1) \mid (\mathsf{rand}, e, b_1) \in \Omega \wedge crit(e)\}$$

We will simply disallow the intruder to apply e to arguments if the result is in the blacklist, effectively denying the intruder the ability to guess the randomization of critical encryptions.

D. Recipes and their Evaluation

We define the set of recipes by the grammar:

$$\begin{aligned} \mathbf{r} ::= {} & \mathsf{accV}(x, \psi) \mid \mathsf{acclb}(\tau, i, \psi) \mid \mathsf{accOb}(\tau, i, \psi) \\ & \mid f(\mathbf{r}_1, \ldots, \mathbf{r}_{arity(f)}) \mid d(\mathbf{r}') \end{aligned}$$

for $x \in \mathcal{V}$, $\tau \in T$, $i \in \mathbb{N}$, $\psi \in \mathbb{N}^*$, $f \in \mathcal{E} \uplus \mathcal{F} \uplus \mathcal{B}$, and $d \in \mathcal{D}$.

The first three constructs, $\mathsf{accV}(x, \psi)$, $\mathsf{acclb}(\tau, i, \psi)$, and $\mathsf{accOb}(\tau, i, \psi)$, represent the intruder accessing a subterm of either a program variable, a value in the inbox, or a value in the outbox, respectively. All three require a path to the subterm ψ. The type τ is the type that the intruder wishes to interpret the bit string as (while for a program variable x this type is determined by $\Delta(x)$). The other two cases mean that the intruder can apply a constructor $f \in \mathcal{E} \uplus \mathcal{F} \uplus \mathcal{B}$ or a destructor $d \in \mathcal{D}$ to any terms that he can already cook.

Recipes may fail, for example if the intruder tries to access a classified subterm using $access(\cdot,\cdot,\cdot)$. It will then evaluate to $undefined$, and if a recipe uses an undefined subrecipe it will also evaluate to $undefined$.

Let $S = (\sigma, ib, ob, C_r, \Omega)$ be a state where $ib = [ib_0, \ldots]$ and $ob = [ob_0, \ldots, ob_n]$ (and ob_j is undefined for $j > n$). Define the evaluation of recipe $\mathbf{r}$ in S under crypto algebra $\mathfrak{C}$ as follows:

$S(\mathsf{accV}(x,\psi))^{\mathfrak{C}} = access(\Gamma(x), \sigma(x)\downarrow^{\mathfrak{C}}_{\Delta(x)}, \psi)^{\mathfrak{C}}$

$S(\mathsf{acclb}(\tau,i,\psi))^{\mathfrak{C}} = access(\bot, ib_i\downarrow^{\mathfrak{C}}_{\tau}, \psi)^{\mathfrak{C}}$ if $\mathfrak{C} \vdash ib_i : \tau$

$S(\mathsf{accOb}(\tau,i,\psi))^{\mathfrak{C}} = access(\bot, ob_i\downarrow^{\mathfrak{C}}_{\tau}, \psi)^{\mathfrak{C}}$ if $\mathfrak{C} \vdash ob_i : \tau$

$S(f(\mathbf{r}_1,\ldots,\mathbf{r}_n))^{\mathfrak{C}} = f(S(\mathbf{r}_1)^{\mathfrak{C}},\ldots,S(\mathbf{r}_n)^{\mathfrak{C}})^{\mathfrak{C}}$
if $f \in \mathcal{E} \uplus \mathcal{F} \uplus \mathcal{B}$, all $S(\mathbf{r}_i)^{\mathfrak{C}}$ are defined,
and $(f, S(\mathbf{r}_1)^{\mathfrak{C}}) \notin blacklist(\mathfrak{C}, \Omega)$

$S(d(\mathbf{r}))^{\mathfrak{C}} = d(S(\mathbf{r})^{\mathfrak{C}})^{\mathfrak{C}}$ if $d \in \mathcal{D}$ and $\neg crit(target(d))$

and $S(\mathbf{r}) = undefined$ otherwise.

VIII. Main Result

We have shown the soundness of the typing and labeling rules of Section VI by proving that a well-typed and well-labeled program fulfills the DY noninterference of Section III.

For DY noninterference to be defined, the constructs we have introduced in the preceding sections must be well-formed with regard to the requirements from Section III. That $\sim_{classic}$ from Section V-B is an equivalence relation is easy to see. We must also show that for any S, T, $\mathfrak{C}$, and $\mathfrak{C}'$ if $S \overset{\mathfrak{C}}{\sim}{}^{\mathfrak{C}'}_{DY} T$ then $S \sim_{classic} T$, for $\sim_{DY}$ defined from the recipes of Section VII. This follows from the fact that the intruder can use the recipe $\mathsf{accV}(x, [])$ to access the bit string stored in any variable with $\Gamma(x) \leq l_{ref}$, $\mathsf{acclb}(\bullet, i, [])$ to access the bit string stored at any index i of the inbox, and $\mathsf{accOb}(\bullet, i, [])$ to access a bit string from the outbox.

In the following, we state the result more formally. Only payload-type variables will be initialized at the start of the program, as is required of initial states in Section V.

Theorem VIII.1. *If $\{x \mid x \in \mathcal{V} \wedge \Delta(x) = \bullet\} \vdash c : \mathcal{X}_{out}$ and $\Vdash c : l$ then, for any $l_{ref} \in \mathcal{L}$, c fulfills DY noninterference, with regard to the program semantics of Section V, the recipes of Section VII, and init and $\sim_{classic}$ of Section V.*

For reasons of space, we omit the proof here. The general strategy is finding a stronger relation than DY noninterference, which we dub the *invariant*, that we can show is preserved. The details of the proof can be found in our Isabelle/HOL formalization.

IX. Conclusion and Related Work

There are several works that combine static analysis of programs with cryptography. Abadi [1] gives a type system for showing secrecy in programs communicating over public channels with cryptography in the Dolev-Yao model. Peeter Laud was the first to give computational soundness results for information flow analyses. In [10] he proves the soundness of an analysis over the graph representation of a program, some of the restrictions of which were lifted in [8]. Laud and Vene [11] give the analysis in the form of a type system, more similar to our approach. Smith and Alpízar [14] make further progress in this vein, among other things, by adding a decryption statement to the considered programming language. Fournet and Rezk [7] consider probabilistic noninterference in a context with an active adversary. They give a probabilistic language with cryptography and untrusted memory, which may be considered to model an untrusted network, and express noninterference as a game defined in that language. They also provide a simpler imperative language that compiles into the target probabilistic language and show that if a program types in the simple language then its implementation in the probabilistic language also types, which guarantees noninterference. Fournet, Planul, and Rezk [6] extend the typing rules to cover cryptographic operators with different properties like encryption with blinding and homomorphic encryptions, where computations on plaintexts can be done on ciphertexts instead. Waye et al. [15] propose CLIO, an information-flow control system that applies cryptographic enforcement of information-flow control over untrusted key-value stores. They formalize security as a game against an active intruder, and prove that programs satisfying certain properties are secure under this game. Another work that considers information flow with encryption is Mitchell et al. [12], which provides a Haskell-based programming environment for homomorphic encryption and secure multiparty computation. They can give a computationally based guarantee that a permissible program in their model will not leak secrets to the honest-but-curious servers it is executed over.

Our work also shares certain similarities with the labeled crypto API of Bhargavan et al. [3] (further extended in Bhargavan et al. [4] to support compositional verification of vertically composed protocols). Their objective is quite different from ours — they verify more complex security protocols, like the Diffie-Hellman ratcheting protocol, especially with regard to dynamic security properties, like post-compromise security and forward secrecy, while we merely want to find the sound ways of using existing cryptographic material at the application level. As such, we have a much simpler analysis and can give a strong privacy-type noninterference guarantee, where they show secrecy-type properties. However, their approach is based on showing an invariant over the global state of the system, which is similar to our approach for showing that noninterference is implied by our typing system (though our invariant is over pairs of states), and they do so by defining a typing system defining the intended use of the different program variables. For example, a key might be a dedicated signing key, and should then never be used for encryption. This is somewhat similar to our requirement that if a given encryption symbol is used for encrypting a certain type of data (for example public payloads) then it may not be used to encrypt incompatible types of data (for example a format with a secret field). Furthermore, they have secrecy labels specifying that only a given set of principals are allowed to read a certain piece of information, very much like our security

labels.

The closest related work is Askarov, Hedin, and Sabelfeld's work on cryptographically-masked flows [2]. They use a concept of *low-equivalence* between pairs of states to define *possibilistic noninterference* for programs with encryptions. For non-cryptographic terms, low-equivalence ($\sim$) is simply that terms of a low security level must be equal in the two states, while terms of a high security level may differ. A single term may contain both high and low subterms, similarly to what is permitted by our model through formats. To define low-equivalence for terms including encryptions, they also introduce a low-equivalence relation for ciphertexts ($\doteq$), which is left arbitrary but for a couple of conditions. They assume a nondeterministic encryption scheme, where the result of an encryption can be any of a set of ciphertexts. They then require of any (possibly equal) pair of encryptions and associated output sets the following: for all ciphertexts in one set there must exist a low-equivalent ciphertext in the other, and there must exist a ciphertext in one set that is not low-equivalent to a ciphertext in the other set. For $u_1 \sim u_2$ to hold, where u_1 is the result of encrypting the term v_1 with the key k_1 and u_2 is the result of encrypting v_2 with k_2, we must have $u_1 \doteq u_2$, $v_1 \sim v_2$, and $k_1 \sim k_2$. They extend low-equivalence to states as expected. Two sets of states are defined as low-equivalent if for any state in one set there exists a low equivalent state in the other set, and vice versa. They give a typing system for commands and show that if a command is well-typed, then the two sets of end states reachable from two well-formed and low-equivalent sets of initial states will be low-equivalent.

A significant difference to our work is that [2] limits the messages that the intruder can send. This amounts to the intruder only sending "well-typed" messages (although not exactly in the sense of well-typing in this paper), so that the normal program variables never contain cryptographic messages. In contrast, our intruder can send arbitrary bit strings, and the program variables can thus take values that contain cryptographic messages (or parts thereof), and the program may compute with these values and conditions may depend on these values. The semantics for this follows just what happens in reality: if the program expects an encrypted message then it will simply apply the respective decryption, and if that succeeds, it will accept any bit string results from the decryption—and we do not exclude that this can be a cryptographic message. Example VI.2 gives a program where we can show an attack in our model that exploits this. As far as we can see, this is incompatible with the model of nondeterministic encryption in [2]. Our model, our requirements, and our proof of a noninterference result are more complicated because of this generalization: the only feasible way we see to model and prove it was to show that for two executions that should be indistinguishable and one crypto algebra $\mathfrak{C}$ for one execution, we can always find a crypto algebra $\mathfrak{C}'$ for the other execution so that the cryptographic terms are identical in both executions, thus the same paths are followed, while the variables and message fields that the intruder cannot see may indeed be different. This thus models in a completely different way that to an intruder with no clue about cryptography, the messages in the two executions look literally the same.

The additional requirement to the program that we need for our result are part of the type-checking. This ensures that a program distinguishes between the intended types of data, and thus only produces well-typed terms, and this property is preserved over all information flows, including communication, since formatting always conveys the intended type of the information. Related to that, we also have the requirement (that is ensured by construction) that every encryption by the program is randomized with a value different from all values that occurred before. This is handled in [2] by the nondeterminism of their model, but it seems a good property of our model that non-randomized encryption (or flaws in the randomization) would allow the intruder to perform guessing attacks or see whether two encrypted messages have the same content.

Like our approach, also the results of [2] are machine-checked, in their case using a theorem prover called Coq at the time. As explained our approach is more general in terms of the messages from the intruder and thus also the proof of the result is rather complicated, which also motivates the use of machine checked proofs because in a pen-and-paper proof it would be so easy to overlook an error in a minor detail at this complexity level. Another advantage of using theorem provers is that we can actually treat programs as mathematical objects and verify multiple properties, of which noninterference is only one.

Laud [9] has shown that noninterference in the possibilistic model of cryptographically-masked flows of [2] (with some minor modifications) implies noninterference in a probabilistic computational model. Note that this concerns only the distinguishability of the messages sent by the program and the low variables (given the cryptography is computationally hard to break), but not the messages the intruder may send to the program where the approach limits the intruder to well-typed messages as described. Given that our model of cryptography is also substantially different to [2] in order to handle an intruder who can send arbitrary bit strings, it is currently unclear how a computational variant of our Dolev-Yao noninterference can be formalized and proved, and we leave this question for future work.

Such a proof would also eliminate the current reliance on a blacklist of values that the intruder cannot guess and on a program semantics that prevents collisions in the random number generator; in a computational proof, these guessing and collision issues would not be impossible in the model, but instead have a negligible probability.

In the future, we would also like to detach the keys from the encryption symbols and allow explicit key generation and sharing. Some authenticity considerations would have to be included in the analysis; one should, for example, not immediately accept a key encrypted with one's public key as coming from the sender one would expect. Complex interactions for transmitting and verifying keys would likely require something

closer to full protocol-verification, but simple operations like generating and transmitting a new session key could be within the realm of our typing-based approach.

Another limitation of the current approach is that we do not allow for a program to handle encrypted messages that they cannot decrypt. An example that we cannot handle is be a server that forwards messages between clients that are encrypted between the clients with a key that the server does not know. This restriction was made to keep the formalization and proofs simple, but we do not see any fundamental issue that would prevent us from lifting this restriction and thus we plan this for future work.

Declassification would be an interesting aspect to examine as future work. The analysis could become more generally applicable if the programmer had the ability to declassify the value contained in a variable at any point in the program. They would then, of course, lose the noninterference guarantees for that variable and any logical consequence of that (if x was just assigned to y and y is then declassified, x would then also be implicitly declassified), but ideally all other guarantees still hold. You might, for example, want to declassify the result of an election without declassifying each vote. A typical approach is to limit the considered pairs of executions to those where the declassification has no effect, and show that noninterference holds for these. Eilers, Müller, and Hitz [5] employ this technique by representing two executions of a program as a single product program and adding `assume`-statements stating that the declassified value is equal in the two executions at the point where declassification happens. This would, however, not work directly in our model since the second execution is existentially quantified.

We want to conclude with the question of whether one could use our information flow analysis also for integrity goals, e.g., with a lattice $\mathit{trusted} \leq \mathit{untrusted}$. (Of course, $\mathit{trusted}$ is the lower label, because untrusted information must not flow into trusted sources.) The first major obstacle to this lies in the current model that allows the intruder to send arbitrary bit strings to the program as input, and thus in general may influence all variables of the program. We would thus have to limit this to an intruder who can send only messages that they already know or that they can produce (but then for untrusted fields insert arbitrary input values). The second major problem arises if the intruder can intercept messages on the network or replay them. While this does not necessarily violate a notion of noninterference (if we consider two executions with the same intercepts and replays), but since the value of trusted variables in general depend on whether a message was received, the intruder *can* thus influence the trusted variables. Also this question we plan to investigate as future work.

References

[1] Martín Abadi. "Secrecy by Typing in Security Protocols". In: *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*. Vol. 1281. Lecture Notes in Computer Science. Springer, 1997, pp. 611–638. DOI: 10.1007/BFB0014571.

[2] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. "Cryptographically-masked flows". In: *Theor. Comput. Sci.* 402.2-3 (2008), pp. 82–101. DOI: 10.1016/J.TCS.2008.04.028.

[3] Karthikeyan Bhargavan et al. "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 523–542. DOI: 10.1109/EUROSP51992.2021.00042.

[4] Karthikeyan Bhargavan et al. "Layered Symbolic Security Analysis in DY*". In: *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part III*. Vol. 14346. Lecture Notes in Computer Science. Springer, 2023, pp. 3–21. DOI: 10.1007/978-3-031-51479-1_1.

[5] Marco Eilers, Peter Müller, and Samuel Hitz. "Modular Product Programs". In: *ACM Trans. Program. Lang. Syst.* 42.1 (2020), article no. 3. DOI: 10.1145/3324783.

[6] Cédric Fournet, Jérémy Planul, and Tamara Rezk. "Information-flow types for homomorphic encryptions". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. ACM, 2011, pp. 351–360. DOI: 10.1145/2046707.2046747.

[7] Cédric Fournet and Tamara Rezk. "Cryptographically sound implementations for typed information-flow security". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008, pp. 323–335. DOI: 10.1145/1328438.1328478.

[8] Peeter Laud. "Handling Encryption in an Analysis for Secure Information Flow". In: *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Vol. 2618. Lecture Notes in Computer Science. Springer, 2003, pp. 159–173. DOI: 10.1007/3-540-36575-3_12.

[9] Peeter Laud. "On the computational soundness of cryptographically masked flows". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008, pp. 337–348. DOI: 10.1145/1328438.1328479.

[10] Peeter Laud. "Semantics and Program Analysis of Computationally Secure Information Flow". In: *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of*

Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Vol. 2028. Lecture Notes in Computer Science. Springer, 2001, pp. 77–91. DOI: 10.1007/3-540-45309-1_6.

[11] Peeter Laud and Varmo Vene. "A Type System for Computationally Secure Information Flow". In: *Fundamentals of Computation Theory, 15th International Symposium, FCT 2005, Lübeck, Germany, August 17-20, 2005, Proceedings*. Vol. 3623. Lecture Notes in Computer Science. Springer, 2005, pp. 365–377. DOI: 10.1007/11537311_32.

[12] John C. Mitchell et al. "Information-Flow Control for Programming on Encrypted Data". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. IEEE Computer Society, 2012, pp. 45–60. DOI: 10.1109/CSF.2012.30.

[13] Sebastian Mödersheim and Georgios Katsoris. "A Sound Abstraction of the Parsing Problem". In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society, 2014, pp. 259–273. DOI: 10.1109/CSF.2014.26.

[14] Geoffrey Smith and Rafael Alpízar. "Secure information flow with random assignment and encryption". In: *Proceedings of the 2006 ACM workshop on Formal methods in security engineering, FMSE 2006, Alexandria, VA, USA, November 3, 2006*. ACM, 2006, pp. 33–44. DOI: 10.1145/1180337.1180341.

[15] Lucas Waye et al. "Cryptographically Secure Information Flow Control on Key-Value Stores". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 1893–1907. DOI: 10.1145/3133956.3134036.

APPENDIX

A. Notes on Typing

There should be no real issue with adapting our work to a more permissive typing system, similar to the one of [2]. If we add a requirement to the base algebra $\mathfrak{B}$ that formats must be distinguishable (i.e., that if $\mathfrak{B}(f)(b_1, \ldots, b_n) = \mathfrak{B}(g)(b'_1, \ldots, b'_m)$ for $f, g \in \mathcal{F}$ then $f = g$) then one would not have problems with a message containing secret information being mistaken for a message with public information, like in Example IV.2, even if the same encryption symbol can be applied to different formats. It would, however, make some of the proofs more involved, and we thus leave it as future work.

To reiterate, our term-typing scheme is quite strict; if we have $\Delta(f) = ((g, \bullet) \to f)$, then we expect an honest agent to only ever produce f-terms where the first input is a g-term and the second is a basic value. For simplicity, we have made the type system so that each encryption operator has to have one type of content, because it is more involved to handle a type system that allows for a disjunction of operators at a given place. A simple workaround we can use is the following syntactic sugar: we can encode a set of formats $f_1, \ldots, f_k$ simply as a single format with $1 + \sum_{i=1}^{k} arity(f_i)$ arguments where the first argument is a value from $1, \ldots, k$, choosing which format it is, as expected; this can also be further reduced as expected if several formats have an argument of the same Δ and Γ. The first argument should be of label $\bot$ since we do not hide the types of messages. If the formats are implemented using something like XML, this combined format can actually be implemented without loss of efficiency, as the unused fields need not even be mentioned.

This will not always work directly, since if you assign to a format with only secret field in a secret context you will have an illegal implicit flow when the syntactic sugar is unfolded. There is, however, also a workaround to this: One can write to temporary variables inside the secret context and then assign these values to the format after the while- or if-statement.