Verifiable Security Policies for Distributed Systems

Felix A. Wolf
Department of Computer Science
ETH Zurich, Zurich, Switzerland
felix.wolf@inf.ethz.ch

Peter Müller
Department of Computer Science
ETH Zurich, Zurich, Switzerland
peter.mueller@inf.ethz.ch

Abstract

In the context of secure information flow, security policies express the classification and declassification of data. Existing policy frameworks are tightly linked to a programming language, which limits their flexibility and complicates reasoning, for instance, during audits. We present a framework for the specification and verification of security policies for distributed systems, where attackers may observe the I/O performed by a program, but not its memory. Our policies are expressed over the I/O behaviors of programs and, thereby, language-agnostic. We present techniques to reason formally about policies, and to verify that an implementation satisfies a given policy. We formalize these verification techniques in Isabelle/HOL. An evaluation on several case studies, including an implementation of the WireGuard VPN key exchange protocol, demonstrates that our policies are expressive, and that verification is amenable to SMT-based verification.

CCS Concepts

• **Security and privacy → Logic and verification**.

Keywords

Security Policy, Declassification Policy, Secure Information Flow, Code Verification, Automated Verification

ACM Reference Format:
Felix A. Wolf and Peter Müller. 2024. Verifiable Security Policies for Distributed Systems. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690303

1 Introduction

For programs handling confidential data, one crucial concern is *secure information flow*, meaning that confidential data is not leaked during the program's execution. In this context, *security policies* express (1) the *classification* of data, for instance, by designating part of the data as sensitive and others as public information, and (2) the *declassification* of data, that is, rules that describe when sensitive data can be deliberately treated as public.

Consider a simple authentication service. A security policy may classify that keys read from disk and network packets have high and low sensitivity, respectively. For declassification, a security policy may permit declassifying messages signed with the read key.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

Recent works [9, 35, 40, 48, 50] have introduced policy frameworks both to formally define security policies and to verify that code actually satisfies a defined policy. These existing approaches have two limitations: (1) Their frameworks are tightly linked to a programming language, which has two drawbacks. First, they cannot express policies in a language-agnostic way, which is for instance useful in distributed systems, where different nodes may be implemented in different languages. Second, reasoning about policies happens at the level of the programming language and, thus, involves the full complexity of the language. (2) Existing approaches enable verifying that an implementation satisfies a policy, but do not support reasoning about the policy itself, in particular, to validate that it expresses the intended security requirements.

This Work. We introduce a new policy framework that addresses these limitations by expressing policies over traces of *I/O actions*, the basic building blocks of communication, such as sending or receiving a message. This language-independent representation is well-suited for distributed systems, where attackers observe the I/O behavior of a program, but *not* the content of the memory.

To specify *classification*, we associate with each I/O action pre- and postconditions that express the sensitivity of outputs and inputs, respectively. For *declassification*, we introduce a designated action $decl(x)$, which declassifies a value x. A security policy is then a tuple of a *classification spec* (the pre- and postconditions for I/O actions) and an *IOD spec*, specifying the traces of I/O actions and declassification actions that an implementation may produce.

Such policies are independent of the program to be verified, a specific programming language, and the verification logic used to prove that an implementation satisfies a policy. In particular, the same policy can be used for multiple different implementations, even with different programming languages, which addresses the first limitation of existing frameworks discussed above. Regarding the second limitation, a key advantage of our framework is that policies can be audited completely independent of code and programming language, both formally and informally. In particular, we introduce a verification technique to show that *all* programs satisfying the policy guarantee that specific data remains confidential even in the presence of declassification.

To prove that a concrete program satisfies a given policy, one can use standard program verification techniques. We show how to use ghost state (state that is used for verification but erased during compilation) to store the trace of I/O actions and declassification actions produced by a program execution. We can then prove that a program satisfies a policy by showing that (1) the stored traces refine the policy's IOD spec and (2) the stored traces satisfy secure information flow as expressed by the policy's classification spec.

For the latter, we introduce a new formalization of secure information flow that can deal with declassification and I/O behavior, and

is expressive enough for language agnostic policies. Our formalization is inspired by observational determinism [33, 43] and amenable to standard verification techniques such as self-composition [11–13, 25]. However, in contrast to observational determinism, our formalization is not restricted to deterministic I/O behavior, which is crucial for concurrent and distributed systems.

We formalize our technique for verifying an implementation against a security policy based on the SecCSL logic [26] and prove its soundness in Isabelle/HOL. We demonstrate the practicality of our approach on a variety of case studies, including an implementation of the WireGuard protocol. These case studies are carried out using Gobra [53], an automated code verifier for Go, which demonstrates that our approach is amenable to automation using SMT solvers. Moreover, they show that our policy framework supports established security policy concepts such as delimited release [45], state-dependent declassification, and robust declassification [56].

Contributions. We make the following contributions:

- We introduce a new policy framework based on I/O behaviors that allows one to express classification and declassification requirements independently of a given program or programming language.
- We show how to reason formally about the guarantees provided by a policy, enabling formal audits of security policies.
- We present a technique to verify that an implementation satisfies a given policy. We formalize this technique based on SecCSL and prove its soundness in Isabelle/HOL.
- We illustrate the expressiveness of our policy framework on several case studies, including an implementation of the Wireguard protocol. These case studies also show that our technique is amenable to SMT-based verification and scales to real-world code of considerable size.

Our formalization, all proofs, and all verified programs are available in our artifact [54].

2 Overview

Fig. 1 gives an overview of our policy framework. In this section, we give a high-level overview of its main components; details are discussed in the subsequent sections.

Language Independence. To obtain a language-independent framework for policies and policy validation, we represent programs as their *IOD behavior*, i.e., the traces of I/O actions and declassifications that the program may produce. This representation has two advantages: (1) As shown by previous works on protocol verification [1, 38, 52], I/O actions and, thereby, IOD behaviors provide a language-agnostic representation of program behavior. (2) Reason about traces of I/O actions allows us to abstract from implementation details such as memory representations and concurrency.

In contrast to policy specifications (including the definition of *policy compliance*, which defines when a policy is satisfied), verifying that a given implementation satisfies a given policy is inevitably language-specific. For this purpose, we instantiate our framework with two concrete program verification approaches: We extend an existing formalization of SecCSL to prove soundness of our approach and use the existing Gobra verifier for our case studies.

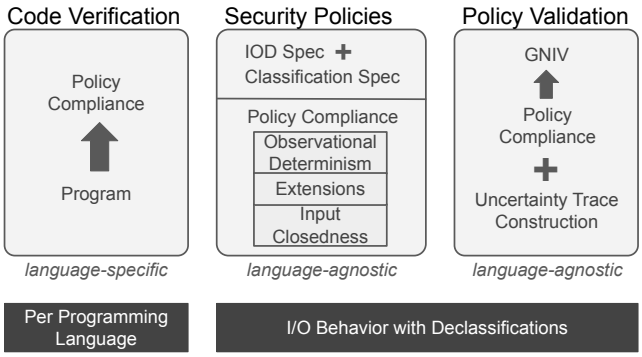

Figure 1: An overview of policy framework. At the center are our security policies and a definition of policy compliance based on a variation of observational determinism. Policy validation (on the right) allows one to prove that a given policy indeed guarantees the intended security properties, in our case an adaptation of Generalized Non-Interference. Both components are based on I/O behavior with declassifications and, thereby, language agnostic. Code verification (on the left) allows one to prove that a given program satisfies a given policy.

Security Policies. As we have explained in the introduction, our language-agnostic security policies consist of a *classification spec* (the pre- and postconditions for I/O actions) and an IOD spec, specifying the traces of I/O actions and declassification actions that an implementation may produce.

The main challenge of using IOD behaviors is to define policy compliance, such that it satisfies three important requirements: (1) The definition must be expressive enough to capture the behavior of realistic programs. (2) The definition must be strong enough to prove guarantees during policy validation. (3) The definition is amenable to standard program verification techniques and tools, in order to minimize the effort that is necessary to adapt code verification to other languages.

We achieve these goals through a combination of three ingredients, namely observational determinism, extensions, and input-closedness. To facilitate code verification, we define policy compliance using observational determinism (OD), which many existing code verification techniques support. However, standard OD does not allow non-deterministic behavior, which is pervasive in concurrent and distributed systems. We solve this problem by enriching traces with information about non-deterministic choices. These *extensions* effectively externalize non-deterministic choices, such that standard observational determinism applies. Soundness is preserved by requiring *input-closedness*, a well-definedness condition for extensions.

Code Verification. Our definition of policy compliance allows us to apply standard code verification techniques and tools. Code verification proves that a program produces only the IOD behaviors permitted by the policy, which is achieved by generating appropriate proof obligations for each I/O and declassification action. Moreover, code verification needs to ensure that the implementation satisfies the classification spec, using standard OD-reasoning.

To prove soundness of our approach, we build on the formalization of SecCSL [26], an existing logic for OD-reasoning. For our case studies, we apply Gobra [53], an off-the-shelf automated verifier for Go programs. Supporting other programming languages and verification tools is straightforward and, in particular, does not require any changes to the language-agnostic parts of our framework.

Policy Validation. We propose a methodology for proving—based only on a policy—that data remains confidential even in the presence of declassification. We formalize this property as *Generalized Non-Interference Modulo Views (GNIV)*. GNIV is a more flexible definition of generalized non-interference (GNI) [22] that permits programs to release whether secret inputs exist as long as the values of secret inputs remain confidential. Such a definition is better suited for distributed systems, where secret inputs may happen as a reaction to public actions. For instance, a server may query a database storing secret data as a reaction to a public query.

To prove GNIV, we build upon techniques for proving standard GNI [28, 31], where we use the guarantees provided by policy compliance to simplify proofs. More concretely, standard GNI is typically proved by showing how from a trace t, one can iteratively construct an uncertainty trace t' that has the same public behavior as t, but any possible secret data. Instead of the same public behavior, we require only that t' performs the same declassifications as t. Policy compliance ensures that if the declassifications are the same, then the public behavior is the same. In Fig. 1, this proof methodology is referred to as *uncertainty trace construction*.

Outline. We introduce our specification language for policies and define policy compliance in Sec. 3. In Sec. 4, we show how we can combine established verification techniques to verify code. Sec. 5 discusses how we validate policies. Sec. 6 presents our Wireguard case study and illustrates how we express established policy specification patterns based on examples from previous works. The section also lists our trust assumptions. Sec. 7 discusses related work, and Sec. 8 concludes.

3 Security Policies

This section presents the representation of I/O behavior (Sec. 3.1), our policy specification language (Sec. 3.2), and our threat model (Sec. 3.3). Lastly, Sec. 3.4 defines when policies are satisfied.

To illustrate our policy framework and its application, we use a small running example throughout this paper. Consider a protocol to query a person's remaining vaccine protection duration from a server: Every user has a pre-established id and private and public key $k_{\text{sk}}, k_{\text{pk}}$. First, a user sends their id. Next, the application sends a challenge n. The user signs the challenge with their private key. Finally, if the challenge was successful, the application sends the remaining protection duration encrypted with the public key. The server acquires the requested data by querying a database, which also returns the client's public key and a list of compatible vaccines. Below is an informal description of the protocol. For the sake of brevity, we simplify messages by omitting addresses, tags, and additional ids.

$$\begin{aligned}
\text{Client} &\to \text{Server}: && id \\
\text{Server} &\to \text{Client}: && n \\
\text{Client} &\to \text{Server}: && \text{enc}(n, k_{\text{sk}}) \\
\text{Server} &\to \text{Database}: && id \\
\text{Database} &\to \text{Server}: && (k_{\text{pk}}, \mathit{date}, \mathit{vaccines}) \\
\text{Server} &\to \text{Client}: && \text{enc}(\mathit{date}, k_{\text{pk}})
\end{aligned}$$

Figure 2: A deployment diagram for the running example. Network communication happens via the receive and send action. The action query **communicates with the database.**

3.1 I/O Behavior

A program's I/O behavior captures all communication with the program's environment. The I/O behavior of a program execution is represented as a sequence of *I/O actions*, which are executions of communication primitives, such as sending or receiving a message. We refer to sequences of I/O actions as *traces*.

I/O actions provide a language-independent representation of a program's I/O behavior [38, 39, 55]. To reason about a program in a specific language, I/O actions can be linked to the I/O library of that language by providing (trusted) specifications to the library methods expressing which I/O action is performed by a method.

I/O Behavior of Programs. I/O actions have the form $N(x, \underline{r})$ for an action name N, an output x forwarded to the environment, and an input r obtained from the environment. E.g., the actions $\text{send}(1, \blacksquare)$ and $\text{recv}(\blacksquare, \underline{0})$ represent sending a 1 and receiving a 0, respectively. We use a designated default value $\blacksquare$ if an action does not have an output or input. We omit $\blacksquare$ arguments when they are clear from the context.

A program's *I/O behavior* is the set of traces of all partial program executions. Considering *partial* executions allows us to represent a non-terminating program execution via the set of finite prefixes of its infinite traces. As a consequence, I/O behaviors are always prefix-closed. For brevity, we sometimes omit partial executions that follow from prefix-closedness, e.g., we write $\{\text{recv}(\underline{0}) \cdot \text{send}(1)\}$ instead of $\{\epsilon, \text{recv}(\underline{0}), \text{recv}(\underline{0}) \cdot \text{send}(1)\}$, where $\cdot$ and ϵ denote concatenation and the empty trace, respectively. Concurrent programs are also represented by a single set of traces. For instance, the trace $\text{recv}(\underline{0}) \cdot \text{recv}(\underline{5}) \cdot \text{send}(6) \cdot \text{send}(1)$ may represent an execution where two threads each increment and forward a received number, namely 0 and 5, respectively.

Example 1. Fig. 2 shows a deployment diagram for our running example. The application communicates with clients via the network and communicates with an external database via remote procedure calls. The action send represents sending a packet over the network, where the action's output is the sent packet. Similarly, the input of the action recv is the received packet. For brevity, a network packet is only the message payload, represented as a bytestring. Network addresses and headers are omitted. The action query represents a

remote procedure call to the external database. The action's output is a user id and the action's input is the stored medical record r, consisting of the public key r_{key}, the protection date r_{date}, and the compatible vaccines r_{vacs}. A trace of a single sequential execution of the protocol may look as follows, where we use enc to denote the value obtained by encrypting a given value with a given key:

$$\mathsf{recv}(\underline{201}) \cdot \mathsf{send}(13) \cdot \mathsf{recv}(\underline{\mathsf{enc}(13, k_{\mathsf{sk}})}) \cdot \\ \mathsf{query}(201, \underline{((k_{\mathsf{pk}}, \texttt{02.02.22}, \mathsf{v}))}) \cdot \mathsf{send}(\mathsf{enc}(\texttt{02.02.22}, k_{\mathsf{pk}}))$$

3.2 Policy Specifications

Security policies classify the sensitivity of data and define what data may be declassified, and when. A successful approach for language-based policy frameworks is to specify (1) classification by annotating library methods to express the sensitivity of inputs and outputs, and (2) when declassification is permitted as a condition on the global state of a program [9, 35, 48, 50]. For instance in our running example, we may permit declassifying enc(*date*, *key*) if (*date*, *key*) is stored in a designated queue used by the implementation to store past queries.

To obtain language-agnostic policies, we lift this approach to I/O behaviors. For classification, we specify pre- and postconditions for I/O actions, which we refer to as *classification spec*. To reason about declassification, we extend I/O behaviors by *declassification actions* of the form $\mathsf{decl}(x)$, representing the declassification of value x. We call this extension *IOD behaviors*. To be language-agnostic, we specify when declassification is permitted as a condition on the trace of produced I/O actions and declassification (instead of the language and implementation-specific program state). We capture these conditions formally by specifying the set of traces of I/O actions and declassification actions that an implementation may produce, which we refer to as *IOD spec*. Our security policies are then a tuple of a classification spec and an IOD spec.

Definition 3.1 (Security Policy)**.** A security policy (Σ, R) is a tuple of a classification spec Σ and an IOD spec R.

Expressing security policies on the level of IOD behaviors allows us to abstract from concrete computations and data representations and, thus, to express policies independent of a concrete implementation or programming language. Prior work demonstrated that traces of I/O actions [1, 52] can nevertheless express the behavior of stateful distributed systems. In particular, specifications of traces can refer to results of computations by using mathematical functions. For instance, a specification $\mathsf{recv}(\underline{x}) \cdot \mathsf{send}(\mathsf{hash}(x))$ represents all executions that receive a value x and then send the hash of x, where hash is a mathematical function describing the result of a hashing algorithm, e.g., SHA-256, without referring to its concrete code implementation. The formal connection between the mathematical function and the code implementation is established during code verification. This approach works for any stateful computation whose result can be described as a mathematical function of prior inputs. By abstracting from concrete states, our policies cannot express declassification based on arbitrary program state, which is not visible in the IOD behavior. However, none of our case studies required this expressiveness.

$$a ::= \mathsf{true} \mid \mathsf{Low}(e) \mid a \wedge a \mid e \Rightarrow a$$

Figure 3: Assertion language for pre- and postconditions. We use a and e to range over assertions and expressions.

3.2.1 Classification Spec. A classification spec expresses sensitivity requirements and guarantees for data via pre- and postconditions for both I/O and declassification actions. E.g., the precondition of the send action may express that the sent payload must be low. We express this specification using the Hoare triple $\{\mathsf{Low}(x)\}\mathsf{send}(x)\{\mathsf{true}\}$, where $\mathsf{Low}(x)$ specifies that x has low sensitivity. Analogously, we can use a postcondition to express that received payloads are assumed to be low: $\{\mathsf{true}\}\mathsf{recv}(\underline{r})\{\mathsf{Low}(r)\}$. For simplicity, we limit sensitivity levels to $\{\mathsf{Low}, \mathsf{High}\}$, where High and Low specify that data is confidential and not confidential, respectively. An extension to arbitrary sensitivity lattices is straightforward [36].

The pre- and postconditions of a classification spec are expressed in the assertion language shown in Fig. 3. Assertions may combine sensitivity with logical constraints, for instance, to express sensitivity depending on the values of inputs and outputs. E.g., for the action $\mathsf{query}(id, \underline{r})$, the postcondition $\mathsf{Low}(r_{\mathsf{key}}) \wedge (id \in \mathit{Public} \Rightarrow \mathsf{Low}(r))$ specifies that the public keys are low and if the id is in some fixed set *Public*, then the entire record is low.

Every output or input that is not explicitly specified as low is, by default, considered to be potentially high. For instance, the triple $\{\mathsf{true}\}\mathsf{getKey}(\underline{r})\{\mathsf{Low}(\mathsf{len}(\mathsf{r}))\}$ expresses that the length of the input r is low, whereas other aspects of r, such as the actual content, are potentially high.

For declassification, the triple $\{\mathsf{Low}(p)\}\mathsf{decl}(p, x)\{\mathsf{Low}(x)\}$ expresses that after declassification, we may assume that x has low sensitivity, capturing information release. The role of the additional parameter p will be explained in Sec. 3.2.2.

I/O actions may leak information even if all outputs actually have low sensitivity because the occurrence of the action itself reveals information about the control flow in the program. To reason about such indirect information flow, we also classify each I/O action as low or high; the occurrence of low actions can be observed by the attacker and, thereby, must not depend on high data. For simplicity, we assume in this paper that all I/O actions are low, i.e., whether an action occurs must never depend on high data. Our implementation provides an annotation to specify action sensitivity.

Example 2. In our running example, we assume that attackers have access to the network and observe sent and received payloads. Conversely, we consider the communication channel with the database to be secure. Moreover, we consider the stored public keys to be low. For the action query, we specify that the queried id and the public key of the record are low, but the protection date and the compatible vaccines are (implicitly) potentially high: $\{\mathsf{Low}(id)\}\mathsf{query}(id, \underline{r})\{\mathsf{Low}(r_{\mathsf{key}})\}$.

3.2.2 IOD Spec. IOD specs specify all traces permitted by a policy. Since traces include declassification actions, the permitted traces capture what data may be declassified and when. Our policy framework does not prescribe how IOD specs are expressed. In this paper, we use *IOD-guarded transition systems*, which extend the transition

systems by Sprenger et al. [52] with declassification. In our implementation and evaluation, we express IOD specs also in separation logic [39, 52, 55] to leverage existing verification tools.

Definition 3.2 (IOD-guarded transition system)**.** An IOD-guarded transition system is a labeled transition system (S, Act, V, G, U), where S is a set of states, Act is a set of action names, V is a set of output and input values, $G : S \times Act \times V \to \{\top, \bot\}$ is a guard, and $U : S \times Act \times V \times V \to S$ is an update function.

An IOD-guarded system induces the transition relation $\to = \{(s_0, \mathsf{N}(x, \underline{r}), s_1) \mid G(s_0, N, x) \wedge s_1 = U(s_0, N, x, r)\}$. We lift the relation to traces, where $(s, \epsilon, s) \in \to_*$ and $(s, t \cdot \mathsf{N}(x, \underline{r}), s'') \in \to_*$ whenever $(s, t, s') \in \to_*$ and $(s', \mathsf{N}(x, \underline{r}), s'') \in \to$ for some s'. Given a set of initial states $S_0 \subseteq S$, the traces of an IOD-guarded system are all traces t with $(s_0, t, s') \in \to_*$ for some states $s_0 \in S_0$ and $s' \in S$.

It is essential for soundness that an IOD spec prescribes all declassifications *deterministically*. To understand why, consider a situation where an IOD spec permits the declassification of either x or y. This would allow implementations to choose which of the two to declassify. In particular, an implementation could make this choice depending on a secret and, thereby, leak it. To prevent such situations, we require that any declassification of sensitive data in an IOD spec is determined by the previous actions on the trace.

This determinism requirement is sound, but too restrictive in practice. Continuing our hypothetical example from the previous paragraph, applications should have the freedom to declassify either x or y, as long as the choice does not depend on a secret. To allow that, we parameterize our declassification action with an additional parameter p, which can be used as a tag, to distinguish different occurrences of declassification. With this addition, we require the declassified value to be determined by the previous actions on the trace *and the value of* p. By requiring p to be low, implementations cannot choose it depending on a secret, thereby avoiding unintentional leaking. The following definition captures this intuition.

Definition 3.3 (Well-defined IOD spec)**.** An IOD spec R is well-defined if for every trace t and all values x_1, x_2, p,

$$t \cdot \mathsf{decl}(p, x_1) \in R \wedge t \cdot \mathsf{decl}(p, x_2) \in R \Rightarrow x_1 = x_2.$$

Example 3. The following IOD-guarded transition system for our running example permits the declassification of the encrypted server response $\mathsf{enc}(\mathit{date}, k_{\mathsf{pk}})$, where date and k_{pk} are the protection date and public key returned from the database.

$$\mathsf{send}(x) : \top \rhd s \qquad \mathsf{recv}(\underline{r}) : \top \rhd s$$
$$\mathsf{query}(\mathit{id}, \underline{d}) : \top \rhd s[\mathit{id} \mapsto (d_{\mathsf{date}}, d_{\mathsf{key}})]$$
$$\mathsf{decl}(\mathit{id}, x) : \mathit{id} \in s \wedge x = \mathsf{enc}(s[\mathit{id}]_{\mathsf{date}}, s[\mathit{id}]_{\mathsf{key}}) \rhd s$$

We use the notation $\mathsf{N}(x, \underline{r}) : G(s, N, x) \rhd U(s, N, x, r)$. The state s (of the transition system, not a concrete implementation) is a map from ids to the last-queried key and date. It is changed only in the update of query. The guard for declassification expresses what data may be classified; it uses id as a low parameter to satisfy the well-definedness requirement from Def. 3.3. The guards of all other actions are true. Note how the mathematical function enc lets us describe the effect of a (possibly stateful) computation performed by the implementation, as explained above.

3.3 Threat Model

We consider attackers that know the executed code and can observe the low data of performed I/O. However, they do not have direct access to the machines on which code is run and cannot inspect memory. We do not consider side channels such as timing information. An extension to timing channels is interesting future work and can be tackled by moving from formal guarantees based on possibilistic secure information flow [51] to a probabilistic model [46].

The observational capabilities of attackers are defined by a security policy's classification spec. A program that complies with a given policy is secure against attackers that can observe (at most) the data and actions classified as low.

3.4 Policy Compliance

Whether a program satisfies a security policy is determined entirely over the program's IOD behavior. To satisfy the policy's IOD spec R, the IOD behavior of the program (that is, its set of traces) has to be a subset of R. To satisfy its classification spec Σ, the program's IOD behavior has to satisfy secure information flow, where requirements and assumptions about low sensitivity are specified by Σ. We focus on the latter property in this subsection, in particular, on defining secure information flow as a variation of observational determinism that permits non-deterministic I/O behaviors.

Note that declassification is handled by the combination of both requirements. The IOD spec expresses where a declassification may occur in a trace, and the classification spec of declassification actions ensures that the occurrence of a declassification action does not depend on a secret (see Sec. 3.2.2) and expresses that declassified data has low sensitivity.

3.4.1 Secure Information Flow. We define secure information flow (SIF) based on observational determinism (OD) [33, 43], a widely-used criterion that is supported by standard code verification techniques [24, 25] and, thus, enables us to use off-the-shelf program verifiers to prove that a program satisfies a policy (see Sec. 4).

However, OD has a critical limitation. A program satisfies OD only if its low outputs are deterministic in the low inputs. This definition ensures that low outputs do not depend on confidential data. However, real-world programs usually have non-deterministic IOD behavior, for instance, due to concurrency and memory allocation, such that standard OD is not applicable.

Existing solutions [26, 37] for language-specific frameworks solve this problem by externalizing the non-deterministic choices. That is, they make program executions artificially deterministic by parameterizing the language semantics with an *oracle* that captures the non-deterministic choices of an execution, for instance, how threads have been scheduled or how memory has been allocated. OD is then satisfied if the low outputs are deterministic in the low inputs *and* the oracle. The same technique is also used to reason about applied π-calculus and its extensions, for instance, to prove observational equivalence [29, 44].

We adapt the idea of externalizing non-deterministic choices to IOD behavior. Instead of parameterizing executions with oracles, we extend traces with the information about non-deterministic choices, which we refer to as *extensions*. While language-specific approaches know where non-deterministic choices happen, our

language-agnostic framework does not have this information. Therefore, we allow extensions at any point in the trace, but introduce a soundness condition called *input-closedness* to ensure that traces are not extended incorrectly.

An IOD behavior then satisfies SIF if there exists an input-closed extension that satisfies OD. In the following, we explain extensions and input-closedness.

We provide formal definitions in Sec. 3.4.2.

Extensions. An IOD behavior T' is an extension of T if T' is obtained by adding *auxiliary actions* into T's traces. The added auxiliary actions are not actually produced by the program, but instead are used to justify that a program satisfies SIF.

Consider the program `(send(1) || send(2))`, which sends 1 and 2 in parallel. The IOD behavior of the program is $\{\mathsf{send}(1) \cdot \mathsf{send}(2), \mathsf{send}(2) \cdot \mathsf{send}(1)\}$. The program does not satisfy OD because the sent message is not deterministic. However, the program is secure since the sent message is independent of any confidential data. Extensions allow us to make the secret-independence of non-deterministic behaviors explicit. In our example, a suitable extension introduces an auxiliary action Sched that captures whether the scheduler executes the left or right parallel branch first: $\{\mathsf{Sched}(\underline{\mathsf{L}}) \cdot \mathsf{send}(1) \cdot \mathsf{send}(2), \mathsf{Sched}(\underline{\mathsf{R}}) \cdot \mathsf{send}(2) \cdot \mathsf{send}(1)\}$. Auxiliary actions have their own classification spec. With the spec $\{\mathsf{true}\}\mathsf{Sched}(\underline{x})\{\mathsf{Low}(x)\}$, our extension satisfies OD as the sent messages are now deterministic in the scheduler choice.

To ensure that declassifications happen only if permitted by the policy, auxiliary actions must not implicitly declassify data, that is, their classification specs are not allowed to describe declassifications, e.g., by specifying that an output is low in the postcondition. For simplicity, we enforce that auxiliary actions always have the spec $\{\mathsf{Low}(x)\}\mathsf{N}(x, \underline{r})\{\mathsf{Low}(r)\}$, i.e., auxiliary actions do not take high data and, therefore, never declassify anything (for auxiliary actions that do not have an output, the precondition $\mathsf{Low}(\blacksquare)$ is equivalent to true). This spec is sufficient to verify our case studies in Gobra. Our Isabelle/HOL formalization defines a weaker criterion for the classification spec of auxiliary actions that expresses that specs do not describe declassifications.

Our formal definition of SIF below allows one to choose an extension for each program and policy. To reduce the necessary specification overhead, our implementation fixes the extension and the classification spec for auxiliary actions.

Example 4. We formalized our approach based on SecCSL [26], a logic for a simple concurrent programming language with references. For this programming language, we introduce two auxiliary actions. For concurrency, the action $\mathsf{Sched}(\gamma, \underline{\tau})$ has as input the information which thread is scheduled next. The output is necessary for input-closedness and explained in Example 5. Because extensions contain which thread is scheduled next, the IOD behavior becomes deterministic. For heap state, the action $\mathsf{Init}(\underline{s})$ is added as the first action of a trace and has as input the initial heap memory. For both actions, input and output are classified as low.

Input-closedness. Without restrictions, extensions can trivially invalidate SIF by laundering confidential inputs. Consider the insecure IOD behavior $\{\mathsf{getKey}(\underline{x}) \cdot \mathsf{send}(x) \mid x \in \mathbb{N}\}$, which first gets a confidential key x and then sends it on a public channel. An invalid extension can mask the origin of x by adding an auxiliary action $\mathsf{In}(\underline{x})$ that has the key as low input, e.g., as $\{\mathsf{getKey}(\underline{x}) \cdot \mathsf{In}(\underline{x}) \cdot \mathsf{send}(x) \mid x \in \mathbb{N}\}$. This IOD behavior satisfies OD. However, since the auxiliary action is not present in an actual program execution, it remains insecure.

To prevent auxiliary actions from masking the origin of inputs, we require extensions to be *input-closed*: an extension must contain traces for all possible inputs of actions. The above extension violates this condition because it contains only traces where the inputs from getKey and In are *the same*. If we add traces where both actions receive *different* inputs, the insecurity becomes apparent.

Example 5. We have formally proved that our extensions chosen for SecCSL are input-closed. The action $\mathsf{Init}(\underline{s})$ trivially maintains input-closedness because the IOD behavior contains traces for all possible initial heap memories. For $\mathsf{Sched}(\gamma, \underline{\tau})$, we have to show that every possible input of Sched may actually be scheduled. In SecCSL's language, all unblocked threads may be scheduled; these are captured by the current program configuration γ. Defining the inputs of Sched accordingly requires us to add γ as an output (see Sec. 3.4.2). By classifying Sched's output and input as low, we recover a standard OD reasoning principle for concurrent programs: If we show that control flow is low, and thus the current configuration is low then we may assume that decisions made by the scheduler are low.

3.4.2 Formal Definition of Secure Information Flow. We will now define the concepts introduced in the previous subsubsection.

Extensions. We introduce a projection $\mathsf{actual}(\cdot)$, removing auxiliary actions from traces, e.g., $\mathsf{actual}(\mathsf{In}(\underline{x}) \cdot \mathsf{recv}(\underline{r})) = \mathsf{recv}(\underline{r})$, where $\mathsf{In}(\underline{x})$ is an auxiliary action. We lift the projection to sets of traces. Therefore, an IOD behavior T' is an extension of T if $\mathsf{actual}(T') = T$ holds.

Input-closedness. Def. 3.4 defines formally when a set of traces is input-closed. As discussed, if an action has some input r, then for every possible input r' of that action, there must be a trace where the action has input r'. An input is possible if it occurs in some trace (expressed with $t' \cdot N(x, \underline{r'})$ in Def. 3.4).

Definition 3.4 (Input-Closed IOD Behaviors)**.** A set of traces T is input-closed if

$$t \cdot N(x, \underline{r}) \in T \land t' \cdot N(x, \underline{r'}) \in T \Rightarrow t \cdot N(x, \underline{r'}) \in T$$

Low Data Projection. To define OD, we have to express which data is required and assumed to be low. In our framework, this is the data classified as low by pre- and postconditions, respectively. For this purpose, we introduce two projections $\cdot\downarrow_\Sigma^{\mathsf{pre}}$ and $\cdot\downarrow_\Sigma^{\mathsf{post}}$ which can be applied to actions and remove any data that is *not* low according to the action's classification spec (for the pre- and postcondition, respectively). For the classification spec of our running example, we have for instance:

$$\mathsf{send}(x)\downarrow_\Sigma^{\mathsf{pre}} = \mathsf{send}(x) \quad \mathsf{recv}(\underline{r})\downarrow_\Sigma^{\mathsf{post}} = \mathsf{recv}(\underline{r})$$
$$\mathsf{query}(id, \underline{d})\downarrow_\Sigma^{\mathsf{post}} = \mathsf{query}(\underline{d_{\mathsf{key}}}) \quad \mathsf{decl}(x)\downarrow_\Sigma^{\mathsf{post}} = \mathsf{decl}(x)$$

Recall that $\mathsf{N}(x)$ and $\mathsf{N}(\underline{r})$ are shorthands for $\mathsf{N}(x, \blacksquare)$ and $\mathsf{N}(\blacksquare, \underline{r})$, respectively. We lift the projections from single actions to traces.

Observational Determinism. We build on the definition of OD introduced by Clarkson and Schneider [22]. Their definition does

not consider progress channels, i.e., information may be released by not producing observable actions, e.g., due to an infinite loop. To prevent such information leakage, we require in addition progress sensitivity [3], enforcing the absence of progress channels. In the definitions below, we refer to Clarkson and Schneider's definition as *progress-insensitive OD*.

An IOD behavior satisfies progress-insensitive OD if for every action, the data expected to be low is deterministic in the data assumed to be low for previous actions. Determinism is expressed formally by considering pairs of extended traces.

Definition 3.5 (Progress-Insensitive Σ-OD)**.** A set of traces T satisfies progress-insensitive Σ-OD if, for all of traces t_1, t_2 and actions e_1, e_2 with $t_1 \cdot e_1 \in T$ and $t_2 \cdot e_2 \in T$,

$$t_1{\downarrow}_\Sigma^{\text{post}} = t_2{\downarrow}_\Sigma^{\text{post}} \Rightarrow e_1{\downarrow}_\Sigma^{\text{pre}} = e_2{\downarrow}_\Sigma^{\text{pre}} .$$

The definition of progress sensitivity is analogous. An IOD behavior satisfies progress sensitivity if whether or not a trace makes progress is deterministic in the data previously assumed to be low. Formally, we specify that if one trace has more progress than another trace, i.e., is longer, then the shorter trace can be extended.

Definition 3.6 (Progress Sensitivity)**.** A set of traces T satisfies progress sensitivity for a classification spec Σ if, for all of traces t_1, t_2 and every action e_1 with $t_1 \cdot e_1 \in T$ and $t_2 \in T$,

$$t_1{\downarrow}_\Sigma^{\text{post}} = t_2{\downarrow}_\Sigma^{\text{post}} \Rightarrow \exists e_2.\ t_2 \cdot e_2 \in T.$$

Definition 3.7 (Σ-OD)**.** A set of traces T satisfies Σ-OD if T satisfies progress-insensitive Σ-OD and progress-sensitivity for Σ.

Secure Information Flow. Given all the defined ingredients, we can define SIF as discussed in Sec. 3.4.1. We use $(\Sigma + \text{NoDecl})$ to denote the classification spec, which uses Σ for actual actions and specifies the triple $\{\text{Low}(x)\}\text{N}(x, \underline{r})\{\text{Low}(r)\}$ for all auxiliary actions.

Definition 3.8 (Σ-SIF)**.** A set of traces T satisfies Σ-SIF if there exists an extension T' such that (1) $\text{actual}(T') = T$, (2) T' is prefix-closed, (3) T' is input-closed, (4) T' satisfies $(\Sigma + \text{NoDecl})$-OD.

4 Code Verification

To enable code verification in a given language, we first equip the language and its libraries to record the performed IOD behavior, introduce auxiliary actions to enable us to verify non-deterministic programs, and prove that the resulting extended IOD behaviors are input-closed. This lets us verify that a program in that language satisfies a security policy by proving the two requirements of policy compliance, namely that the IOD behavior satisfies the IOD spec and that the extended IOD behavior satisfies OD.

We illustrate code verification using Gobra [53], an automated code verifier for Go. We provide the necessary background on Gobra in Sec. 4.1. We discuss how we record IOD behavior in Sec. 4.2, and explain how we verify policy compliance for a given program in Sec. 4.3.

4.1 Background on Gobra

Gobra uses a variant of separation logic [42, 49] to reason about memory and concurrency. Each memory location is associated with a *permission*, which is created when the location is allocated. Permissions are held by method executions and transferred between methods upon call and return, but they cannot be duplicated or forged. A method may access a location only if it holds the associated permission. Note that permissions are a notion used only for verification, but not present in the executable program; consequently, they do not incur any run-time overhead.

Which permissions to transfer upon call and return is specified in the callee method's pre- and postcondition, respectively. In such assertions, the permission to a memory location `l`, for instance, a pointer or an array, is denoted as `acc(l)` and includes permission to all locations of an array or fields of a struct. Gobra distinguishes between write permissions `acc(l)` and read permissions, which we will denote as `acc(l,read)`. As the name suggests, reading from and writing to memory location requires that read and write permissions are held, respectively. Verification ensures that writes are exclusive, whereas multiple functions may hold read permissions and, thus, read concurrently.

Functional properties are expressed via standard assertions, including side-effect free expressions, calls to side-effect free functions (so-called *pure functions*), and *old-expressions*, which refer to the value that an expression had right before a method was called.

It is often useful to instrument programs with additional state and operations for the sole purpose of verification, for instance, to track data about a program execution that is not explicitly maintained by the program. Such verification-only code is called *ghost code* [27] and gets removed during compilation. Gobra ensures that ghost code cannot affect the execution of non-ghost code to ensure that erasing ghost code does not change the program semantics. For instance, an attempt to assign a ghost variable to a non-ghost variable is rejected by the tool.

Gobra previously did not support OD reasoning. We implemented support for standard OD reasoning based on an existing product construction [24, 25], which simulates two executions of the input program by a single execution of the constructed product program, which can then be verified by off-the-shelf verifiers such as Gobra. Our extended version of Gobra supports the *relational low assertion* `low(·)` [11, 25]; `low(e)` expresses that the value of the expression `e` is low according to OD (as formalized in Sec. 3.4.2). As such, `low(e)` holds if `e`'s value is deterministic in the values known to be low.

4.2 Recording IOD Behavior

For every programming language, we need to define how we abstract executions of programs in that language to IOD traces. As discussed in Sec. 3.1, we assume that I/O actions are performed by a set of trusted library methods. For declassification, we add a *ghost* method to the I/O library that is called to declassify data. In order to reason about the IOD behavior of a program, we use an existing specification technique [9, 35]: we record the produced IOD trace of a program execution explicitly *in ghost state*. To this end, we add a global trace data structure to the program, which is accessed via a ghost pointer `Trace`. This trace is initially empty and gets extended by library methods that produce an I/O or declassification action. To describe this effect, we equip each such method with a specification that expresses which action it appends to the recorded trace. Using this technique, the abstraction of a program execution

```
1  req acc(Trace) && acc(msg)
2  ens acc(Trace) && *Trace = old(*Trace) · send(Abs(msg))
3  func Send(msg []byte)

5  req acc(Trace)
6  ens acc(msg)
7  ens acc(Trace) && *Trace = old(*Trace) · recv(Abs(msg))
8  func Recv() (msg []byte)

10 req acc(Trace)
11 ens acc(Trace) && *Trace = old(*Trace) · decl(p, value)
12 ghost func Declassify(p, value any)
```

Figure 4: Specifications of library methods producing IOD actions. Specifications about `Trace` describe an atomic effect. Data is declassified via the `Declassify` ghost method. `Abs` returns the bytes stored in an array.

to an IOD trace is explicitly available for verification, but does not incur any run-time overhead since the trace structure is ghost code and will be erased during compilation.

Fig. 4 shows the specification of some library methods, including declassification. Recall that we simplify IOD actions for the sake of brevity, so arguments such as target addresses are omitted. The preconditions, preceded by `req`, specify that permissions to the ghost pointer `Trace` and the parameter array `msg` is transferred to the method. The postconditions, preceded by `ens`, specify that the permission to the global trace is returned to the caller, such that the caller can use it for subsequent calls. Moreover, the postconditions express which action has been appended to the trace. The expression `old(*Trace)` denotes the value of the trace before the call. The function `Abs` returns the sequence of bytes stored in an array. For instance, `*Trace = old(*Trace) · send(Abs(msg))` expresses that at some point during the call, the trace is appended with the action `send(Abs(msg))`.

As discussed in Sec. 3.4, (extended) traces include, besides I/O and declassification actions, auxiliary actions that describe nondeterministic choices. These choices are typically taken implicitly by the language semantics, without invoking an explicit operation. For instance, the action Sched occurs whenever the run-time system schedules a different thread. For each programming language, we define which auxiliary actions are produced and prove, based on the language semantics, that the resulting extended IOD behavior is input-closed. In contrast to I/O and declassification actions, we do *not* record auxiliary actions on the global trace. Instead, we prove that any requirement imposed on auxiliary actions is indeed enforced by the used verification logic wherever such an auxiliary action may occur. For instance, $\mathsf{Sched}(\gamma, \underline{\tau})$ requires that the current program configuration is low, which is enforced in Gobra and SecCSL by requiring that all branch conditions are low.

In summary, our ghost trace records the I/O and declassification actions performed by a program execution, but not the auxiliary actions, which are handled differently. Since program verification proves properties of a program for all possible executions and, in particular, for all possible values of our ghost trace, verification captures the program's entire IOD behavior (except auxiliary actions). Note that the entire machinery to maintain the ghost trace and to handle auxiliary actions needs to be set up once for a programming language and can then be re-used for the verification of each program written in that language.

4.3 Verifying Policy Compliance

To verify that a program satisfies a policy, we have to prove that (1) the program's IOD behavior satisfies the IOD spec and (2) the program's *extended* IOD behavior satisfies Σ-OD for the classification spec (together with input-closedness proved for the programming language, Σ-OD implies Σ-SIF). In this subsection, we explain how we specify security policies in Gobra, discuss how we verify these two properties, and illustrate verification on our running example.

Specifying Security Policies in Gobra. We express security policies in Gobra as implementations of an interface `Policy`. This interface is defined in a re-usable library and prescribes three functions that need to be defined for each concrete security policy (see top of Fig. 5). This library also contains a representation of actions, the states of the IOD transition system, assertions, and specifications as terms of an algebraic datatype (ADT), together with functions that yield these terms. For instance, `True()` and `Low(e)` are function calls that yield terms for the assertions `true` and `Low(e)`, respectively. Our library defines a function for each assertion of the assertion language defined in Fig. 3. Similarly, the call `Spec{P,Q}` yields a tuple term consisting of the precondition `P` and the postcondition `Q`.

Lines 7–22 in Fig. 5 show the implementation `Vac` of the `Policy` interface for our running example. The classification spec of a security policy is captured by the function `Classification`. Lines 7–11 express the classification spec for our running example as discussed in Example 2. The `Classification` function takes an action and returns its spec, consisting of a precondition and a postcondition. The function uses pattern matching to distinguish the different actions. The prefix `?` binds matched arguments. For instance, the pattern `query{?id,?f}` matches the action `query` and binds its output and input to the variables `id` and `f`, respectively. For instance, the case for `query{?id,?f}` expresses that the action requires `id` to be low and ensures that the key of the resulting record `f` is low. The default case at Line 11 handles receive and send actions. The ADT destructors `.out` and `.in` return an action's output and input, respectively.

The IOD-guarded transition system defining the IOD spec of a security policy is expressed via the functions `Guard` and `Update` (Lines 13–22). The guard function takes the state of the transition system and an action, and yields whether the action is enabled in that state. The update function also takes a state and an action and updates the state. Both function definitions correspond directly to the IOD transition system presented in Example 3.

Verifying the IOD Spec. As we discussed in Sec. 4.2, our global trace records all actions performed by a program. Therefore, we can prove that a program's IOD behavior satisfies the IOD spec by showing that the recorded trace is one of the traces induced by the IOD-guarded transition system (see Sec. 3.2.2). In other words, we need to prove that there exists a state in the transition system that is reachable by performing the actions in the recorded trace. This property holds trivially at the program start, when the recorded trace is empty. We impose a proof obligation that this property is

```
interface Policy {
  pure Classification(Action) Spec
  pure Guard(State,Action) bool
  pure Update(State,Action) State
}
```

```
pure func (Vac) Classification(action Action) Spec {
  return (match action {
  case query{?id,?f}: Spec{Low(id),    Low(f.key)}
  case decl{?id,?x}:  Spec{Low(id),   Low(x)}
  case ?a:            Spec{Low(a.out),Low(a.in)} })}
```

```
pure func (Vac) Guard(st State, action Action) bool {
  return (match action {
  case decl{?id,?x}:
      id ∈ st && x = enc(st[id].date,st[id].key)
  case _: true })}

pure func (Vac) Update(st State, action Action) State {
  return (match action {
  case query{?id,?f}: st[id := {f.date,f.key}]
  case _: st })}
```

Figure 5: The `Policy` interface (top segment) prescribing the functions that need to be implemented to define a concrete security policy in Gobra. The two bottom segments specify the security policy for our running example.

preserved whenever the trace is extended (that is, when an I/O or declassification action is performed).

To encode this approach, our reusable library defines a function `Reaches(`p`, `t`, `s`)` to express that t is a trace of the IOD spec defined by the policy p and reaches state s from the initial state. Conceptually, we impose a proof obligation $\exists$ *st* :: `Reaches(p, *Trace, `*st*`)` for each operation that performs an action to check that the trace extended by the performed action is still permitted by the IOD-guarded transition system (here, `p` is the instance of the security policy). In practice, we avoid the existential quantifier by storing the transition system state explicitly in a ghost variable and updating it using the `Update` function of the policy whenever an action is performed. This allows us to instantiate the existential quantifier directly and, thereby, avoid a well-known weakness of SMT solvers.

Imposing proof obligations whenever an action is performed (instead of checking that the trace is permitted at the end of the program) leads to simpler proof obligations and works for non-terminating programs. However, this approach cannot verify programs that branch on a secret, but perform equivalent actions in both branches, e.g., `if h {Send(1)} else {Send(1)}`, where `h` is confidential. This limitation is not relevant for our code verification in Gobra and SecCSL, where we disallow branching on secrets anyway.

Verifying Observational Determinism. To prove that a program satisfies OD, we have to prove progress-insensitive Σ-OD (Def. 3.5) and progress sensitivity (Def. 3.6). The latter is trivial in our setting: since we do not allow branching on secrets, the termination of loops and calls (that is, progress) cannot depend on a secret.

To verify progress-insensitive Σ-OD, we have to prove that the arguments of each action on the recorded trace that are classified as low by the action's preconditions are deterministic in the arguments of the previous actions on the trace that are classified as low by their postconditions. As for verifying the IOD spec, this property holds trivially for the empty trace and we check that this property is preserved whenever the trace is extended. Before performing an action, we assume `low(*Trace`$\downarrow_\Sigma^{\text{post}}$`)` and then check after the action that `low(*Trace`$\downarrow_\Sigma^{\text{pre}}$`)` holds. Our reusable library defines functions `Pre(`p`,`t`)` and `Post(`p`,`t`)` to express the low data projections $t\downarrow_\Sigma^{\text{pre}}$ and $t\downarrow_\Sigma^{\text{post}}$, respectively.

Concurrency Reasoning. As we have seen so far, our proof obligations for code verification are expressed in terms of the recorded trace, which is stored in a mutable ghost data structure. Standard separation logic ensures that mutable state is exclusively owned: only one method can hold the permission to the data structure at any point in the execution. This is problematic for concurrent implementations, where multiple threads may perform IOD actions and, thus, need mutable access to the trace.

In Gobra, we solve this problem using *shared invariants* [30], which express assertions that always hold. Data structures that are governed by a shared invariant may be updated concurrently under two conditions: (1) the update preserves the shared invariant and second, (2) the update is performed atomically (such that other threads cannot observe intermediate states in which the shared invariant does not hold).

For our ghost trace, we use the shared invariant shown at the top of Fig. 6. It provides the permission to access the ghost trace, and expresses that the current trace is compatible with the IOD transition system and its pre-projection is low. Performing an IOD action satisfies the two conditions above: (1) adding a new action to the trace preserves the shared invariant; (2) since these updates affect only ghost state, they can be treated as atomic.

Running Example. Fig. 6 shows our running example in Gobra. Lines 5–18 show snippets of an implementation together with some of the required proof annotations. The shown snippet can be run in parallel by multiple threads.

At Line 5, the trace is extended with a recv action. The permission to `Trace`, necessary for the call, is obtained from the shared invariant. The invariant holds trivially after Line 5 as recv's precondition and guard are `true` and its update keeps the state unchanged. At Line 6, before the next action and after having verified `low(Pre(Vac{}))`, we assume `low(Post(Vac{}), *Trace)`, establishing that the received message is low. The other calls are verified similarly.

Line 7 checks whether the request is already processed. The IOD spec specifies that we may declassify only the most recently queried data for `id` and, thus, use it for a reply. The code handles this requirement by letting at most one thread process requests for a specific id. It stops if the request is already being handled (Line 8). Note that requests for different ids can be processed in parallel.

The precondition of the query action (Line 10) requires that `Abs(id)` is low, which we get from Line 6. Furthermore, the transition system's state gets updated with the queried date and key (Line 12). As discussed, we use the transition system's `Update` function to keep track of the new state.

Line 14 encrypts the date with the key. The specification of the method `Encrypt` shown at Lines 20–23 relates calls of the `Encrypt` method to the mathematical function enc, which illustrates how we connect concrete computations to the abstract state tracked in an

```
1  pred SharedInvariant() {
2    acc(Trace) && low(Pre(Vac{},*Trace)) &&
3    ∃ st :: Reaches(Vac{}, *Trace, st)
4
5  id := Recv()
6  // assume low(Post(Vac{}), *Trace)
7  already_processed := queue.add(id)
8  if already_processed { /* stop */ }
9  ...
10 info := Query(id)
11 // assume low(Post(Vac{}), *Trace)
12 // update transition system state
13 ...
14 reply := Encrypt(info.date, info.key)
15 ghost Declassify(Abs(id),Abs(reply))
16 // assume low(Post(Vac{}), *Trace)
17 Send(reply)
18 queue.remove(id)
19
20 req acc(data, read) && acc(key, read)
21 ens acc(data, read) && acc(key, read) && acc(ciph)
22 ens Abs(ciph) = enc(Abs(data), Abs(key))
23 func Encrypt(data, key []byte) (ciph []byte)
```

Figure 6: Verification of the running example in Gobra (Lines 5–18) together with the used shared invariant (Lines 1–3). Lines 20–23 shows the specification of the encryption method, relating calls to the mathematical function enc**. Proof annotations are indicated with the keyword ghost.**

IOD spec. After the call, we get from `Encrypt`'s postcondition that `Abs(reply)` is equal to enc(`Abs(info.date)`, `Abs(info.key)`). The guard of the declassification (Line 15) holds since we declassify the most recent date encrypted with the most recent key. As a technicality, ensuring the absence of other queries since Line 10 requires some concurrency reasoning, which we omitted to focus on the essentials. The call to the `Send` method at Line 17 completes the request. We can show that the sent payload is low due to the assumption gained from the declassification at Line 16.

5 Policy Validation

Like code, security policies may contain errors due to human failure. The aim of validating a security policy is to increase the confidence that the policy specifies the intended security requirements. We validate policies by proving properties that hold for all IOD behaviors satisfying the policy. In this section, we focus on validating that a policy does not permit the release of information that is intended to remain confidential.

Consider an incorrect variant of our running example's IOD spec (Example 3), where a declassification action $\mathsf{decl}(id, x)$ is permitted whenever x is the encryption of the date and key queried for *id if* (instead of *and*) *id* has been queried. This IOD spec is bad since it permits the declassification of any value, e.g., the confidential list of compatible vaccines, if the id has not been queried yet.

We use two approaches to validate policies: (1) Since implementations refine a policy's IOD spec R, any trace property P satisfied by the IOD spec is also satisfied by the implementation. E.g., for our running example, we may prove that a declassification is permitted only if the id has been queried beforehand. (2) The combination of IOD spec and classification spec enables us, instead of just validating properties about when declassification is permitted, to prove directly that specific data remains confidential even in the presence of declassification. Approach (1) requires standard reasoning about transition systems; we focus on Approach (2) in this section.

We formalize the property that data remains confidential as Generalized Non-Interference Modulo Views (GNIV). As mentioned in Sec. 2, GNIV is an adaptation of generalized non-interference [22] (GNI) that can handle distributed systems. We first discuss the definition of GNIV for passive attackers (Sec. 5.1) and show how to prove it (Sec. 5.2). Afterwards, we extend GNIV to active attackers (Sec. 5.3). As defined in Sec. 3.3, passive attackers are able to observe the low data of all performed I/O actions. Active attackers are in addition able to change the low inputs of actions.

5.1 Definition of GNIV

Declassifications are able to release information about high data. Therefore, not all data classified as high by the classification spec remains confidential. To capture the intended confidentiality, we use a function Λ, referred to as *view*, that takes a trace and returns the data that we want to prove remains confidential in the presence of declassification. We lift Λ to sets of traces, denoted with $\cdot\downarrow_\Lambda$.

Before we discuss the definition of GNIV, we first illustrate why standard GNI is ill-suited for distributed systems. GNI is satisfied by an IOD behavior if for every pair of traces $t_1, t_2 \in T$, there exists a trace t_u with the same low data as t_1 and the same secret as t_2 (i.e., $\Lambda(t_u) = \Lambda(t_2)$). We refer to t_u as the *uncertainty trace*. As mentioned before, the issue is that GNI rules out IOD behaviors that let an attacker learn whether a secret input exists, regardless of whether the attacker actually is able to learn the value of the secret input itself. Consider the program below and a view $\Lambda_0 : t \mapsto t\downarrow_{\mathsf{getKey}}$, where $t\downarrow_{\mathsf{getKey}}$ denotes the sequence of inputs of getKey actions occurring in the trace t. In the code, getKey actions are produced by calls to the `GetKey` method:

```
h1 := GetKey(); if Recv() { h2 := GetKey(); }
```

The program does not satisfy GNI for Λ_0 because the low input received from `Recv` implies whether a second getKey action happens. More formally, there exists no uncertainty trace that has the same secrets as the trace $\mathsf{getKey}(\underline{h_1}) \cdot \mathsf{recv}(\mathsf{true}) \cdot \mathsf{getKey}(\underline{h_2})$ but also the same low data as the trace $\mathsf{getKey}(\underline{h_1'}) \cdot \mathsf{recv}(\mathsf{false})$ for all values h_1, h_2, h_1'. GNI fails because the number of getKey actions is different depending on a trace's low data. However, we consider the program secure since no information about the inputs of getKey actions is released (the program does not even use these inputs).

Our definition of GNIV solves this issue by adapting GNI in two ways. First, when comparing the uncertainty trace and trace t_2, we consider secrets that are intended to remain confidential according to a view Λ. Second, we compare the two traces only up to the common number of secret inputs by allowing the uncertainty trace to have more or fewer actions, accounting for different numbers of secret inputs:

Definition 5.1 (Compatibility)**.** An uncertainty trace t_u is compatible with a secret h for a view Λ, denoted as $t_u \#_\Lambda h$, if there exists a trace t' with $\Lambda(t') = h$ and $t_u \leq t' \vee t' \leq t_u$.

For our example, the uncertainty trace $\mathsf{getKey}(\underline{\mathsf{h1}}) \cdot \mathsf{recv}(\underline{\mathsf{false}})$ is compatible with the secret of trace $\mathsf{getKey}(\underline{h_1}) \cdot \mathsf{recv}(\mathsf{true})$.

For the definition of GNIV, to express a trace's low data, we use the projection $\cdot\downarrow_\Sigma$, combining $\cdot\downarrow_\Sigma^{\text{pre}}$ and $\cdot\downarrow_\Sigma^{\text{post}}$, e.g., $\mathsf{query}(id, \underline{d})\downarrow_\Sigma = \mathsf{query}(id, \underline{d_{\mathsf{key}}})$.

Definition 5.2 (GNIV)**.** An IOD behavior T satisfies GNIV for a classification spec Σ and view Λ, if for every secret $h \in T\downarrow_\Lambda$ and every low data $l \in T\downarrow_\Sigma$, there exists an uncertainty trace $t_u \in T$,

$$t_u\downarrow_\Sigma = l \wedge t_u \#_\Lambda h.$$

5.2 Proving GNIV

To prove that all IOD behaviors satisfying a policy also satisfy GNIV (for some view), we construct an uncertainty trace step by step. For every trace t of the IOD spec and every possible secret h (according to the view), we show that there is an uncertainty trace t_u that has the same declassifications as t but is also secret compatible with h.

We formalize the step-by-step construction as a *trace construction plan*, a function that takes the secret h, the uncertainty trace constructed so far, the low data (in particular, the declassifications) that still have to be constructed, and the next action $\mathsf{N}(x, \underline{r})$, and returns the input r' that replaces r to create the uncertainty trace. E.g., for our previous example with getKey, a suitable plan ξ satisfies $\xi(h_1 \cdot h_2, l, \epsilon, \mathsf{getKey}(\underline{h_1'})) = h_1$, i.e., the plan replaces the secret input h_1' of the first getKey action (the trace constructed so far is empty) with the first value h_1 of the sequence of secrets $h_1 \cdot h_2$ that the uncertainty trace has to be compatible with. The low data l that still has to be constructed is either $\mathsf{recv}(\underline{\mathsf{true}})$ or $\mathsf{recv}(\underline{\mathsf{false}})$.

To prove that a policy entails GNIV for a view, we have to show that there exists a trace construction plan that satisfies three conditions: (1) The plan must be well-defined in the sense that it does not change low data and does not return impossible inputs. (2) The plan must be *secret-compatible*, meaning that created uncertainty traces are actually compatible with secret h. (3) The plan must be *declassification-compatible*, meaning that created uncertainty traces are permitted to declassify the same values as the original trace.

Theorem 1 (Passive Attacker Security)**.** *Given a program's IOD behavior T that satisfies a security policy (Σ, R). The IOD behavior T satisfies GNIV for the classification spec Σ and a view Λ, if there exists a well-defined plan ξ that is secret- and declassification-compatible.*

We next define the three conditions formally.

Well-Defined Plans. Well-definedness of plans is straightforward. Since only inputs are modified by a plan, data classified by preconditions remains unchanged trivially. The condition $\exists t'.\ t' \cdot \mathsf{N}(x, \underline{r'}) \in R$ captures that the returned input must be an actual input of the action.

Definition 5.3 (Well-defined Plan)**.** A plan ξ is well-defined for a policy (Σ, R), if

$$t \cdot \mathsf{N}(x, \underline{r'}) \in R \wedge r' = \xi(h, l, t, \mathsf{N}(x, \underline{r})) \\ \Rightarrow (\exists t'.\ t' \cdot \mathsf{N}(x, \underline{r'}) \in R) \wedge \mathsf{N}(x, \underline{r'})\downarrow_\Sigma^{\text{post}} = \mathsf{N}(x, \underline{r})\downarrow_\Sigma^{\text{post}}.$$

Secret-Compatible Plans. Every trace produced by a plan must be compatible with the secret h. Since the empty trace is always compatible with h, i.e., $\epsilon \#_\Lambda h$, we only require that appending the next modified action maintains compatibility.

Definition 5.4 (Secret-Compatible)**.** For a policy (Σ, R) and a view Λ, a plan ξ is secret-compatible if for every secret $h \in R\downarrow_\Lambda$, every trace t and action $\mathsf{N}(x, \underline{r})$ with $t \cdot \mathsf{N}(x, \underline{r}) \in R$, and every l, r',

$$t \#_\Lambda h \wedge r' = \xi(h, l, t, \mathsf{N}(x, \underline{r})) \Rightarrow t \cdot \mathsf{N}(x, \underline{r'}) \#_\Lambda h$$

Declassification-Compatible Plans. GNIV requires that all permitted declassifications are not influenced by the view. Thus, if a declassification is permitted in the original trace, then the same declassification must be permitted in the constructed trace.

Formally, for every prefix t constructed by a plan, whenever a declassification $\mathsf{decl}(p, x)$ is the next action, then the declassification must be permitted after t, i.e., $t \cdot \mathsf{decl}(p, x) \in R$. To quantify over constructed prefixes, we define the image $\mathsf{Img}_{\Sigma,R}(\xi, h, l_p, l_c)$ of a plan ξ as the set of all prefixes that may be constructed by ξ for the secret h and the low data l_p and l_c of the prefix and continuation, respectively.

Definition 5.5 (Plan Image)**.** We define the image of a plan ξ inductively via

$$\epsilon \in \mathsf{Img}_{\Sigma,R}(\xi, h, \epsilon, l_c)$$

$$t \in \mathsf{Img}_{\Sigma,R}(\xi, h, l_p, l_n \cdot l_c) \wedge t \cdot \mathsf{N}(x, \underline{r}) \in R \wedge \mathsf{N}(x, \underline{r})\downarrow_\Sigma = l_n \\ \Rightarrow t \cdot \mathsf{N}(x, \underline{\xi(h, l_c, t, \mathsf{N}(x, \underline{r}))}) \in \mathsf{Img}_{\Sigma,R}(\xi, h, l_p \cdot l_n, l_c)$$

Definition 5.6 (Declassification-Compatible)**.** For a policy (Σ, R) and a view Λ, a plan ξ is declassification-compatible if for every secret $h \in R\downarrow_\Lambda$ and every low data $l_p, \mathsf{decl}(p, \underline{x}), l_c$,

$$\forall t.\ t \in \mathsf{Img}_{\Sigma,R}(\xi, h, l_p, \mathsf{decl}(p, \underline{x}) \cdot l_c) \wedge (\exists x'.\ t \cdot \mathsf{decl}(p, x') \in R) \\ \Rightarrow t \cdot \mathsf{decl}(p, x) \in R.$$

Note that the condition $t \in \mathsf{Img}_{\Sigma,R}(\xi, h, l_p, \mathsf{decl}(p, \underline{x}) \cdot l_c)$ does not guarantee that a declassification may actually happen after the modified prefix, which we capture with $\exists x'.\ t \cdot \mathsf{decl}(p, x') \in R$.

Example 6. For our running example, we show GNIV for two views. Without additional assumptions, we show that the unused secret data from queries, namely the list of compatible vaccines, remains confidential. A suitable plan ξ_0 replaces (1) all queried vaccines with the secret according to the view and (2) all queried dates with the date that is encrypted in the next declassification. Otherwise, inputs remain unchanged. I.e., $\xi_0(h \cdot hs, l, t, \mathsf{query}(id, \underline{d}))$ returns $d[\mathsf{date} \mapsto \mathsf{Next}(id, \mathsf{I}), \mathsf{vac} \mapsto h']$, where $\mathsf{Next}(id, \mathsf{I})$ returns the date encrypted in the next declassification for id in l and h' is the relevant entry of h. We have to also replace the queried data to prove that subsequent declassifications are permitted.

The plan ξ_0 is trivially well-defined and secret-compatible. The plan ξ_0 is also declassification-compatible because, if a declassification may happen next, then there was a previous query action that was modified accordingly by ξ_0.

Given strong assumptions about encryption, we also show that parts of the queried dates remain confidential. The challenge is that after replacing a queried date, a subsequent declassification declassifies a different ciphertext. We resolve this issue by partitioning queried dates into confidential days and non-confidential milliseconds, where we assume that a plan can change milliseconds to obtain the desired ciphertexts. More formally, we assume that for every public key k, and dates d_0, d_1, we can change the milliseconds of d_1 such that the encryption of d_0 and the modified d_1 with k are

the same. For a Dolev-Yao attacker, this assumption implies that an attacker does not know the private keys. Under this assumption, we prove that days of queried dates remain confidential. A suitable plan replaces the day of queried dates with the targeted secret and replaces the corresponding milliseconds such that subsequent declassification is correct.

Our incorrect variant of the IOD spec from the beginning of this section does not satisfy GNIV for either view. In particular, our defined plans are not declassification-compatible for the incorrect policy. If a declassification may happen next, then we are not guaranteed that there exists a previous query action whose encrypted date and key are being declassified.

5.3 Active Attacker

GNIV is strong enough to provide guarantees against active attackers that are also able to modify the low inputs of all actions. In this subsection, we define *Active-GNIV*, a variation of GNIV for active attackers, and show under which conditions GNIV entails Active-GNIV.

We parameterize Active-GNIV with the set of considered attackers. In our model, active attackers are able to change inputs of actions, e.g., by intercepting messages and modifying the payloads. We formalize an attacker as a function $A : \mathrm{Tr} \times Act \times \mathbb{U} \times \mathbb{U} \to \mathbb{U}$ that takes (1) the trace of past actions and (2) the next action $\mathsf{N}(x, \underline{r})$, and returns an attacker-chosen input r' that replaces r. To reason about attackers, we define the image of an attacker $\mathsf{Attacks}(A, T)$ as the set of traces that are possible under the influence of an attacker A for an IOD behavior T.

Definition 5.7 (Attacker Image)**.** The image of an attacker $A : \mathrm{Tr} \times Act \times \mathbb{U} \times \mathbb{U} \to \mathbb{U}$ is the smallest set that satisfies:

$$\epsilon \in \mathsf{Attacks}(A, T)$$

$$t \in \mathsf{Attacks}(A, T) \wedge t \cdot \mathsf{N}(x, \underline{r}) \in T \Rightarrow t \cdot \mathsf{N}(x, \underline{A(t, \mathsf{N}(x, \underline{r}))}) \in \mathsf{Attacks}(A, T)$$

Active-GNIV is then a variation of GNIV where for every considered attacker, the uncertainty traces must exist under the influence of the attacker. In particular, the uncertainty traces must exist regardless of whether the secret h is possible under the influence of the attacker or not.

Definition 5.8 (Active-GNIV)**.** An IOD behavior T satisfies Active-GNIV for a set of attackers $\mathcal{A}$, a classification spec Σ, and a view Λ, if for every attacker $A \in \mathcal{A}$, secret $h \in T\downarrow_\Lambda$, and low data $l \in \mathsf{Attacks}(A, T)\downarrow_\Sigma$, there exists a trace $t_u \in \mathsf{Attacks}(A, T)$,

$$t_u\downarrow_\Sigma = l \wedge t_u \#_\Lambda h.$$

GNIV entails Active-GNIV if all considered attackers are *low-limited*. Intuitively, an attacker is low-limited if the attacker may only observe and modify low data. We capture this intuition formally by defining that low-limited attackers cannot distinguish traces with the same low data, i.e., if one trace is possible under the attacker, then every trace with the same low data is possible, too.

Definition 5.9 (Low-limited Attacker)**.** An attacker $A : \mathrm{Tr} \times Act \times \mathbb{U} \times \mathbb{U} \to \mathbb{U}$ is low-limited for a classification spec Σ, if

$$\forall T, t \in \mathsf{Attacks}(A, T), t' \in T.\ t\downarrow_\Sigma = t'\downarrow_\Sigma \Rightarrow t' \in \mathsf{Attacks}(A, T).$$

Corollary 1. *Given an IOD behavior T that satisfies GNIV for the classification spec Σ and a view Λ. T satisfies Active-GNIV for the attacker set $\mathcal{A}$ if every attacker $A \in \mathcal{A}$ is low-limited.*

6 Case Study

To show that our policy framework is powerful and applicable to real-world programs, we verified an implementation of the WireGuard protocol against an appropriate security policy defined by us (Sec. 6.2). Before discussing the case study, we first list our trust assumptions (Sec. 6.1). Lastly, we discuss several smaller programs that illustrate how we express specification patterns from previous works (Sec. 6.3). All verified programs are available in our artifact [54].

6.1 Trust Assumptions

As mentioned before, we have fully formalized and proved in Isabelle/HOL an instantiation of our policy framework that uses the SecCSL logic for code verification. To benefit from more automation, we verified the programs discussed in this section using Gobra as shown in Sec. 4. When using Gobra, we make two assumptions: (1) We assume that our annotations for trusted libraries that specify I/O behavior (Sec. 4.2) are satisfied and that the resulting I/O behavior is input-closed. (2) We assume that Gobra is sound, i.e., if Gobra reports a successful verification, then the verified code actually satisfies the provided specifications.

For a security policy to be meaningful, we additionally assume that attackers satisfy our threat model (Sec. 3.3).

6.2 The WireGuard VPN

WireGuard is a widely-used Virtual Private Network (VPN). In the protocol, two agents first establish a secret session key in a handshake phase and then use this key to exchange messages in a transport phase. For our case study, we reuse results from Arquint et al. [1]. They verify that a modified version of WireGuard's official Go implementation [23] refines an I/O spec (without declassifications) generated from a Tamarin [14] model of the protocol.

For a security policy, we defined an IOD spec by extending the I/O spec of Arquint et al. with declassification actions. Furthermore, we defined a classification spec from scratch. We then verified that the implementation of the initiator role from Arquint et al. satisfies this policy. We were able to fully reuse the refinement proof by Arquint et al. To verify the code, we added only additional proof annotations to verify that our added declassification actions are permitted, and to verify secure information flow.

The IOD spec of our security extends Arquint et al.'s I/O spec by declassifications. However, their I/O spec is expressed using separation logic [39, 52] rather than a transition system. In this formalism, the permitted I/O behavior is expressed via a co-inductive separation logic predicate that is parameterized by the current position in and the abstract state of the protocol; these two parameters correspond to the transition system state in IOD transition systems. The body of the predicate consists of a number of cases (conjuncts) that each describe (1) the condition under which an I/O action may take place, (2) the action and its arguments (expressed as separation logic predicates), and (3) the effect of the action on the protocol position and abstract state. These ingredients correspond directly

to the guard and update functions of our IOD transition systems. In fact, Sprenger et al. [52] formally proved the equivalence between I/O specifications in separation logic and I/O-guarded transition systems, and Arquint et al.'s I/O spec is indeed generated from such transition systems. Due to this equivalence, and since our framework does not prescribe how IOD specs are expressed (Sec. 3.2), we were able to re-use Arquint et al.'s I/O spec. Since this spec is expressed in separation logic, it is compatible with Gobra's verification technique. Code verification proceeds analogously to the approach shown in Sec. 4.

Security Policy. For the classification spec, we classify that long-term private keys, generated ephemeral keys, and user messages encrypted during the transport phase are secret. All other inputs and outputs, such as network messages, public keys, and timestamps are low data. Furthermore, we classify that the size of private keys is low, too. To reason about keys, setting up the long-term keys and generating ephemeral keys are captured as I/O actions.

For the IOD spec, to specify which declassifications are permitted, we introduce the sets *Out* and *In* that, for the current point in the protocol, contain the messages that may be sent and are being processed, respectively. Our IOD spec definition derives these sets from the transition system state (resp. the state parameters of the I/O spec expressed in separation logic). We then permit three groups of declassifications: (1) Every message m that may be sent $m \in Out$ may also be declassified. (2) Similarly, for every encryption $\mathsf{enc}(m, k)$ occurring in *Out* and decryption $\mathsf{dec}(m, k)$ occurring *In*, we may declassify whether the encryption or decryption fails. (3) We may declassify whether WireGuard's well-definedness condition holds, namely whether the responder's public key to the power of the initiator's private key or ephemeral key is zero (where the keys are also derived from the transition system state).

Our verification approach is expressive enough to verify the implementation against the security policy. All non-deterministic effects in the program, in particular, concurrency and error handling of network sockets, are handled by our auxiliary actions.

Regarding guarantees for the policy, Arquint et al. proved that the I/O spec satisfies key agreement and forward secrecy, which are preserved by our IOD spec. These guarantees entail that if the protocol is in the transport phase according to the transition system state, then indeed a successful handshake between the actors has been established. This allows the IOD spec to express that declassifications are permitted only in the transport phase by expressing a corresponding precondition for declassification actions in terms of the transition system state, such that they are guaranteed to occur after a successful handshake.

Code Changes. We have taken the implementation from Arquint et al. as is, inheriting their changes to the official Go implementation. The official Go implementation was changed in two ways: (1) To reduce verification effort, DDos protection, load balancing, and metrics were omitted. In particular, load balancing requires complex concurrency reasoning not supported by Gobra. (2) -Cryptographic operations and network operations were moved into trusted libraries. The individual steps processing a connection, i.e., parsing and constructing messages, have remained unchanged.

Statistics. The initiator consists of 345 lines of code (LOC) that we have verified. The security policy consists of 18 LOC for the classification spec and 219 LOC for the IOD spec. The specification mechanism for the IOD spec used by Arquint et al. is more verbose than the one shown in Sec. 4. Out of the 219 lines, 147 lines are generated from the verified Tamarin protocol model and only 27 lines contain relevant information for declassification. To verify that the code satisfies the policy, 714 lines of proof annotations were necessary, 123 of which were added for this work. The lines added for this work are either low assertions, annotations to use the shared invariant, or annotations to prove that the declassification conditions are satisfied. The annotation overhead of proof annotations per line of code is around 2, which is typical for SMT-based deductive verifiers. Verification takes 15 minutes on a Lenovo T480s with an Intel Core i7-8650U and 24 GB of RAM. Compared to Arquint et al., the verification time has increased by 13 minutes. This increase is due to the added secure information flow reasoning.

#	Program	LOC	LOS	LOP	T [s]
1	vaccinations	91	15	150	42
2	vaccinations (quantitative)	91	18	165	75
3	database [9]	116	33	190	76
4	embargoed information [4]	13	20	46	36

Figure 7: Programs used to illustrate expressiveness. We list the number of lines of Go code (LOC), security policy (LOS), proof annotations (LOP), and the average verification time in seconds.

6.3 Expressiveness

To show that our approach supports common specification patterns of security policies, we specified and verified several smaller programs. Fig. 7 depicts statistics about these programs. For robust declassification [56] and secret data over public channels, we took policies from the literature [4, 9] and wrote code implementations in Go (Program #4 and #3 respectively). For declassification with quantitative criteria, we extended our running example (Program #2). We also list our running example (Program #1). We focus our discussion on how we express specification patterns. For details about the verified examples, we refer readers to our artifact [54].

State-Dependent Declassification. In this pattern, declassifications are permitted based on the state of an execution. We use this pattern in our WireGuard case study, where, for instance, we permit declassifications involving user inputs only after a successful handshake. When using IOD-guarded transition systems, we express state-dependent declassification straightforwardly by capturing in the transition system state all relevant information, e.g., the protocol phase. The declassification's guard may then permit declassifications based on the captured state.

Quantitative Criteria. As an extension of state-dependent declassification, declassification with quantitative criteria permits declassification based not only on the actual execution state but also based on past declassifications. Since we treat declassification as actions themselves, we express this specification pattern analogously to state-dependent declassification. As an illustrating example, we extended our running example such that declassifications are permitted at most 10 times per id (Program #2 of Fig. 7).

For this change, we extended the state of our IOD-guarded transition system to also store the number of past declassifications per id, which is then increased in the update of the declassification action.

Robust Declassification. In this pattern, declassification is permitted only for values with high integrity, i.e., trusted data [56]. The aim is to ensure that attackers are not able to influence when and what is declassified in unintended ways. In our framework, we reason about integrity by defining an additional classification spec, specifying which data has high integrity. We then specify in the precondition for declassification that the declassified data has high integrity. For endorsement, i.e., the act of elevating the integrity of data, we add an action endorse analogously to declassification, which is then also governed by the IOD spec. We verify the code for both classification specs. Because integrity is the dual of confidentiality [17], we are able to use our low assertions to also express high integrity. In particular, we do not have to adapt our formalism. As an example, we verified a program by Askarov and Myers [4] where data may be declassified if a received timestamp is older than a specified embargo time (Program #4 in Fig. 7). The received timestamp is endorsed only if it is in the past.

Secret Data from Public Inputs. In our running example, secret data originates from a remote database. However, for many applications, secret data arrives encrypted over the public network. In our framework, we express encrypted sources of secret data by classifying the decryption key as confidential. Classifying the key is sufficient, as all data derived using the confidential key is considered confidential itself. In particular, we are guaranteed that programs do not unintentionally release decrypted payloads. As an example, we verified a program inspired by Banerjee at al. (Program #3). A medical database receives encrypted medical records, parts of which are declassified and forwarded to an auditing company.

7 Related work

We compare our work to other policy frameworks with code verification. We distinguish between top-down approaches [40], which generate code from abstract models, and bottom-up approaches [9, 35, 48, 50], which target existing written code.

Bottom-up Policy Frameworks. Closest to our work, but not language-agnostic, Murray et al. [35] specify declassification policies as a condition on a trace of values and a relational assertion, specifying when and what may be declassified, respectively. To populate the trace, programs are also annotated with specifications capturing how this trace is extended. In contrast to our approach, where traces record IOD actions, programs can add arbitrary values to their traces using program annotations. As a consequence, their declassification policies are more flexible than ours, but policies provide weaker guarantees by themselves without further knowledge about the program annotations. For classification, trusted libraries are annotated directly with pre- and postconditions containing low assertions.

Previous frameworks specify policies similarly. Banerjee et al. [9] permit declassification based on a condition on the global program state. Schoepe et al. [48] and Smith [50] specify a predicate, defining whether a concrete declassification statement is permitted depending on the current and initial program state respectively.

Programs are verified using a relational verification logic [35, 48, 50], a type system [4, 6, 10], or a combination thereof [9].

Definition of Security. Most aforementioned bottom-up frameworks [9, 35, 48] define security based on the epistemic definition introduced by Askarov and Sabelfeld [5], which has been extended in various ways [2, 8, 20, 21]. A program is secure if the *attacker uncertainty*, i.e., the set of secrets compatible with low data, remains unchanged for every execution step of the program, except for declassification. Some frameworks [7, 9, 35] require further that the reduction of attacker uncertainty caused per declassification is bounded based on the program and policy.

Another approach is to define security via a variation of low-bisimulations [32, 51]. The definition considers pairs of executions with equal low data. A program is secure if for every step of one execution, the other execution is able to perform a step that again establishes equal low data. For declassifications, pairs that do not agree on the declassified value are disregarded [6, 45, 50].

An advantage of these epistemic- and simulation-based definitions is that they consider timing-channels. Furthermore, epistemic definitions provide immediate guarantees against attackers. However, these definitions require a fixed language semantics, making them ill-suited for our purposes.

Where-Declassification. As analyzed in detail by Sabelfeld and Sands [47], in contrast to our policies, specifying when and what may be declassified, several approaches specify *where* in a program, declassification is permitted [6, 18, 48, 50]. As mentioned before, Schoepe et al. [48] and Smith [50] permit declassification for concrete declassification statements in the program, thereby describing where declassifications happen. A line of work [18, 19, 34] defines policies by assigning *flow lock specs* to data. A flow lock spec is a set of logical locks that have to be opened to release data over a specific channel. These logical locks are opened through static annotations in the code, capturing the position of relevant places in the code. The work by Menz et al. [34] extends this approach to a higher-order language.

Top-down Policy Frameworks. In the approach proposed by Popescu et al. [40], programs are specified as I/O automata producing I/O actions. For verification, the automata is first checked against a policy and then automatically translated into a functional programming language. Importantly, the translation maintains the guarantees provided by the policy. For classification, policies specify public observations and confidential data of the automata's transitions. For declassification, policies specify under which condition declassification is *not* permitted and how much of the secrets must be protected. Their approach is used to verify a conference management system [41] and a social media platform [15, 16].

8 Conclusion

We have introduced a novel policy framework, where policies are specified and validated at the level of I/O behavior. This abstraction enables us to specify security policies independent of programs and programming languages, and to provide guarantees for all programs satisfying a security policy based on the policy alone. To validate policies, we introduce GNIV, entailing for passive and certain active attackers, that a selection of data remains confidential even in the presence of declassification. For code verification, we verify that

programs satisfy our policies using a combination of standard code verification techniques. Our approach is powerful, compatible with different verification techniques, and applicable to real-world code.

We see multiple possible directions for future work. One direction is to automate proving guarantees provided by policies. In our framework, we prove such guarantees manually in Isabelle/HOL. Another direction is to extend our framework to other versions of secure information flow such as probabilistic non-interference.

Acknowledgements. This work was funded by the Werner Siemens-Stiftung (WSS). We thank the WSS for their generous support of this project.

References

[1] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. 2023. Sound Verification of Security Protocols: From Design to Interoperable Implementations. In *SP*. IEEE, 1077–1093.

[2] Aslan Askarov and Stephen Chong. 2012. Learning is Change in Knowledge: Knowledge-Based Security for Dynamic Policies. In *CSF*. IEEE Computer Society, 308–322.

[3] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *ESORICS (LNCS, Vol. 5283)*. Springer, 333–348.

[4] Aslan Askarov and Andrew C. Myers. 2010. A Semantic Framework for Declassification and Endorsement. In *ESOP (LNCS, Vol. 6012)*. Springer, 64–84.

[5] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *S&P*. IEEE Computer Society, 207–221.

[6] Aslan Askarov and Andrei Sabelfeld. 2007. Localized delimited release: combining the what and where dimensions of information release. In *PLAS*. ACM, 53–60.

[7] Aslan Askarov and Andrei Sabelfeld. 2009. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *CSF*. IEEE Computer Society, 43–59.

[8] Musard Balliu, Mads Dam, and Gurvan Le Guernic. 2011. Epistemic temporal logic for information flow security. In *PLAS*. ACM, 6.

[9] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2008. Expressive Declassification Policies and Modular Static Enforcement. In *SP*. IEEE Computer Society, 339–353.

[10] Gilles Barthe, Salvador Cavadini, and Tamara Rezk. 2008. Tractable Enforcement of Declassification Policies. In *CSF*. IEEE Computer Society, 83–97.

[11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM (LNCS, Vol. 6664)*. Springer, 200–214.

[12] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification. In *LFCS (LNCS, Vol. 7734)*. Springer, 29–43.

[13] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Math. Struct. Comput. Sci.* 21, 6 (2011), 1207–1252.

[14] David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. 2022. Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols. *IEEE Secur. Priv.* 20, 3 (2022), 24–32. https://doi.org/10.1109/MSEC.2022.3154689

[15] Thomas Baeuereiss and Andrei Popescu. 2021. CoSMed: A confidentiality-verified social media platform. *Arch. Formal Proofs* 2021 (2021).

[16] Thomas Bauereiss and Andrei Popescu. 2021. CoSMeDis: A confidentiality-verified distributed social media platform. *Arch. Formal Proofs* 2021 (2021).

[17] Ken Biba. 1977. Integrity Considerations for Secure Computer Systems. (1977).

[18] Niklas Broberg and David Sands. 2006. Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In *ESOP (LNCS, Vol. 3924)*. Springer, 180–196.

[19] Niklas Broberg and David Sands. 2009. Flow-sensitive semantics for dynamic information flow policies. In *PLAS*. ACM, 101–112.

[20] Niklas Broberg, Bart van Delft, and David Sands. 2015. The Anatomy and Facets of Dynamic Policies. In *CSF*. IEEE Computer Society, 122–136.

[21] Andrey Chudnov and David A. Naumann. 2018. Assuming You Know: Epistemic Semantics of Relational Annotations for Expressive Flow Policies. In *CSF*. IEEE Computer Society, 189–203.

[22] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *CSF*. IEEE Computer Society, 51–65.

[23] Jason A. Donenfeld. [n. d.]. Go Implementation of WireGuard. https://git.zx2c4.com/wireguard-go. [Online; accessed 11-March-2021].

[24] Marco Eilers, Severin Meier, and Peter Müller. 2021. Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security. In *CAV (1) (LNCS, Vol. 12759)*. Springer, 718–741.

[25] Marco Eilers, Peter Müller, and Samuel Hitz. 2018. Modular Product Programs. In *ESOP (LNCS, Vol. 10801)*. Springer, 502–529.

[26] Gidon Ernst and Toby Murray. 2019. SecCSL: Security Concurrent Separation Logic. In *CAV (2) (LNCS, Vol. 11562)*. Springer, 208–230.

[27] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The spirit of ghost code. *Formal Methods Syst. Des.* 48, 3 (2016), 152–174.

[28] Joseph A. Goguen and José Meseguer. 1984. Unwinding and Inference Control. In *S&P*. IEEE Computer Society, 75–87.

[29] Jean Goubault-Larrecq, Catuscia Palamidessi, and Angelo Troina. 2007. A Probabilistic Applied Pi-Calculus. In *APLAS (LNCS, Vol. 4807)*. Springer, 175–190.

[30] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650.

[31] Heiko Mantel. 2003. *A uniform framework for the formal specification and verification of information flow security.* Ph. D. Dissertation. Saarland University, Saarbrücken, Germany.

[32] Heiko Mantel, David Sands, and Henning Sudbrock. 2011. Assumptions and Guarantees for Compositional Noninterference. In *CSF*. IEEE Computer Society, 218–232.

[33] John McLean. 1992. Proving Noninterference and Functional Correctness Using Traces. *J. Comput. Secur.* 1, 1 (1992), 37–58.

[34] Jan Menz, Andrew K. Hirsch, Peixuan Li, and Deepak Garg. 2023. Compositional Security Definitions for Higher-Order Where Declassification. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 406–433.

[35] Toby Murray, Mukesh Tiwari, Gidon Ernst, and David A. Naumann. 2023. Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications. In *CCS*. ACM, 1746–1760.

[36] David A. Naumann. 2006. From Coupling Relations to Mated Invariants for Checking Information Flow. In *ESORICS (LNCS, Vol. 4189)*. Springer, 279–296.

[37] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. 2006. Information-Flow Security for Interactive Programs. In *CSFW*. IEEE Computer Society, 190–201.

[38] Wytse Oortwijn and Marieke Huisman. 2019. Practical Abstractions for Automated Verification of Message Passing Concurrency. In *IFM (LNCS, Vol. 11918)*. Springer, 399–417.

[39] Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *ESOP (LNCS, Vol. 9032)*. Springer, 158–182.

[40] Andrei Popescu, Thomas Bauereiss, and Peter Lammich. 2021. Bounded-Deducibility Security (Invited Paper). In *ITP (LIPIcs, Vol. 193)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:20.

[41] Andrei Popescu, Peter Lammich, and Thomas Bauereiss. 2021. CoCon: A Confidentiality-Verified Conference Management System. *Arch. Formal Proofs* 2021 (2021).

[42] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

[43] A. W. Roscoe. 1995. CSP and determinism in security modelling. In *S&P*. IEEE Computer Society, 114–127.

[44] Mark Dermot Ryan and Ben Smyth. 2011. Applied pi calculus. In *Formal Models and Techniques for Analyzing Security Protocols*. Cryptology and Information Security Series, Vol. 5. IOS Press, 112–142.

[45] Andrei Sabelfeld and Andrew C. Myers. 2003. A Model for Delimited Information Release. In *ISSS (LNCS, Vol. 3233)*. Springer, 174–191.

[46] Andrei Sabelfeld and David Sands. 2000. Probabilistic Noninterference for Multi-Threaded Programs. In *CSFW*. IEEE Computer Society, 200–214.

[47] Andrei Sabelfeld and David Sands. 2005. Dimensions and Principles of Declassification. In *CSFW*. IEEE Computer Society, 255–269.

[48] Daniel Schoepe, Toby Murray, and Andrei Sabelfeld. 2020. VERONICA: Expressive and Precise Concurrent Information Flow Security. In *CSF*. IEEE, 79–94.

[49] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 2:1–2:58.

[50] Graeme Smith. 2022. Declassification Predicates for Controlled Information Release. In *ICFEM (LNCS, Vol. 13478)*. Springer, 298–315.

[51] Geoffrey Smith and Dennis M. Volpano. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *POPL*. ACM, 355–364.

[52] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. 2020. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 152:1–152:31.

[53] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *CAV (1) (LNCS, Vol. 12759)*. Springer, 367–379.

[54] Felix A. Wolf and Peter Müller. 2024. *Verifiable Security Policies for Distributed Systems.* https://doi.org/10.5281/zenodo.13686927

[55] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32.

[56] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *CSFW*. IEEE Computer Society, 15–23.