A Framework for Composing Noninterferent Languages

Andreas Gampe and Jeffery von Ronne

The University of Texas at San Antonio
{agampe,vonronne}@cs.utsa.edu

Abstract. Software is frequently implemented using multiple specialized languages for different tasks. Security type systems, one technique to secure software, have previously only been studied for programs written entirely in a single language. We present a framework that facilitates reasoning over composite languages. In it, guarantees of sufficiently related component languages can be lifted to the composed language. This can significantly lower the burden necessary to certify that such composite programs are safe. Our simple but powerful approach relies, at its core, on computability and security-level separability. Besides noninterference, we also show how the framework can accomodate notions of declassification. To demonstrate the applicability of this framework, we show that a standard secure while language from the literature satisfies all necessary requirements of a composition host, and sketch how to securely embed an expressive fragment of SQL.

1 Introduction

To manage the rising complexity of creating software, systems are routinely created out of (mostly) independent components where each component can be specified, implemented and verified separately. In the language domain, one example for this is the composition of fragments from different languages into a complete program. A standard use case is the separation of storage and program logic concerns by embedding SQL queries into program code.

In parallel with the need to cope with this rising complexity, privacy is becoming an increasingly important property of software. Utilizing a security type system is one way to formally and soundly verify software against privacy policies and enforce properties like noninterference [6]. Noninterference ensures that any compliant program cannot leak private information to public channels. Many such type systems exist (e.g., [22, 7, 14, 24, 1, 8], for an overview see [19]) and the approach has been extended to cover entire distributed systems [11]. In all of this work, however, exactly one language is treated.

In contrast, how can we (statically) guarantee the safety of programs that are composed from elements in different languages? We propose to compose security-typed languages into *composed languages*, such that well-typed programs in a composed language can be guaranteed to comply with noninterference. This paper studies an approach that, under certain assumptions, makes it possible to leverage proofs of non-interference of well-typed host language and well-typed embedded language programs to prove noninterference of well-typed composed language programs. In order to generalize this composition over security-typed host and security-typed embedded languages that use different proofs that well-typed programs are noninterferent, our approach relies on host languages being complete with respect to being able to compute any noninterferent function over its data types. This allows us to establish that executing noninterferent code does not introduce any behaviors that could not be observed in the host language.

This paper validates the approach on a composition of a security-typed while language and a simple security-typed SQL fragment. We demonstrate a constructive technique to prove completeness with respect to noninterfering computations on the example of the while language, and formally prove all requirements to complete our framework approach.

The contributions of this paper include the description of a general framework for composition of security-typed languages, such that noninterference of the composed language can be established from the proofs of noninterference of the component languages; showing how two notions of declassification fit into and can be enforced by the framework; and demonstrating the framework on a composition of a security-typed while language.

This paper is structured as follows. Section 2 gives a short overview over background material. Next, in Section 3 we detail our goals of composition on the example of a student information system. Our framework approach is outlined in Section 4. Section 5 describes a case study, in which we compose a while language with a security-typed SQL fragment.

Section 7 compares against some related work, while Section 8 concludes.

2 Background

Most security type systems are based on the lattice model of security, [3] in which security information is taken from a lattice of security "levels". These levels facilitate both specifying how information is allowed to flow and also formalizing the assumed abilities of various attackers. Informally, (1) computations that involve private (=high) information need to be classified private, (2) private information cannot be assigned directly to public storage, and (3) private information is not allowed to be leaked indirectly to public storage through the program's control flow.

The most simple and non-trivial lattice used is $\{L, H\}$ with the obvious ordering ($L \sqsubseteq L$, $L \sqsubseteq H$, $H \sqsubseteq H$), but many more complicated lattices have been proposed (e.g., [15]). We only focus on lattice-based security in our framework. Noninterference can be generalized with the help of an indistinguishability relation that defines which parts of values may be distinguishable (and thus should not be influenced).

3 Motivation

Our ultimate goal is to *prove* safe the composition of practical languages. We use the example of a system composed of application logic and backend storage here. The application logic is written in an imperative language, while the storage is accessed with a SQL dialect. For example, imagine a university system that stores students' data. We can model this with a table that stores a record for each student, e.g., the student's name, room number, and several grades. Mandated by law, the grades are private information and must not be shared with unauthorized personnel, while room numbers can be used in a university directory. We can model this security by assigning low confidentiality to the name and room number, and high confidentiality to the grades. Now general staff can be classified as low, too, so to be able to access a student's room number. We can write a program that reads the database and writes this information to a low output, e.g., a generally accessible website. Note that `eval` will process the nested query and return the result. In a more practical language, the explicit use of this construct may be hidden by a layer of syntactic sugar.

```
Program list-students;
Schema:  students: name=L, room=L, grade1=H;
Code:
 length{L} = eval("SELECT count(*) FROM students");
 i{L} = 0;
 while i < L do
  name{L},room{L},grade1{L} = eval("SELECT name,room, grade1
    FROM students LIMIT $x,$x",i);
  print-public name, room, grade1
  i++;
```

This program should be rejected because of the leak of the grade, namely that the level in the host language(L) does not correspond with the level in the embedded language(H). However, if that part is removed, the program is valid and should pass the information flow checker. Similarly, updates in the database need to be protected, e.g., the following program needs to be rejected.

```
Program update-room;
Schema:  students: name=L, room=L, grade1=H;
Code:
 grade1{H} = eval("SELECT grade1 FROM students WHERE name='...'");
 if grade1>3 then
  eval("UPDATE room=3 WHERE name='...'");
```

Assume that our SQL dialect is proven secure (see Section 5.2), as well as the While language (e.g., [22]). We would now like to prove that the composed language is also secure.

4 Framework for Composition

To establish that well-typed composite program are secure, that is, noninterfering, we view composite programs as host-language contexts (a context is a program with holes), where all holes have been filled with some eval statements. Our goal is to show that we can fill the context with well-typed host program fragments such that each fragment matches the effect of the corresponding function implemented in the embedded language, and the new complete program remains well-typed. If this is possible, then the composite program has an equivalent well-typed host program, which is guaranteed to be noninterfering by the host-level type system guarantees. It follows that the composite program must be noninterfering, too. The following three subsections describe sufficient requirements guaranteeing such a transformation.

4.1 Simulation

Simulation is sufficient to find a host-language fragment that matches the effects of embedded language programs. Simulation requires that for each each function of interest computed in the embedded language, there exists a program in the host language that simulates its behavior. If $\mathcal{I} \subseteq \mathcal{L}_{\mathcal{E}}$ is the class of programs of interest, a subset of all programs of the embedded language, $\mathcal{L}_H$ is the class of all host programs, and $[\cdot]$ is the computed function of a program, we may formalize this as $\forall p \in \mathcal{I}. \exists p' \in \mathcal{L}_H.[p] \approx [p']$. In short, the host language is computationally at least as powerful as the embedded language.

In practice, we consider host languages that are Turing-complete, that is, every Turing-computable function can be computed using a single security level, and the embedded language might be Turing-computable, that is, every embedded-language program computes a Turing-computable function. Then the host language can obviously simulate any embedded-language program. Note that any other specific level of computability is acceptable, as long as it allows covering all embedded-language behaviors.

For security-typed composite languages we are only interested in the class of noninterfering computations of the embedded language. If a composed program is typed, then the definition of `eval` and its typing rules ensure that the embedded code fragment is typable in the embedded language. This means that the embedded program is noninterfering by virtue of the embedded language's noninterference statement. Now since the simulation is required to compute an equivalent function, and noninterference is a semantic property that only relies on inputs and outputs, the simulation is also noninterfering. Note that this of course only holds if the interface established by `eval` satisfies constraints about the values and types involved, for example, preserving indistinguishability of values.

Finally, note that only this step is specific to a certain composition. For each composition, it needs to be ensured that the embedded language is at most as powerful as the host language. All following steps are specific to the host language and can be reused for other compositions. However, we think that the simulation step is broadly applicable, because most host languages are expected to be Turing-complete general-purpose languages that are at the top of a realistic computability hierarchy.

4.2 Typability

While we now know that the simulation itself is noninterfering, it remains to see whether this holds for the whole program the simulation is a part of. We reduce this problem to two steps, the first of which is typability. If we can show that the simulation is typable, host-level noninterference will apply and secure the computation.

Note that the framework is a theoretical tool to prove composed languages secure. As such, the actual simulation is not important for running the composed program. This means that we can use *any* simulation that is equivalent to the embedded code fragment, no matter how complex, if it helps us find a typing.

We define the following property.

Definition 1 (Security Completeness). *A security-typed language $\mathcal{L}$ is security-complete if and only if for every program c that computes a function f, where f is noninterfering with respect to security signature $\mathcal{S}$, there exists a program c' that also computes f and is typable corresponding to $\mathcal{S}$.*

In the case of a security-complete host language, the typability step is immediately obvious, since the simulation is noninterferent. We will then use the typable simulation program guaranteed to exist. The While language we investigate in this paper is security-complete, as we will show in Theorem 2.

Note that all practical general-purpose security-typed languages seem to be security-complete for functions over integers. In other work [4] we study the limits of security-completeness and basic requirements on a security type system. We derive sufficient conditions for versions of security-completeness for functional languages with nonrecursive and recursive algebraic datatypes, as well as languages with heaps and side effects including a simple class-based object-oriented language.

4.3 Replacement

Last, if we can replace all instances of `eval` in the composite program with the corresponding simulations, and can show that typings and meaning are properly preserved, we have found a pure host-language program that is host-typable, which implies its noninterference. Again, noninterference can then be reflected onto the composite program, which is functionally equivalent. This is the third step, which is also only host-language specific.

Note that this is not a traditional substitution lemma. Standard substitution lemmas are concerned with replacing a variable with a term, and preserving a typing in the process. In our case, a whole construct, namely `eval`, must be replaced, and the typing of the replaced term may be under a different environment to account for the possible temporaries and side effects of the simulation.

Furthermore, one has to show that the programs before and after substitution are functionally equivalent. However, as mentioned, this has to be proven only once and can be reused for other compositions involving this host language.

5 Case Study

This section formalizes a case study for applying our framework. In Subsection 5.1 we introduce While. The following subsection sketches our fragment of SQL, gives its type system, and states noninterference. Subsection 5.3 formalizes the composed language. In the last subsection, we outline the proofs required for applying the framework to the composed language. A full development can be found in the accompanying technical report [5].

5.1 Host: WHILE

We base our exposition on a simplified version of [22]. This is a simple While language, with the following expressions and statements.

$$e ::= n \mid x \mid e \odot e \qquad c ::= x \ := \ e \mid c; c' \mid \mathsf{if}\, e \,\mathsf{then}\, c \,\mathsf{else}\, c' \mid \mathsf{while}\, e \,\mathsf{do}\, c$$

The language only supports integer values. A state μ binds variables to integers. We use a natural semantics. The semantics connects an input state and a statement with an output state and is fully standard and thus elided here.

Since the language only supports integers, it is not necessary to have a ground type system. A typing environment Γ binds variables to security levels. Judgments have the form $\Gamma \vdash e : \ell$ and $\Gamma \vdash c : \ell\, \mathsf{ok}$ and are also standard.

Two states are indistinguishable if they agree on all observable variables: $\mu_1 \sim_{\Gamma,\ell} \mu_2 \iff \forall x. \Gamma(x) \sqsubseteq \ell \implies \mu_1(x) = \mu_2(x)$. Noninterference states that a typed computation, when started with indistinguishable states, results in indistinguishable states. For a proof we refer to [22].

5.2 Embedded: SQL

We only sketch our security-typed fragment of SQL here. States are sets of named tables of integer data. The syntax of the fragment is given by

$$\begin{aligned} \hat{t} &::= \dot{t} \mid \dot{t}\,\mathsf{join}\,\dot{t}'\,\mathsf{on}\, i = i' \qquad\qquad \dot{e} ::= \dot{n} \mid i \mid \dot{x} \mid \dot{e} \oplus \dot{e}' \\ \dot{c} &::= \mathsf{select}\, \dot{e}_1, \ldots, \dot{e}_n \,\mathsf{from}\, \hat{t} \,\mathsf{where}\, \dot{e}' \mid \mathsf{insert}\, i = \dot{e} \,\mathsf{into}\, \dot{t} \mid \\ &\quad\ \mathsf{update}\, i = \dot{e} \,\mathsf{in}\, \dot{t} \,\mathsf{where}\, \dot{e}' \mid \mathsf{delete}\,\mathsf{from}\, \dot{t} \,\mathsf{where}\, \dot{e} \end{aligned}$$

The semantics is straightforward. We add a security-type system with judgments relative to environments Δ that map table columns to security labels. The type judgment has the form $\Delta, \dot{\Gamma} \vdash \dot{c} : (\ell_1, \ldots, \ell_n)^{\ell'} / \ell_s\, \mathsf{ok}$, where ℓ_i are the levels of the result columns, ℓ' is the level of the whole result and ℓ_s is a lower bound on the side effect of $\dot{c}$. States are indistinguishable with respect to Δ and ℓ if the erasure of columns not typed at or below ℓ is equivalent. Typable queries can be shown to be noninterfering.

Theorem 1 (SQL Noninterference).

$$\begin{aligned}
&\forall \mathit{\Delta}, \ell_\bullet, \ell', \ell'', \ell_x, c, \mu_1, \mu_2, \mu_1', \mu_2', \dot{n}_1, \dot{n}_2, s_1, s_2.\\
&\quad \mathit{\Delta}, x : \ell_x \vdash c : \ell_\bullet^{\ell'} / \ell'' \,\mathrm{ok} \wedge \mu_1 \dot{\sim}_{\mathit{\Delta},\ell_o} \mu_2 \wedge \dot{n}_1 \dot{\sim}_{\ell_x,\ell_o} \dot{n}_2 \wedge \mu_1, c[x := \dot{n}_1] \Rightarrow \mu_1', s_1 \wedge\\
&\quad \mu_2, c[x := \dot{n}_2] \Rightarrow \mu_2', s_2\\
&\qquad\qquad\qquad\qquad \Longrightarrow \mu_1' \dot{\sim}_{\mathit{\Delta},\ell_o} \mu_2' \wedge s_1 \dot{\sim}_{\ell_\bullet^{\ell'},\ell_o} s_2
\end{aligned}$$

5.3 Composed Language

This section formalizes the extension of the host language, to create a composed language of WHILE and SQL. We restrict ourselves to transfer simple integers. We add the statement $x_1, \ldots, x_n := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$ for evaluation, where x_i designate the variables that will hold the result, and x' is a host-level variable that is a parameter for the embedded computation $\dot{c}$. We will use $x_\bullet$ to denote a list of variables.

We extend the host semantics in the following two ways. First, the embedded state ν is threaded through the original reduction rules, which do not update the embedded states themselves. In the case of WHILE, this is unambiguous. In languages with multiple nested reductions this threading will induce a certain evaluation order. Second, reduction of eval is given by the following new rules.

$$\frac{\nu, \dot{c}[\dot{x} := \mu(x')] \Rightarrow \nu', \emptyset}{\mu, \nu, x_\bullet := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c} \Rightarrow \mu[x_1 := 0][\ldots][x_n := 0], \nu'}$$

$$\frac{\nu, \dot{c}[\dot{x} := \mu(x')] \Rightarrow \nu', s \quad s \neq \emptyset \quad s(0) = (n_1, \ldots, n_n)}{\mu, \nu, x_\bullet := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c} \Rightarrow \mu[x_1 := n_1][\ldots][x_n := n_n], \nu'}$$

Typing rules are changed accordingly. That is, $\mathit{\Delta}$ is threaded through the original typing rules, and statement types are extended by the lower bound of changes in the embedded state. Furthermore, we add the following typing rule for eval.

$$\frac{\mathit{\Gamma} \vdash x' : \ell \quad \mathit{\Delta}, [\dot{x} : \ell] \vdash \dot{c} : \ell_\bullet^{\ell_2} / \ell_s \,\mathrm{ok} \quad \forall i. \ell_i \sqcup \ell_2 \sqsubseteq \mathit{\Gamma}(x_i)}{\mathit{\Gamma}, \mathit{\Delta} \vdash x_\bullet := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c} : (\prod \mathit{\Gamma}(x_i)) \sqcap \ell_S \,\mathrm{ok}}$$

Indistinguishability is lifted component-wise, that is, $\mu_1, \nu_1 \ddot{\sim}_{\mathit{\Gamma},\mathit{\Delta},\ell} \mu_2, \nu_2 \iff \mu_1 \sim_{\mathit{\Gamma},\ell} \mu_2 \wedge \nu_1 \dot{\sim}_{\mathit{\Delta},\ell} \nu_2$. This suffices because SQL values, that is, result sets, do not occur as values in the composed language. Then noninterference is stated analogously to the WHILE case.

$$\begin{aligned}
&\forall \ell, \mathit{\Gamma}, c, \mu_1, \nu_1, \mu_2, \nu_2, \mu_1', \nu_1', \mu_2', \nu_2'.\\
&\quad \mathit{\Gamma} \vdash c : \ell \,\mathrm{ok} \wedge \mu_1, \nu_1 \ddot{\sim}_{\mathit{\Gamma},\mathit{\Delta},\ell} \mu_2, \nu_2 \wedge \mu_1, \nu_1, c \Rightarrow \mu_1', \nu_1' \wedge \mu_2, \nu_2, c \Rightarrow \mu_2', \nu_2'\\
&\qquad\qquad\qquad\qquad \Longrightarrow \mu_1', \nu_1' \ddot{\sim}_{\mathit{\Gamma},\mathit{\Delta},\ell} \mu_2', \nu_2'
\end{aligned}$$

Remarks In general, our framework can support more complicated compositions. We allow auxiliary functions α, α^τ, γ and γ^τ, establishing how certain types and their values can be translated. We require monotonicity with respect to security levels and non-decreasing when composed for the type translations α^τ and γ^τ, and preservation of indistinguishability for value translations α and γ. For our example, both α and α^τ are simply identity functions. γ takes a list of tuples and projects it such that the result is the first tuple, if such exists, or zeros otherwise. γ^τ takes the element and length labels of SQL and combines them into their upper bound to account for the type of the data values, as well as for the result being empty or not. $\sqcup$ is monotonous, so γ^τ is. Indistinguishability of two lists of tuples implies that they are either both empty or agree on their first element (or observable components thereof). Thus, indistinguishable results sets will be projected to indistinguishable tuples at the host level.

5.4 Proof Sketch

This section applies our framework approach outlined in section 4 and formally verifies that the composition of While and SQL from the previous section is safe, that is, typable composed programs are noninterfering. For this, recall the main steps:

1. Embedded-language programs can be simulated in the host
2. The simulation is noninterferent and can be typed
3. Replace embeddings with the simulation; the now pure-host program is typable, implying noninterference of the composed program

Simulation For the first step, we need to establish that the host language is computationally at least as powerful as the embedded language. In our case, we note that WHILE is Turing-complete, and SQL is Turing-computable. Thus, any program $\dot{c}$ can be simulated by a program c.

Noninterference The second step of our approach is split into two parts: noninterference and typability. Noninterference is derived from the conditions of the eval. Namely, given indistinguishable inputs, the respective values of the parameter will be indistinguishable. This is preserved when switching to the embedded language. By noninterference theorem of the embedded language and the embedded fragment being well-typed, the outputs must be indistinguishable. The translation to the host level preserves this property. The final update of the host state will thus maintain state indistinguishability. Overall, we have that indistinguishable inputs to an eval lead to indistinguishable output.

Noninterference is a semantical property defined over inputs and outputs. Since the simulation is functionally equivalent, that is, produces the same outputs for the same inputs, the simulation is also noninterferent.

Typing We now need to show that the simulation is typable with respect to the types of the eval. That is, we need to type the simulation such that the input corresponding to the eval's parameter is typed the same, i.e., as $\Gamma(x)$, as well as all the column encodings according to the typing given by Δ. Note that, in fact, it is not even necessary to show typability of the simulation found in the first step. In many languages, many programs compute the same function. Only one program out of the class of equivalent programs needs to be typable. We use the following theorem to posit the existence of such a program.

Theorem 2 (Security Completeness). *If function f is computable in WHILE, and noninterferent under a signature given by Γ, then there exists a WHILE program c that is typable under a typing environment Γ' that is an extension of Γ.*

We only sketch the proof here, which relies on two auxiliary lemmas. First, any WHILE program is typable at a single level ℓ. Second, two typed WHILE programs can be composed such that the composition is typable. Equipped with these lemmas, we construct a typable program for every output of the original program at that output's level, which is possible by the first lemma. We can substitute the inputs at a higher security level with arbitrarily chosen constants, e.g., zero. Noninterference guarantees that the output remains correct. Finally, we can compose all separate programs to compose the whole output, as validated by the second lemma.

Replacing eval The last step of our approach ties the previous subsections together and shows how to replace the eval for the simulation. Given an eval statement, let c_{eval} be its typable host simulation.

Lemma 1 (Substitution Typable). *Assume a context $E[\bullet]$, that is, a composed program with a statement hole. Furthermore assume an $x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$, Γ, Δ and ℓ such that $\Gamma, \Delta \vdash E[x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}] : \ell\, \mathrm{ok}$. Then there exists an extension Γ' of Γ such that $\Gamma', \Delta \vdash E[c_{eval}] : \ell\, \mathrm{ok}$.*

We can use this lemma to iteratively replace all eval statements in the original composed program with their respective simulations. The result is a typing of a pure-host statement under the composed-language rules. The next lemma states that such a typing induces the corresponding host-level typing of the statement.

Lemma 2 (Eval-free Composed To Host). *If a statement c that does not contain* eval *is typed under Γ and Δ as $\Gamma, \Delta \vdash c : \ell\, \mathrm{ok}$, then c can be typed as $\Gamma \vdash c : \ell\, \mathrm{ok}$.*

Corollary 1 (Simulation Pure-Host Typable). *If c is a composite program that is typable as $\Gamma, \Delta \vdash c : \ell\, \mathrm{ok}$, then there exists an extension Γ' of Γ that encodes Δ such that the simulation program c_{eval}, where all* eval *statements have been substituted for their simulation, is typable as $\Gamma' \vdash c_{eval} : \ell\, \mathrm{ok}$.*

This formally proves that the simulation program is noninterferent by noninterference of typed host-language programs. Now, the final step needs to formally show that the simulation program is equivalent to the original program. This is obviously modulo the behavior of temporaries of the simulation, which are exposed to the host.

Theorem 3 (Simulation Equivalence up to Γ). *If $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$, then there exists an extension Γ' such that $\Gamma' \vdash c_{eval} : \ell\,\mathrm{ok}$, and for all $\mu_1, \mu_1', \nu, \nu', \mu_2$ where $\Gamma \vdash \mu_1\,\mathrm{ok}$, $\Gamma' \vdash \mu_2\,\mathrm{ok}$ and μ_2 is an extension of μ_1 such that the extension encodes ν, and $\mu_1, \nu, c \Rightarrow \mu_1', \nu'$, then there exists μ_2' an extension of μ_1' such that $\mu_2, c_{eval} \Rightarrow \mu_2'$ and the extension encodes ν'.*

Corollary 2 (Replacement). *For a typable composed program c, the program c_{eval} created by replacing all* `eval` *statements with a corresponding simulation, is typable and functionally equivalent to c.*

This concludes all three steps of our framework. Together, this formally shows that all composed-language programs can be translated into pure host-language programs that remain typed with a corresponding typing environment and are functionally equivalent to the composed program. Noninterference of the host language applies to typed host programs, so the translation is noninterfering. Since the composed program computes the same function, it is also guaranteed to be noninterfering.

6 Declassification

In practice noninterference is too strong a property to enforce. A canonical example is a login process, which compares a given string to a stored password and allows access if they are identical. However, this constitutes a leak from the viewpoint of noninterference.

Declassification is necessary for intensional information release, relabeling data so that it becomes accessible. This is required when a system needs to leak information to function, the canonical example being a login process. The main questions are under what circumstances a declassification should be allowed and what security guarantee this entails. For a general overview and classification we refer to [21]. There is no general agreed-upon best definition. We will demonstrate how two examples can be integrated into our framework.

6.1 Delimited Release

In Delimited Release [20], declassification expressions define escape hatches. A program is secure iff under indistinguishable inputs, where all hatch expressions declassified to an observable level have equivalent outputs for those inputs, the outputs of the program are equivalent. [20] shows how a type-and-effect system can be used to enforce delimited release. The effects here are variables used in declassifaction (D), and variables modified (U). A program is guaranteed to be secure, if $D \cap U = \emptyset$.

Given two languages with such type-and-effect systems, we can extend the typing of an embedding to also encompass effects. An example is

$$\frac{\Gamma \vdash e : \ell, D_e \quad \Delta, [\dot{x} : \ell] \vdash \dot{c} : \ell_v/\ell_s, D_{\dot{c}}, U_{\dot{c}} \quad \ell_v \sqsubseteq \Gamma(x) \quad D_{\dot{x}} = \begin{cases} Vars(e) & \dot{x} \in D_{\dot{c}} \\ \emptyset & else \end{cases}}{\Gamma, \Delta \vdash x := \mathrm{eval}\, e \,\mathrm{in}\, \dot{c} : \Gamma(x_i) \sqcap \ell_S, D_e \cup D_{\dot{c}} \cup D_{\dot{x}}, U_{\dot{c}} \cup \{x\}}$$

In words, the declassification effect of an eval encompasses the declassification in the parameter expression and the embedded declassification effect. Also, if the embedded effect contains the embedded parameter, then all variables in the parameter expression are involved in declassification. Similarly, the modified variables are made up of the variables modified in the embedded fragment, and the result variable at the host level.

6.2 Robust Declassification

Robust declassification [23] informally defines a system potentially including declassification to be *robust* if an attacker cannot deduce additional information when attacks are applied. This is an example of the *who* dimension of declassification, as the attacker does not have influence over what gets declassified.

In [16], a type system is proposed that enforces a variant robust declassification. A secure While language with both confidentiality and integrity is extended with a declassification expression and holes. Attacks are defined to be noninterfering programs that cannot influence high-integrity variables. A program is complete when holes are filled with attacks. The type system then enforces that for a typable program with holes, for any two derived complete programs and all inputs, if an attacker cannot distinguish runs of the first program, then she cannot distinguish runs of the second.

To show how two languages enforcing robust declassification can be composed, we first abstract the guarantees of the language of [16], transposing them to requirements on traces in a state transition system. We call this property step-wise robustness. We then show that a system satisfying the requirements is robust with respect to the definition in [23]. Finally, given two systems guaranteeing step-wise robustness, we show that the disciplined composition given by typed embedding continues to be step-wise robust.

Let $\mathcal{S} = (\varSigma, \mapsto, L, D, A, E, \varGamma)$ be an (annotated) state transition system, where states are tuples of the form $\sigma = (\sigma_{HH}, \sigma_{HL}, \sigma_{LH}, \sigma_{LL}, l)$ with l denoting an abstract location from L. $\mapsto$ is a binary relation over states, D is a predicate describing which locations may declassify data, that is, over which $\mapsto$ is not required to be noninterference-preserving. A is a predicate describing which locations denote attacks, E maps locations to exit points of single-exit regions of which the location is a part, and $\varGamma$ is a mapping of locations to security levels. Traces τ are (possibly empty) sequences of states such that $\tau(0) \mapsto \tau(1) \mapsto \dots$. $\tau.e$ is the last state of τ if τ is finite and $\perp$ otherwise. Traces (and states) are concatenated with $\oplus$.

A system $\mathcal{S}$ is valid if certain properties hold for $\varGamma$ and E, namely

- $\forall \sigma \mapsto \sigma'.\ \exists k \geq 0.\ E(\sigma.l) = E^k(\sigma'.l) \land \forall 0 \leq i < k.\ \varGamma(\sigma.l) \sqsubseteq \varGamma(E^i(\sigma'.l))$
- $\forall l.\varGamma(E(l)) \sqsubseteq \varGamma(l)$
- $\forall \sigma_1 \mapsto \sigma_1', \sigma_2 \mapsto \sigma_2'.\sigma_1 \approx_{C/I} \sigma_2 \land \sigma_1.l = \sigma_2.l \land \sigma_1'.l \neq \sigma_2'.l \implies \varGamma(\sigma_1'.l) \in H_{C/I} \land \varGamma(\sigma_2'.l) \in H_{C/I}$.
- $\forall l.D(l) \implies \varGamma(l) \in H_I \land \varGamma(l) \in L_C$
- $\forall l.A(l) \implies \varGamma(l) \in L_C$

where $E^0(l) = l$ and $E^{k+1}(l) = E^k(E(l))$. Here $L_{I/C}$ are the low-integrity/low-confidentiality levels, and $H_{I/C}$ correspondingly. We define confidentiality-indistinguishability satisfying $\sigma \approx_C \sigma' \implies \sigma_{LH} = \sigma'_{LH} \wedge \sigma_{LL} = \sigma'_{LL}$, and analogously integrity-indistinguishability satsifying $\sigma \approx_I \sigma' \implies \sigma_{LH} = \sigma'_{LH} \wedge \sigma_{HH} = \sigma'_{HH}$. The reverse direction is required to hold whenever also $\sigma.l = \sigma'.l$; or $\Gamma(\sigma.l) \in H$, $\Gamma(\sigma'.l) \in H$ and for the smallest k and k' such that $E^k(\sigma.l) \in L$ and $E^{k'}(\sigma'.l) \in L$ the elements $E^k(\sigma.l)$ and $E^{k'}(\sigma'.l)$ are the same; or $\Gamma(\sigma.l) \in H$, $\Gamma(\sigma'.l) \in L$ and $\sigma'.l = E^k(\sigma.l)$ is minimal in k such that $E^k(\sigma.l) \in L$; or the symmetric case of the last one. For termination-sensitive noninterference, $\perp$ is only equivalent to itself, while for termination-insensitive noninterference $\perp$ is low-equivalent with every state.

We model attacks as part of $\mathcal{S}$, that is, $\mapsto_A \subseteq \mapsto$, where a single transition captures the whole attack. The requirements on $\mapsto_A$ then are that attacks are noninterferent computations, thus indistinguishable inputs to an attack lead to indistinguishable outputs; attacks do not change the high-integrity parts of a state; and start in a low-confidentiality locations, that is $\forall l. A(l) \implies \Gamma(l) \in L_C$. These are the standard definitions from [16]. Further, attacks are only allowed at specific locations denoted by A, and for all attacks $\sigma_1 \mapsto_A \sigma_2$ and $\sigma'_1 \mapsto \sigma'_2$ where $\sigma_1.l = \sigma'_1.l$ we have $\sigma_2.l = \sigma'_2.l$. Last, to capture that attacks stand for (terminating) computations, we require they satisfy a closure property: all states with locations $A(l)$ have an attack transition, and only such relations.

A system $\mathcal{S}$ is step-wise robust with respect to attacks $\mapsto_A$ iff

1. $\mathcal{S}$ is valid and deterministic in non-attack transitions/locations
2. $\forall \sigma_1, \sigma'_1, \sigma_2.\ \sigma_1 \approx_I \sigma_2 \wedge \sigma_1 \mapsto \sigma'_1 \implies \exists \tau'_2. \forall i < len(\tau'_2). \tau'_2(i) \approx_I \sigma_2 \wedge \tau'_2.e \approx_I \sigma'_1$
   Given two integrity-indistinguishable states, a step of one can be matched by zero or more steps of the other so that the result remains integrity-indistinguishable.
3. $\forall \sigma_1, \sigma'_1, \sigma_2.\ \sigma_1 \approx_C \sigma_2 \wedge \sigma_1 \mapsto \sigma'_1 \wedge \neg D(\sigma_1) \implies \exists \tau'_2. \forall i < len(\tau'_2). \tau'_2(i) \approx_C \sigma_2 \wedge \tau'_2.e \approx_C \sigma'_1$
   Given two confidentiality-indistinguishable states and the first is not declassifying, a step of the first can be matched by zero or more steps of the second so that the result remains confidentiality-indistinguishable.
4. $\forall \sigma_1, \sigma_2.\ \sigma_1 \approx_{I/C} \sigma_2 \wedge D(\sigma_1) \implies \exists \tau'_2. \forall i < len(\tau'_2). \tau'_2(i) \approx_{I/C} \sigma_2 \wedge D(\tau'_2.e)$
   If a state is marked as declassifying , then a declassification must be reachable from all integrity-equivalent/confidentiality-equivalent states.
5. $\forall \sigma, \sigma'.\ \sigma \mapsto \sigma' \wedge D(\sigma) \implies \sigma_{LL} = \sigma'_{LL}$
   Declassification does not influence the LL part of a state.

The language in [16] is a structured and well-behaved While language with small-step semantics defined over heaps M and statements c. Given a typed program $\Gamma, pc \vdash c[\bullet]$, we can translate it to a corresponding transition system $\mathcal{S}$. This system satisfies that for any attack a and intermediate state $\langle M, c' \rangle$ such that $\langle M_0, c[a] \rangle \to^* \langle M, c' \rangle$, $\langle M, c' \rangle \equiv \sigma \mapsto \sigma' \equiv \langle M', c'' \rangle$ can be matched to a (multi-)step $\langle M, c' \rangle \to^* \langle M', c'' \rangle$. With this we can derive the following theorem.

Theorem 4 (Language-robust implies step-wise robust). *If $\Gamma, pc \vdash c[\bullet]$, then there exists $\mathcal{S}$ that is (a) able to simulate all runs of c under any attack a and (b) $\mathcal{S}$ is step-wise robust.*

Next we show that step-wise robustness is a meaningful declassification guarantee by showing that it implies robustness as adapted from [23]. Let $\mapsto_{skip} \subseteq \mapsto_A$ such that for all $\sigma \mapsto_{skip} \sigma'$ we have the four non-location components equivalent. A system restricted to those attack transitions can be considered not under attack. Informally, trace-based robustness states that if two starting states are observationally equivalent in the base system, then they are observationally equivalent when under attack, where observational equivalence is the equivalence of the sets of traces generated starting at the start states modulo indistinguishability. Formally, $\mathcal{O}_\sigma(\mathcal{S}, \approx) = \{\tau / \approx | \tau \in \mathcal{T}(\mathcal{S}, \sigma)\}$ and $\mathcal{S}$ is robust with respect to attack A iff for all σ and σ', $\mathcal{O}_\sigma(\mathcal{S}, \approx) = \mathcal{O}_{\sigma'}(\mathcal{S}, \approx) \implies \mathcal{O}_\sigma(\mathcal{S} \cup A, \approx) = \mathcal{O}_{\sigma'}(\mathcal{S} \cup A, \approx)$. The requirements for step-wise robustness allow to derive the following theorem.

Theorem 5 (Step-wise robust implies robust). *If $\mathcal{S} = (\Sigma, \mapsto, D)$ is step-wise robust with respect to $\mapsto_A$, then $(\Sigma, \mapsto \setminus \mapsto_A \cup \mapsto_{skip})$ is robust with respect to A and $\approx_C$.*

Finally, we apply step-wise robustness to composition. Given systems $\mathcal{S}_1$ and $\mathcal{S}_2$, we create a composed system by defining $\mathcal{S} = (\Sigma = \Sigma_1 \times \Sigma_2 \times L, \mapsto, L = L_1 \cup L_2, D = D_1 \cup D_2, A = A_1 \cup A_2, E = E_1 \cup E_2, \Gamma = \Gamma_1 \cup \Gamma_2))$ and $(\sigma_1, \sigma_2, s) \mapsto (\sigma_1', \sigma_2', s')$ only if (1) $s = l_1$, $s' = l_1'$ and $\sigma_1 \mapsto_1 \sigma_1'$ and $\sigma_2 = \sigma_2'$ where $\sigma_1.l = l_1$ and $\sigma_1'.l = l_1'$, (2) $s = l_2$, $s' = l_2'$ and $\sigma_1 = \sigma_1'$ and $\sigma_2 \mapsto_2 \sigma_2'$ where $\sigma_2.l = l_2$ and $\sigma_2'.l = l_2'$, (3) $s = l_1$, $s' = l_2$, $\sigma_1 = \sigma_1'$, $\sigma_1.l = l_1$ corresponds to $x_1, \ldots, x_n :=$ $\mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$ and σ_2' is an update of σ_2 corresponding to $\alpha(\sigma_1(x'))$ and $\sigma_2'.l = l_2$ corresponds to $\dot{c}$, or (4) $s = l_2$, $s' = l_1$, $\sigma_2 = \sigma_2'$, there is no successor to σ_2 in $\mathcal{S}_2$, $\sigma_2.l = l_2$, σ_1 corresponds to $x_1, \ldots, x_n := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$, σ_1' corresponds to the successor of σ_1 and is an update of σ_1 corresponding to $\gamma(\sigma_2(\dot{c}))$ with $\sigma_1'.l = l_1$. These four options correspond to either pure-host or pure-embedded computation ((1) and (2)), or invoking and returning from an eval.

For security, we straightforwardly lift $\approx_C$ and $\approx_I$ to the composition. If $\mathcal{S}_1$ and $\mathcal{S}_2$, the composite program is typed, and the embedded program is terminating if the eval is typed under high *pc*, then we can show the following theorem.

Theorem 6 (Composition is step-wise robust). *Given the above assumptions, the composition $\mathcal{S}$ is step-wise robust.*

7 Related Work

To the best of our knowledge, this is the first work to make use of completeness in the context of security-typed languages, which we studied in [4]. Kahrs [9] studied completeness for basic type systems, where the question is if all computable functions that are "well-going" can be typed. Kahrs uses transitions systems, whereas our goal is to permit easy adoption of existing languages.

Traditional work in security-typed languages attempts to broaden the permissibility of the type system, that is, accept more programs as typed and thus secure. For an overview we refer to [19]. Our work is orthogonal to such efforts. We show that, under certain constraints, there are always programs that compute a given noninterferent function.

Basic type-safety of composition has been investigated, e.g., [2] studied an SQL-like extension to $C^{\#}$. However, the query language was fully incorporated into the host, which seems infeasible for the sheer amount and expressivity of languages considered for embedding in practice. Compositionality of noninterference has been studied in different contexts (e.g., [13, 18, 12] for overviews). The goal here is to derive constraints on when composing secure program fragments of *one* formal system yields a secure result. In contrast, our fragments are derived from *different* security-typed languages (here While and SQL), and we base our compositionality result around the notion of *security completeness*.

There are several related approaches to secure a complex system. Li and Zdancewic [10] describe a web programming language extended with an interface to a relational database (note only simple select statements), where the language and interface are security typed. But this means that the storage side needs to be fully trusted. Arrows and monads (e.g.,[17] and references therein) can be used to isolate and control information flows in a library fashion. We note that a library approach to embedded languages restricts expressivity and conciseness to that of the host language.

Fabric [11] extends JIF to a secure distributed system, with storage an integral part. We believe that separation of concerns is important and our approach allows a modular proof of the safety of the whole system, composed from smaller fragments. Also note that, persistent storage is but one use of an embedded language, and it seems unrealistic to expect one language to excel in all domains.

8 Conclusion

In this paper, we have presented a framework that can be used to show the security of a language that is composed from host and embedded languages that are themselves secure. The approach is based on a simulation of the embedded component's behavior and the accompanying proof that the noninterference of the component's computation guarantees that there exists a typable program for the simulation. We can then refer to the strong guarantees of the host language to prove a composite program noninterferent.

This result allows us to develop separate type systems for individual languages, and lift the results to compositions of languages. It thus significantly reduces the burden of showing security in modern programs that employ many programming languages for different tasks like data retrieval and modification. In this paper we have shown safe the composition of a While language and the data manipulation fragment of SQL. Our effort for the While language can be reused, because our framework can be instantiated for other embedded languages with minimal effort.

References

1. Banerjee, A., Naumann, D.A.: Stack-based access control and secure information flow. J. Funct. Program. 15(2), 131–177 (Mar 2005)
2. Bierman, G., Meijer, E., Schulte, W.: The essence of data access in cω: The power is in the dot. In: In ECOOP'02 (2002)
3. Denning, D.E.: A lattice model of secure information flow. Commun. ACM 19, 236–243 (May 1976)
4. Gampe, A., von Ronne, J.: Security completeness: Towards noninterference in composed languages. In: Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security. pp. 27–38. PLAS '13, ACM, New York, NY, USA (2013)
5. Gampe, A., Von Ronne, J.: A framework for composing noninterferent languages. Tech. Rep. CS-TR-2013-014, Department of Computer Science, The University of Texas at San Antonio (2013), http://galadriel.cs.utsa.edu/tr/2013-14.pdf
6. Goguen, J.A., Meseguer, J.: Security Policies and Security Models, vol. pages, pp. 11–20. IEEE (1982)
7. Heintze, N., Riecke, J.G.: The slam calculus: programming with secrecy and integrity. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '98 (1998)
8. Hunt, S., Sands, D.: From exponential to polynomial-time security typing via principal types. In: Proceedings of the 20th European conference on Programming languages and systems. ESOP'11/ETAPS'11 (2011)
9. Kahrs, S.: Well-going programs can be typed. In: Proceedings of the 6th international conference on Typed lambda calculi and applications. TLCA'03 (2003)
10. Li, P., Zdancewic, S.: Practical information-flow control in web-based information systems. In: Proceedings of the 18th IEEE workshop on Computer Security Foundations. CSFW '05 (2005)
11. Liu, J., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: a platform for secure distributed computation and storage. In: SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (2009)
12. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium. CSF '11 (2011)
13. McCullough, D.: Specifications for multi-level security and a hook-up. Security and Privacy, IEEE Symposium on 0, 161 (1987)
14. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL) (1999)
15. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proc. 17th ACM Symp. on Operating System Principles (1997)
16. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification and qualified robustness. J. Comput. Secur. 14(2), 157–196 (Apr 2006)
17. Russo, A., Claessen, K., Hughes, J.: A library for light-weight information-flow security in haskell. In: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 13–24. Haskell '08, ACM, New York, NY, USA (2008)
18. Ryan, P.Y.A., Schneider, S.A.: Process algebra and non-interference. J. Comput. Secur. 9(1-2), 75–103 (Jan 2001)
19. Sabelfeld, A., Myers, A.: Language-based information-flow security. Selected Areas in Communications, IEEE Journal on 21(1), 5 – 19 (Jan 2003)

20. Sabelfeld, A., Myers, A.: A model for delimited information release. In: Software Security - Theories and Systems, Lecture Notes in Computer Science, vol. 3233, pp. 174–191. Springer Berlin Heidelberg (2004)
21. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: Proceedings of the 18th IEEE workshop on Computer Security Foundations. pp. 255–269. CSFW '05 (2005)
22. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. 4, 167–187 (January 1996)
23. Zdancewic, S., Myers, A.C.: Robust declassification. In: Proceedings of the 14th IEEE workshop on Computer Security Foundations. pp. 5–. CSFW '01 (2001)
24. Zdancewic, S., Myers, A.C.: Secure information flow and cps. In: ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems. pp. 46–61. Springer-Verlag, London, UK (2001)

A Languages

A.1 WHILE

The semantics is as follows.

$$\mu, x := e \Rightarrow \mu[x := \mu(e)] \qquad \frac{\mu, c \Rightarrow \mu' \quad \mu', c' \Rightarrow \mu''}{\mu, c; c' \Rightarrow \mu''}$$

$$\frac{\mu(e) \neq 0 \quad \mu, c \Rightarrow \mu'}{\mu, \text{if}\, e\, \text{then}\, c\, \text{else}\, c' \Rightarrow \mu'} \quad \frac{\mu(e) = 0 \quad \mu, c' \Rightarrow \mu'}{\mu, \text{if}\, e\, \text{then}\, c\, \text{else}\, c' \Rightarrow \mu'}$$

$$\frac{\mu(e) \neq 0\ \mu, c \Rightarrow \mu'\ \mu', \text{while}\, e\, \text{do}\, c \Rightarrow \mu''}{\mu, \text{while}\, e\, \text{do}\, c \Rightarrow \mu''} \qquad \frac{\mu(e) = 0}{\mu, \text{while}\, e\, \text{do}\, c \Rightarrow \mu}$$

Typing is defined by the following rules.

$$\Gamma \vdash n : \ell \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma \vdash e : \ell \quad \Gamma \vdash e' : \ell}{\Gamma \vdash e \odot e' : \ell} \qquad \frac{\Gamma \vdash e : \ell \quad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell'}$$

$$\frac{\Gamma \vdash x : \ell \quad \Gamma \vdash e : \ell' \quad \ell' \sqsubseteq \ell}{\Gamma \vdash x := e : \ell} \qquad \frac{\Gamma \vdash x : \ell \quad \Gamma \vdash e : \ell' \quad \ell' \sqsubseteq \ell}{\Gamma \vdash c; c' : \ell\, \text{ok}}$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma \vdash c : \ell\, \text{ok} \quad \Gamma \vdash c' : \ell\, \text{ok}}{\Gamma \vdash \text{if}\, e\, \text{then}\, c\, \text{else}\, c' : \ell\, \text{ok}} \qquad \frac{\Gamma \vdash e : \bot \quad \Gamma \vdash c : \ell\, \text{ok}}{\Gamma \vdash \text{while}\, e\, \text{do}\, c : \ell\, \text{ok}}$$

A.2 SQL

We formalize a simplified version of the data retrieval and manipulation fragment of SQL. We allow (finitely many) named tables of integer data. Names $\dot{t}$ are drawn from a set $\mathcal{T}$. The set of tables is static and finite for a particular program: we do not allow table creation or deletion. Tables have a finite number of columns $\mathcal{I}_{\dot{t}}$, where we assume for simplicity that column names are distinct between tables. We use i to range over *all* column names. We have the following syntax, where we use a dot to distinguish from the WHILE syntax.

$$\begin{aligned}\hat{t} &::= \dot{t} \mid \dot{t}\,\mathrm{join}\,\dot{t}'\,\mathrm{on}\,i = i' \qquad\qquad \dot{e} ::= \dot{n} \mid i \mid \dot{x} \mid \dot{e} \oplus \dot{e}' \\ \dot{c} &::= \mathrm{select}\,\dot{e}_1, \dots, \dot{e}_n\,\mathrm{from}\,\hat{t}\,\mathrm{where}\,\dot{e}' \mid \mathrm{insert}\,i = \dot{e}\,\mathrm{into}\,\dot{t} \mid \\ &\qquad \mathrm{update}\,i = \dot{e}\,\mathrm{in}\,\dot{t}\,\mathrm{where}\,\dot{e}' \mid \mathrm{delete}\,\mathrm{from}\,\dot{t}\,\mathrm{where}\,\dot{e}\end{aligned}$$

A table state T is represented as a finite map of naturals to records r, which are a total map of column names to integers. Note that we identify states up to an order-preserving remapping. A database state ν is a finite map of table names to table states. For simplicity of handling table joins we extend the set of table names by table expressions, e.g., if A and B are table names and i and i' column names, then $A\,\mathrm{join}\,B\,\mathrm{on}\,i = i'$ is a table name.

Expressions can be evaluated for a record, denoted by $r(\dot{e})$, in the obvious way. Expressions do not change the database. Table expressions project and manipulate table states from a database state in the obvious way. Table projection and manipulation is formally defined in the following way:

$$\begin{aligned}&r \oplus r' = \lambda i. \begin{cases} r.i & i \in dom(r) \\ r.i' & i \in dom(r') \wedge i \notin dom(r) \\ \bot & \text{else} \end{cases} \\ &\nu(\hat{t}) = \nu(\dot{t}) \quad \text{if } \hat{t} = \dot{t} \\ &\nu(\dot{t}\,\mathrm{join}\,\dot{t}'\,\mathrm{on}\,i = i') = \lambda n. \begin{cases} r \oplus r' & \begin{aligned} &r = \nu(\dot{t})(n/|\nu(\dot{t})|).i \wedge \\ &r' = \nu(\dot{t}')(n\%|\nu(\dot{t})|) \wedge \\ &r.i = r'.i' \end{aligned} \\ \bot & \quad else \end{cases}\end{aligned}$$

A natural semantics reduces a statement and database state to a result and a state. We use the following auxiliary definitions.

$$T^{\neq}_{\dot{e}} = \lambda n. \begin{cases} r & T(n) = r \wedge r(\dot{e}) \neq \dot{0} \\ \bot & otherwise \end{cases} \qquad T^{=}_{\dot{e}} = \lambda n. \begin{cases} r & T(n) = r \wedge r(\dot{e}) = \dot{0} \\ \bot & otherwise \end{cases}$$

$$\begin{gathered}(i_1 : \dot{n}_1, \dots).i_j \leftarrow \dot{n}' = (i_1 : \dot{n}_1, \dots, i_j : \dot{n}', \dots) \\ T^{\leftarrow}_{i,\dot{e},\dot{e}'} = \lambda n. \begin{cases} r.i \leftarrow r(\dot{e}) & T^{\neq}_{\dot{e}'}(n) = r \\ T(n) & otherwise \end{cases}\end{gathered}$$

Here $T^{\neq}_{\dot{e}}$ restricts the domain of T to the records satisfying $\dot{e}$, while $T^{=}_{\dot{e}}$ restricts to the opposite. Finally $T^{\leftarrow}_{i,\dot{e},\dot{e}'}$ computes a new map where columns i of records r that satisfy $\dot{e}'$ are updated to $r(\dot{e})$. Now with the intent that $\bot(\dot{e}) = \bot$ the semantic rules are defined as follows.

$$\nu, \mathrm{select}\, \dot{e}\, \mathrm{from}\, \hat{t}\, \mathrm{where}\, \dot{e}' \Rightarrow \nu, \lambda n. \left(\nu(\hat{t})^{\neq}_{\dot{e}'}(n)(\dot{e}_1), \dots\right)$$

$$\begin{aligned}&\nu, \mathrm{insert}\, i_j = \dot{n}\, \mathrm{into}\, \dot{t} \Rightarrow \\ &\qquad \nu[\dot{t} := \nu(\dot{t}) + (i_1 : \dot{0}, \dots, i_j : \dot{n}, \dots)], \emptyset \\ &(\mathit{where}\ +\ \mathit{extends}\ \nu(\dot{t})'\mathit{s\ domain})\end{aligned}$$

$$\nu, \mathrm{update}\, i = \dot{e}\, \mathrm{in}\, \dot{t}\, \mathrm{where}\, \dot{e}' \Rightarrow \nu[\dot{t} := \nu(\dot{t})^{\leftarrow}_{i,\dot{e},\dot{e}'}], \emptyset$$

$$\nu, \mathrm{delete\, from}\, \dot{t}\, \mathrm{where}\, \dot{e} \Rightarrow \nu[\dot{t} := \nu(\dot{t})^{=}_{\dot{e}}], \emptyset$$

We assign security levels to tables for two functions. First, all tables have a simple level that describes the confidentiality of the table itself. Observers not at or above this level cannot access a table at all. Second, we assign security levels to columns, that is, all records have the same security level for the corresponding columns. We denote the mapping of column names of a table to security levels by δ, and $\ell_\delta = \bigsqcap \delta(i)$. For simplicity, we map the table security level to the synthetic column name *table*.

For typing purposes, we collect all δ in Δ which maps from table names. For table expression names, we define the following evaluation.

$$\Delta(\dot{t}_1\, \mathrm{join}\, \dot{t}_2\, \mathrm{on}\, i_1 = i_2) = \\ \lambda i. \begin{cases} \bigsqcup_{i \in \{1,2\}} \left(\Delta(\dot{t}_i)(\mathit{table}) \sqcup \Delta(\dot{t}_i)(i_i)\right) & i = \mathit{table} \\ \Delta(\dot{t}_1)(i) & i \in \Delta(\dot{t}_1) \\ \Delta(\dot{t}_2)(i) & i \in \Delta(\dot{t}_2) \wedge i \notin \Delta(\dot{t}_1) \\ \bot & \text{else} \end{cases}$$

The type system is defined by the following sets of rules, where $\dot{\Gamma}$ is a typing environment mapping variables to security levels. Note that typing of expressions refers to a table typing δ, whereas statement typing uses Δ. Well-formedness requirements for δ and Δ are standard.

$$\delta, \dot{\Gamma} \vdash \dot{n} : \ell \quad \delta, \dot{\Gamma} \vdash i : \delta(i) \quad \delta, \dot{\Gamma} \vdash \dot{x} : \dot{\Gamma}(\dot{x})$$

$$\frac{\delta, \dot{\Gamma} \vdash \dot{e} : \ell_1 \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell_2}{\delta, \dot{\Gamma} \vdash \dot{e} \oplus \dot{e}' : \ell_1 \sqcup \ell_2} \quad \frac{\delta, \dot{\Gamma} \vdash \dot{e} : \ell \quad \ell \sqsubseteq \ell'}{\delta, \dot{\Gamma} \vdash \dot{e} : \ell'}$$

$$\frac{\delta = \Delta(\hat{t}) \quad \forall i. \delta, \dot{\Gamma} \vdash \dot{e}_i : \ell_i \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell \quad \ell \sqcup \delta(\mathit{table}) \sqsubseteq \ell'}{\Delta, \dot{\Gamma} \vdash \mathrm{select}\, \dot{e}_1, \dots, \dot{e}_n\, \mathrm{from}\, \hat{t}\, \mathrm{where}\, \dot{e}' : (\ell_1, \dots, \ell_n)^{\ell'} / \top\, \mathrm{ok}}$$

$$\frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \ell \quad \ell \sqsubseteq \delta(i)}{\Delta, \dot{\Gamma} \vdash \mathrm{insert}\, i = \dot{e}\, \mathrm{into}\, \dot{t} : \bot^{\bot} / \ell_\delta\, \mathrm{ok}} \quad \frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \delta(\mathit{table})}{\Delta, \dot{\Gamma} \vdash \mathrm{delete\, from}\, \dot{t}\, \mathrm{where}\, \dot{e} : \bot^{\bot} / \ell_\delta\, \mathrm{ok}}$$

$$\frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \ell_1 \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqsubseteq \delta(i)}{\Delta, \dot{\Gamma} \vdash \mathrm{update}\, i = \dot{e}\, \mathrm{in}\, \dot{t}\, \mathrm{where}\, \dot{e}' : \bot^{\bot} / \delta(i)\, \mathrm{ok}}$$

For SQL we have to formalize to notions of indistinguishability. We use a projection to erase all information in a record that is not typed at or below ℓ, and lift it to sequences.

$$\downarrow^{\delta}_{\ell}(r).i = \begin{cases} r.i & \delta(i) \sqsubseteq \ell \\ 0 & \text{otherwise} \end{cases} \qquad \downarrow^{\delta}_{\ell}(f) = \lambda n. \downarrow^{\delta}_{\ell}(f(n))$$

We define two result sets as indistinguishable with respect to a type $\ell^{\ell'}_{\bullet}$ (where $\ell_{\bullet}$ defines a sequence of labels), abusing notation of projection, as $s_1 \dot{\sim}_{\ell^{\ell'}_{\bullet}, \ell_o} s_1 \iff \ell' \sqsubseteq \ell_o \implies (|s_1| = |s_2| \wedge \downarrow^{\ell_{\bullet}}_{\ell_o}(s_1) = \downarrow^{\ell_{\bullet}}_{\ell_o}(s_2))$. Projection is also used to define indistinguishability of two table states, which are indistinguishable with respect to observer level ℓ if they agree on all columns at most ℓ: $T_1 \dot{\sim}_{\delta,\ell} T_2 \iff \delta(\mathit{table}) \sqsubseteq \ell \implies \downarrow^{\delta}_{\ell}(T_1) = \downarrow^{\delta}_{\ell}(T_2)$. Finally, indistinguishability of table states is lifted component-wise to database states: $\nu_1 \dot{\sim}_{\Delta,\ell} \nu_2 \iff \forall t \in \mathcal{T}. \nu_1(t) \dot{\sim}_{\Delta(t),\ell} \nu_2(t)$

B Proofs

B.1 SQL

The proof of SQL noninterference relies on two auxiliary lemmas about table projections. The first one formulates a statement for restrictions in indistinguishable inputs.

Lemma 3 (Restriction Indistinguishability). *Assume δ, $\ell_e \sqsubseteq \ell_o$, ℓ_x, T_1, T_2, $\dot{n}_1$, $\dot{n}_2$, and $\dot{e}$ such that $\delta, [\dot{x} : \ell_x] \vdash \dot{e} : \ell_e$, $T_1 \dot{\sim}_{\delta,\ell_o} T_2$, and $\dot{n}_1 \dot{\sim}_{\ell_x,\ell_o} \dot{n}_2$, then $T_1{}^{\neq}_{\dot{e}[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_1{}^{\neq}_{\dot{e}[\dot{x}:=\dot{n}_2]}$ and $T_1{}^{=}_{\dot{e}[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_1{}^{=}_{\dot{e}[\dot{x}:=\dot{n}_2]}$.*

Proof. First assume that $\delta(\mathit{table}) \sqsubseteq \ell_o$, the other case is trivial. Then $\downarrow^{\delta}_{\ell_o}(T_1) = \downarrow^{\delta}_{\ell_o}(T_2)$. Next, either $\dot{e}$ does not contain $\dot{x}$, or $\ell_x \sqsubseteq \ell_e$. In the latter case, $\dot{n}_1 = \dot{n}_2$. So always $\dot{e}[\dot{x} := \dot{n}_1] = \dot{e}[\dot{x} := \dot{n}_2] = \dot{e}'$. Now for each $n \in \mathit{dom}(T_1) = \mathit{dom}(T_2)$, we have $T_1(n)(\dot{e}') = T_2(n)(\dot{e}')$, because the records must agree on the columns referenced in $\dot{e}'$ by $\downarrow$. Thus, a record gets selected for $T_1{}^{\neq}_{\dot{e}'}$ if and only if it gets selected for $T_2{}^{\neq}_{\dot{e}'}$, and selected for $T_1{}^{=}_{\dot{e}'}$ if and only if it gets selected for $T_2{}^{=}_{\dot{e}'}$. Since the records are not modified otherwise, it follows that $T_1{}^{\neq}_{\dot{e}'} \dot{\sim}_{\delta,\ell_o} T_1{}^{\neq}_{\dot{e}'}$ and $T_1{}^{=}_{\dot{e}'} \dot{\sim}_{\delta,\ell_o} T_1{}^{=}_{\dot{e}'}$.

This lemma states that typed table joins from indistinguishable inputs create indistinguishable outputs.

Lemma 4 (Join Noninterference). *Assume Δ, ℓ_o, ν_1 and ν_2 such that $\nu_1 \dot{\sim}_{\Delta,\ell_o} \nu_2$. Then $\nu_1(\hat{t}) \dot{\sim}_{\Delta(\hat{t}),\ell_o} \nu_2(\hat{t})$.*

Proof. By analysis of $\hat{t}$.

$\hat{t} = \dot{t}$. Follows by the definition of $\dot{\sim}_{\Delta,\ell_o}$.

$\hat{t} = \dot{t}_1 \operatorname{join} \dot{t}_2 \operatorname{on} i_1 = i_2$. Let $\delta_1 = \Delta(\dot{t}_1)$, $\delta_2 = \Delta(\dot{t}_2)$, $T_1^1 = \nu_1(\dot{t}_1)$, $T_2^1 = \nu_1(\dot{t}_2)$, $T_1^2 = \nu_2(\dot{t}_1)$, and $T_2^2 = \nu_2(\dot{t}_2)$. Furthermore, let $\delta_j = \Delta(\hat{t})$ and $T_j^1 = \nu_1(\hat{t})$ and $T_j^2 = \nu_2(\hat{t})$. Let $\delta_j(\mathit{table}) \sqsubseteq \ell_o$, the other case is trivial. Then by construction $\delta_1(\mathit{table}) \sqsubseteq \ell_o$ and $\delta_2(\mathit{table}) \sqsubseteq \ell_o$. Thus $\downarrow_{\ell_o}^{\delta_1}(T_1^1) = \downarrow_{\ell_o}^{\delta_1}(T_1^2)$ and $\downarrow_{\ell_o}^{\delta_2}(T_2^1) = \downarrow_{\ell_o}^{\delta_2}(T_2^2)$. We need to show that $T_j^1 \dot{\sim}_{\delta_j,\ell_o} T_j^2$. First note that δ_j agrees on the levels with δ_1 and δ_2 for the respective columns. Next, selecting records to join in $\oplus$ happens at $\delta_1(i_1)$ and $\delta_2(i_2)$. By this case, these are $\delta_1(i_1) \sqsubseteq \ell_o$ and $\delta_2(i_2) \sqsubseteq \ell_o$. Thus the original values are retained in $\downarrow_{\ell_o}^{\delta_j}(\ldots)$, which by indistinguishability means that the respective tables have equivalent values in those columns. Thus corresponding records are merged for both inputs. Pick a generated record. For any column typed at or below ℓ_o, we have that the corresponding column from T^1 or T^2 is also typed at or below ℓ_o by construction of join. Thus, the corresponding values in T_1 and T_2 are equivalent, which means that the column in the joined records are equivalent. Thus, $\downarrow_{\ell_o}^{\delta_j}(T_j^1) = \downarrow_{\ell_o}^{\delta_j}(T_j^2)$.

Finally, the proof of SQL noninterference is a case analysis, invoking the previous two lemmas in the case of select queries.

Theorem 1 (SQL Noninterference).

$$\begin{array}{l}\forall \Delta, \ell_\bullet, \ell', \ell'', \ell_x, c, \mu_1, \mu_2, \mu_1', \mu_2', \dot{n}_1, \dot{n}_2, s_1, s_2. \\ \Delta, x : \ell_x \vdash c : \ell_\bullet^{\ell'} / \ell'' \text{ ok} \wedge \mu_1 \dot{\sim}_{\Delta,\ell_o} \mu_2 \wedge \dot{n}_1 \dot{\sim}_{\ell_x,\ell_o} \dot{n}_2 \wedge \\ \mu_1, c[x := \dot{n}_1] \Rightarrow \mu_1', s_1 \wedge \mu_2, c[x := \dot{n}_2] \Rightarrow \mu_2', s_2 \wedge \\ \implies \mu_1' \dot{\sim}_{\Delta,\ell_o} \mu_2' \wedge s_1 \dot{\sim}_{\ell_\bullet^{\ell'},\ell_o} s_2\end{array}$$

Proof. By case analyis for c. We use the following definitions where applicable: $\delta = \Delta(\dot{t})$, $T_1 = \nu_1(\dot{t})$, $T_1' = \nu_1'(\dot{t})$, $T_2 = \nu_2(\dot{t})$ and $T_2' = \nu_2(\dot{t})$.

$\operatorname{select} \dot{e}_\bullet \operatorname{from} \hat{t} \operatorname{where} \dot{e}'$. The semantics of select shows that database states remain unchanged, i.e., $\nu_1 = \nu_1'$ and $\nu_2 = \nu_2'$. Thus, by premises, the output states are indistinguishable at ℓ_o. Now a case decision on the typing of the result sets. If $\ell' \not\sqsubseteq \ell_o$, then the results are automatically indistinguishable by construction. So assume that $\ell' \sqsubseteq \ell_o$. Let $T_1 = \nu_1(\hat{t})$, $T_2 = \nu_2(\hat{t})$, and $\delta = \Delta(\hat{t})$. By Lemma 4, it follows that $T_1 \dot{\sim}_{\delta,\ell_o} T_2$. Next, by Lemma 3, we have that $T_1{}^{\neq}_{\dot{e}'[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_2{}^{\neq}_{\dot{e}'[\dot{x}:=\dot{n}_2]}$. This means that the same number of records gets selected from each table, which means that the outputs have the same number of elements, that is, $|s_1| = |s_2|$.
Now assume also that $\ell_i \sqsubseteq \ell_o$. Then either $\dot{e}_i$ does not include $\dot{x}$, or $\ell_x \sqsubseteq \ell_o$. So, $\dot{e}_i[\dot{x} := \dot{n}_1] = \dot{e}_i[\dot{x} := \dot{n}_2]$. Furthermore, it follows that $\dot{e}_i$ only uses columns typed at or below ℓ_o. As derived above, $T_1{}^{\neq}_{\dot{e}'[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_2{}^{\neq}_{\dot{e}'[\dot{x}:=\dot{n}_2]}$, or $\downarrow_{\ell_o}^{\delta}(T_1{}^{\neq}_{\dot{e}'[\dot{x}:=\dot{n}_1]}) = \downarrow_{\ell_o}^{\delta}(T_2{}^{\neq}_{\dot{e}'[\dot{x}:=\dot{n}_2]})$. Thus, the selected records agree on columns at or below ℓ_o. Then the evaluation of $\dot{e}_i[\dot{x} := \dot{n}_{1/2}]$ will create the same result for corresponding records. Thus, the projected records have the same values for the result of $\dot{e}_i$. It follows that $\downarrow_{\ell_o}^{\ell_\bullet}(s_1) = \downarrow_{\ell_o}^{\ell_\bullet}(s_2)$.

$\mathrm{update}\, i = \dot{e} \,\mathrm{in}\, \dot{t} \,\mathrm{where}\, \dot{e}'$ For an update, the result sequences are obviously equivalent and thus indistinguishable. Assume that $\delta(\mathit{table}) \sqsubseteq \ell_o$, the opposite case is trivial. First, only the table $\dot{t}$ is potentially modified. All other tables remain the same. Second, the size of $\dot{t}$ doesn't change, only record data is modified. Now several case decisions on whether (1) $\ell_1 \sqsubseteq \ell_o$ and (2) $\ell_2 \sqsubseteq \ell_o$.

FF,FT,TF. Then $\delta(i) \not\sqsubseteq \ell_o$. It follows that since $\downarrow^{\delta}_{\ell_o}(T_1) = \downarrow^{\delta}_{\ell_o}(T_2)$ we have $\downarrow^{\delta}_{\ell_o}(T_1') = \downarrow^{\delta}_{\ell_o}(T_2')$.

TT. Then neither $\dot{e}$ nor $\dot{e}'$ do contain x, or $\ell_x \sqsubseteq \ell_o$, so that $\dot{n}_1 = \dot{n}_2$. It follows that $\dot{e}[x := \dot{n}_1] = \dot{e}[x := \dot{n}_2]$ and $\dot{e}'[x := \dot{n}_1] = \dot{e}'[x := \dot{n}_2]$ and does not use columns or variables not at or below ℓ_o. Since $T_1 \dot{\sim}_{\delta,\ell_o} T_2$ it follows that corresponding records are selected for modification, and that the same updates are computed for each. Thus, $\downarrow^{\delta}_{\ell_o}(T_1') = \downarrow^{\delta}_{\ell_o}(T_2')$.

$\mathrm{insert}\, i = \dot{e} \,\mathrm{into}\, \dot{t}$ Analogous to update, the result set indistinguishability is obvious. Assume that $\delta(\mathit{table}) \sqsubseteq \ell_o$, the opposite case is trivial. In general, by indistinguishability of the input states we have that the inputs have the same number of records, which means that the outputs have the same number of records, namely one more. If $\ell_\delta \not\sqsubseteq \ell_o$, then all columns are not at or below ℓ_o. In that case, $\downarrow^{\delta}_{\ell_o}(T_1') = \downarrow^{\delta}_{\ell_o}(T_2')$. Now assume the opposite. We have that the type of $\dot{e}$ is at or below $\delta(i)$ by premise. If $\delta(i) \not\sqsubseteq \ell_o$, then by the argument above the value is not visible in the projection, so that again $\downarrow^{\delta}_{\ell_o}(T_1') = \downarrow^{\delta}_{\ell_o}(T_2')$. If however $\delta(i) \sqsubseteq \ell_o$, then if $\dot{e}$ contains x, then $\ell_x \sqsubseteq \ell_o$, which means that $\dot{n}_1 = \dot{n}_2$. So always $\dot{e}[x := \dot{n}_1] = \dot{e}[x := \dot{n}_2]$. Thus, the value computed is the same for both runs. Thus the additional records in the projection are equivalent.

$\mathrm{delete\,from}\, \dot{t} \,\mathrm{where}\, \dot{e}$ Analogous to update and insert, the result set indistinguishability is obvious. Assume that $\delta(\mathit{table}) \sqsubseteq \ell_o$, the opposite case is trivial. Then if $\dot{x}$ appears in $\dot{e}$ it holds that $\dot{n}_1 = \dot{n}_2$. Thus, always $\dot{e}[x := \dot{n}_1] = \dot{e}[x := \dot{n}_2]$. Furthermore, since the inputs are indistinguishable at ℓ_o, it follows that corresponding records will be deleted. Thus, the outputs will be indistinguishable, too, since they will be restricted in the same way.

B.2 While

The proof of this theorem requires a number of auxiliary lemmas. We start with a statement that allows to type any program at a single arbitrary level.

Lemma 5 (Single-level Typability). *For all programs c and expressions e and security levels ℓ, there is Γ_ℓ such that $\Gamma_\ell \vdash c : \ell\,\mathrm{ok}$ and all variables are mapped to ℓ in Γ_ℓ.*

Proof. Let $\mathcal{V}$ be the set of variables mentioned in c or e. Then define Γ_ℓ as mapping all variables in $\mathcal{V}$ to ℓ. Now proceed by structural induction on c and e. We show select cases.

Literal: The typing rule is polymorphic in the security level. Thus, typing at ℓ is permissible.

Variable: The variable x is an element of $\mathcal{V}$. Thus, $\Gamma_\ell(x) = \ell$.

Assignment: By inductive hypothesis, the expression can be typed as ℓ. Furthermore, analogously to the previous case, the variable is typed as ℓ. Thus, assignment is permissible at $\ell\, ok$.

Condition: By inductive hypothesis, the condition as well as the branches are typable at ℓ. Thus, if is typable at $\ell\, ok$.

The next lemma states the existence of a typed program that computes the composition of two given typed programs.

Lemma 6 (WHILE Composition). *Given two programs c_a and c_b that are typed under Γ_a and Γ_b and compute f_a and f_b, respectively, where outputs of c_b agree with inputs of c_a, there exists a program $c_{a\circ b}$ that is typed under $\Gamma_{a\circ b}$ and computes $f_{a\circ b} = f_a \circ f_b$. Furthermore, $\Gamma_{a\circ b}$ agrees with Γ_a and Γ_b under respective renamings of variables.*

Proof. For simplicity we assume that c_a and c_b agree on the variables that are used to pass results, that is, outputs of c_b with respect to f_b are named the same as inputs of c_a with respect to f_a. We denote those variables by $\overrightarrow{y}$. No other variables are shared. Note that this can be accomplished by consistently renaming variables. Now $\Gamma_a(\overrightarrow{y}) = \Gamma_b(\overrightarrow{y})$.

Let $\Gamma_{a\circ b}(x)$ be $\Gamma_a(x)$ if x appears in c_a, and let $\Gamma_{a\circ b}(x)$ be $\Gamma_b(x)$ if x appears in c_b. Let $c_{a\circ b} = c_b; c_a$. Inspection of the semantic rule for sequencing shows that this program fulfills the functional requirements, that is, computes $f_{a\circ b}$.

By construction, c_a and c_b can be typed under $\Gamma_{a\circ b}$. Namely, $\Gamma_{a\circ b}$ is a weakening of both Γ_a and Γ_b. Thus, by inspection of the typing rule for sequencing $c_{a\circ b}$ is typable under $\Gamma_{a\circ b}$.

Theorem 2 (Security Completeness). *If a function f is computable in WHILE, and noninterferent under a signature given by Γ, then there exists a WHILE program c that is typable under a typing environment Γ' that is an extension of Γ.*

Proof. Let c_f be the program computing f which is guaranteed to exist by computability. We will show that we can separate inputs and outputs such that the partial computations are typable. Take an arbitrary variable x in c_f. We will construct a program c_x that computes the final value of x after c_f and which is typable. Given $\mathcal{S}$, we have a corresponding typing environment Γ_f. Let $\ell_x = \Gamma_f(x)$. We instantiate noninterference at level ℓ_x. This states that the result of x will be the same under input states that are indistinguishable with respect to ℓ_x, that is, for states where all variables y typed below ℓ_x or equal to ℓ_x are the same. It follows that we can choose arbitrary values for the other variables, e.g., the constant 0.

For each variable y typed below or equal to ℓ_x, create a new variable y' typed at ℓ_x. This includes x, which yields x'. For each such variable z not typed below ℓ_x, create a new variable z' typed at ℓ_x. Create the program c_f' by replacing all y and z in c_f with the corresponding y' and z'. This function is obviously equivalent to c_f with respect to x and thus noninterferent.

Let c_y be the program that initializes all variables y' to the values of y, e.g., a sequence of $y' := y$. This is typable as ℓ_x ok, since all y are typed at or below y', which are typed at ℓ_x. Next, let c_0 be the program that initializes all variables z' to zero, e.g., a sequence of $z' := 0$. This is typable as ℓ_x ok, since zero can be typed at ℓ_x. Finally, c'_f is typable at level ℓ_x by Lemma 5.

We can compose c_y, c_0 and c'_f, which yields the program c_x typable at ℓ_x ok by Lemma 6. Now for two inputs indistinguishable at level $\ell \sqsupseteq \ell_x$ we have that the final values of x' are equivalent. It follows that the value for x' is equivalent to the required value of x.

After constructing programs c_x for each variable x, we only need to compose them such that each variable x' is computed. Typability is ensured by composition. If an exact match is preferred, all variables x' can be copied back to the original x. This is permissible since the variables have the same typings.

Lemma 1 (Substitution Typable). *Assume a context $E[\bullet]$, that is, a composed program with a statement hole. Furthermore assume an $x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$, Γ, Δ and ℓ such that $\Gamma, \Delta \vdash Eiona[x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}] : \ell\,\mathrm{ok}$. Then there exists an extension Γ' of Γ such that $\Gamma', \Delta \vdash E[c_{eval}] : \ell\,\mathrm{ok}$.*

Proof. First note that this is not a traditional substitution lemma, since the substituted element is not just a variable. Furthermore, the typing environment is actually growing since we need to type the temporaries of the simulation. The proof is an induction over the derivation of the typing of $E[x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}]$.

$E = \bullet$: Then the statement is correct by the proofs in the previous subsections.

$E = x := e$ or $E = x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$ Vacuous, no nested (composed-level) statement.

$E = c; E'$ or $E = E'; c$: In both cases typing ends with the sequencing rule, which yields that both c and $E'[x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}]$ are typed under Γ and Δ. By inductive hypothesis, $E'[c_{eval}]$ is typed under a Γ' and Δ, where Γ' is an extension of Γ. By weakening, c can be typed under Γ'. By the sequencing rule, we gain a complete typing of the sequence again. The cases for conditionals $\mathrm{if}\, e \,\mathrm{then}\, c \,\mathrm{else}\, E'/\mathrm{if}\, e \,\mathrm{then}\, E' \,\mathrm{else}\, c$ and loop $\mathrm{while}\, e \,\mathrm{do}\, E'$ are analogous.

Lemma 2 (Eval-free Composed To Host). *If a statement c that does not contain* eval *is typed under Γ and Δ as $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$, then c can be typed as $\Gamma \vdash c : \ell\,\mathrm{ok}$.*

Proof. By induction on the derivation of $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$. Since c does not contain eval, no sub-statement contains an eval, either. Thus the inductive hypothesis applies. Furthermore, the eval typing rule cannot appear in the derivation, because it only appears to eval. We can thus reconstruct a host-level typing corresponding to the (copied) composed-level typing rule.

Theorem 3 (Simulation Equivalence up to Γ). *If $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$, then there exists an extension Γ' such that $\Gamma' \vdash c_{eval} : \ell\,\mathrm{ok}$, and for all $\mu_1, \mu_1', \nu, \nu', \mu_2$ where $\Gamma \vdash \mu_1\,\mathrm{ok}$, $\Gamma' \vdash \mu_2\,\mathrm{ok}$ and μ_2 is an extension of μ_1 such that the extension encodes ν, and $\mu_1, \nu, c \Rightarrow \mu_1', \nu'$, then there exists μ_2' an extension of μ_1' such that $\mu_2, c_{eval} \Rightarrow \mu_2'$ and the extension encodes ν'.*

Proof. We find Γ' by Lemma 1. Now induction on the derivation of $\mu_1, \nu, c \Rightarrow \mu_1', \nu'$. We show select cases.

$x := e$. This base case results in $c_{eval} = c$. Then we can set $\mu_2' = \mu_2[x := \mu_2(e)]$, which is an extension of $\mu_1' = \mu_1[x := \mu_1(e)]$, because e is restricted to variables in μ_1 and μ_2 agrees with μ_1 on those. The extension also encodes ν' correctly, because $\nu = \nu'$ and x is the only changed mapping.

$\mathrm{while}\, e\, \mathrm{do}\, c'$. We treat the case that $\mu_1(e) \neq 0$, the other is analogous and simpler. First, $c_{eval} = \mathrm{while}\, e\, \mathrm{do}\, c'_{eval}$. e can only refer to variables in the domain of μ_1 because of c being well-typed. μ_2 agrees with μ_1 on all variables in μ_1. Thus $\mu_2(e) = \mu_1(e) \neq 0$. Now, by inductive hypothesis, because $\mu_1, \nu, c' \Rightarrow \mu_1'', \nu''$ and c is typed, there exists μ_2'' such that $\mu_2, c'_{eval} \Rightarrow \mu_2''$, where μ_2'' is an extension of μ_1'' that encodes ν''. Also, by inductive hypothesis, because $\mu_1'', \nu'', \mathrm{while}\, e\, \mathrm{do}\, c' \Rightarrow \mu_1', \nu'$, there exists μ_2' such that $\mu_2'', \mathrm{while}\, e\, \mathrm{do}\, c'_{eval} \Rightarrow \mu_2'$, where μ_2' is an extension of μ_1' that encodes ν'. The cases for if and sequence are similar.

$x := \mathrm{eval}\, e\, \mathrm{in}\, \dot{c}$. Then $c_{eval} = c'; x := x'$, where c' is the simulation of eval and the assignment writes back the value from temporary x'. By the properties of the simulation we have that μ_2'', the state after evaluating c', is identical to μ_1 on the domain of μ_1, because the pure simulation is constructed to not affect original variables. The extension part however encodes ν' by property of being a simulation. Finally, $\mu_2' = \mu_2''[x := \mu_2''[x']]$, and we have $\mu_2''[x'] = \mu_1'[x]$ by virtue of the simulation. Thus, the extension part of μ_2' over μ_1' encodes ν', and μ_2' agrees with μ_1' on the domain of μ_1'.

B.3 Declassification

Auxilliary Lemmas

Lemma 9 (Single Exit). *For all states σ in a valid system $\mathcal{S}$ with $\Gamma(E(\sigma.l)) \sqsubset \Gamma(\sigma.l)$, it holds for all traces $\tau \in \mathcal{T}(\mathcal{S}, \sigma)$ generated from σ that one of the following is satisfied*

- $\forall i \leq len(\tau).\ pc_E \sqsubset \Gamma(\tau(i).l)$
- $\tau = \tau_1 \mapsto \sigma_E \mapsto \tau_2$, $\forall i \leq len(\tau_1).\ pc_E \sqsubset \Gamma(\tau_1(i).l)$, *and* $\sigma_E.l = E(\sigma.l)$

where $pc_E = \Gamma(E(\sigma.l))$.

Proof. Let τ be a trace generated by σ. Let $l = \sigma.l$ and $l_E = E(\sigma.l)$. Let $\ell = \Gamma(l)$. Note that $pc_E \sqsubset \ell$. Let k be the first index in τ such that $\ell \not\sqsubseteq \Gamma(\tau(k).l)$.

If such a k exists, it must be $k > 0$. Now for the prefix of τ made up of the first $k-1$ elements, we have that $\forall i < k. \ell \sqsubseteq \Gamma(\tau(i).l)$. We apply part 1 of validity iteratively to the prefix. Let l_1 be the location before a step, and l_2 be the location after the step. Then before a step, we have $l_1 \neq l_E$, $\ell \sqsubseteq \Gamma(l_1)$, and $\exists j.\ E^j(l_1) = l_E \wedge \forall 0 \leq i < j. \ell \sqsubseteq \Gamma(E^i(l_1))$. By part 1 of validity, $\exists m \geq 0. E(l_1) = E^m(l_2)$ and $\forall 0 \leq i < m.\ \Gamma(l_1) \sqsubseteq \Gamma(E^i(l_2))$. If $m = 0$, then we have $l_2 = E(l_1)$. Now also $\ell \sqsubseteq \Gamma(l_2)$. Thus $l_2 \neq l_E$, so $\exists j. E^j(l_2) = l_E$ and $\forall 0 \leq i < j. \ell \sqsubseteq \Gamma(E^i(l_2)$. If $m = 1$, then the exit-chain property is preserved immediately. If $m > 1$, then $\Gamma(E(l_1)) \sqsubseteq \Gamma(l_1)$ (part 2 of validity) and $\Gamma(l_1) \sqsubseteq \Gamma(E(l_2))$ and $\Gamma(l_1) \sqsubseteq \Gamma(l_2)$ (part 1), so by transitivity the exit-chain property is preserved. We thus have $\tau_1 = \tau(0) \mapsto^* \tau(k-1)$ with $\forall i \leq len(\tau_1) = k-1.\ pc_E \sqsubset \ell \sqsubseteq \Gamma(\tau_1(i).l)$.

Now inspect $\tau(k-1) \mapsto \tau(k)$. Again part 1 of validity applies. It cannot be the case that $m \geq 1$, as then k was not minimal. So $m = 0$. Then $\tau(k).l = E(\tau(k-1).l)$. From the above property of τ_1, we have that $\exists j.\ E^j(\tau(k-1).l) = l_E \wedge \forall 0 \leq i < j. \ell_E \sqsubset \Gamma(E^i(\tau(k-1).l)$. It follows that $j = 1$, as all other cases are contradictory: $j = 0$ implies $\tau(k-1).l = l_E$ and so k not minimal; $j > 1$ implies $\ell_E \sqsubset \Gamma(E(\tau(k-1).l)) = \Gamma(\tau(k).l)$. So $\tau(k).l = l_E$.

The case when k does not exist follows the derivation of τ_1 above.

Translation We partition M straightforwardly such that a variable x falls into the component given by $\Gamma(x)$. We could map c directly to l. However, for technical reasons it is easier to annotate statements with integers in the tradition of program analysis. The abstract location can then be equated to an instruction pointer and we implicitly restrict ourselves only to statements that may appear in program runs. We then let $\mapsto$ over these states according to the small-step semantics of the language. *All* possible attacks are integrated into $\mathcal{S}$. If a state corresponds to a location denoting a hole, we add transitions covering any noninterfering computation that does not modify high-integrity variables. This basically collapses attack code into a big-step transition. If l denotes a hole in $c[\bullet]$, then $A(l)$. We let Γ of $\mathcal{S}$ be the pc of the judgment of the corresponding location in the tree of $\Gamma, pc \vdash c[\bullet]$. We let E be the location after the end of the immediately enclosing control statement.

This translation produces a valid system.

Lemma 10 (Translation is valid). *The translation of $\Gamma, pc \vdash c[\bullet]$ is valid.*

Proof. Validity is defined as:

- $\forall \sigma \mapsto \sigma'.\ \exists k \geq 0.\ E(\sigma.l) = E^k(\sigma'.l) \wedge \forall 0 \leq i < k.\ \Gamma(\sigma.l) \sqsubseteq \Gamma(E^i(\sigma'.l))$
  Let $l = \sigma.l$ and $l' = \sigma'.l$. $\mapsto$ is generated by the small-step semantics. Induction over (unrolled) semantic rules. We show select cases.
  - $[x := a]^l ; [c]^{l'} \to [c]^{l'}$.
    Then $E(l) = E(l')$ and $\Gamma(l) = \Gamma(l')$ and case $k = 1$ applies.
  - $[\text{if } e \text{ then } [c_1]^{l_1} \text{ else } [c_2]^{l_2} \text{ end}]^l ; [c_3]^{l_3} \to [c_1]^{l_1} ; [c_3]^{l_3}$, where $e \Downarrow \text{true}$ and $\Gamma(e) = H$.
    Then $\Gamma(l_1) = H$ and $E(l_1) = l_3$ and $E(l) = E(l_3)$. Thus case $k = 2$ applies.
  - $[c_1]^{l_1} ; [c_3]^{l_3} \to [c^3]^{l_3}$ (continued from above).
    Then, as $l_3 = E(l_1)$, the case $k = 0$ applies.

– $\forall l.\mathit{\Gamma}(E(l)) \sqsubseteq \mathit{\Gamma}(l)$
  Let c' be the control structure immediately enclosing l. Then the *pc* of c' is lower or equal to the *pc* at l and equal to the one at $E(l)$. Thus, $\mathit{\Gamma}(E(l)) \sqsubseteq \mathit{\Gamma}(l)$.
– $\forall \sigma_1 \mapsto \sigma_1', \sigma_2 \mapsto \sigma_2'.\sigma_1 \approx_{C/I} \sigma_2 \wedge \sigma_1.l = \sigma_2.l \wedge \sigma_1'.l \neq \sigma_2'.l \implies \mathit{\Gamma}(\sigma_1'.l) \in H_{C/I} \wedge \mathit{\Gamma}(\sigma_2'.l) \in H_{C/I}$.
  The only way that successor locations can vary is if $l = \sigma_1.l = \sigma_2.l$ denotes a control structure. If different confidentiality/integrity-equivalent lead to different branches, the condition must depend on the hi-confidentiality/lo-integrity part of the state. Then, since the program is typed, the *pc* at $\sigma_1'.l$ and $\sigma_2'.l$ must be hi-confidential/lo-integrity.
– $\forall l.D(l) \implies \mathit{\Gamma}(l) \in H_I \wedge \mathit{\Gamma}(l) \in L_C$
  This follows immediately from the typing rules.
– $\forall l.A(l) \implies \mathit{\Gamma}(l) \in L_C$
  This follows immediately from the typing rules.

With this setup, one can show the following theorem.

Theorem 4 (Language-robust implies step-wise robust). *If $\mathit{\Gamma}, pc \vdash c[\bullet]$, then $\mathcal{S}$ constructed as above is (a) able to simulate all runs of* c *under any attack a and (b) $\mathcal{S}$ is step-wise robust.*

We first note that we investigate a termination-sensitive version of [16]. We feel this is valid because, in fact, proofs to the soundness of the type system proposed in [16] are only correct if the type system is termination-sensitive.

For the first part, let l_c be the location assigned to $c[\bullet]$. Take an arbitrary M_0 and a, and generate the (deterministic) execution started by $\langle M_o, c[a]\rangle$.

Proceed by induction on the length of an execution prefix. For all steps $\langle M^1, c^1\rangle \to \langle M^2, c^2\rangle$ that are not part of executing a, the result follows immediately since $\mapsto$ is generated by the small-step semantics. So assume c^1 is part of executing a. Then there is a (possibly empty) prefix of the computation such that the prefix ends just before executing a, that is, $\langle M', a; c'\rangle$. This prefix is matched by σ'. Since a is an attack, it is a noninterfering, high-integrity-preserving, non-declassifying computation. Thus, if a terminates, there exists σ'' such that $\sigma' \mapsto_A \sigma''$ and σ'' is updated with the effect of a on σ'. Complete the execution of a in c^1, that is, $\langle M'', c'\rangle$, as we assumed that a terminates on M'. Then it follows that $\langle M'', c'\rangle \equiv \sigma''$, as c' is the successor of the hole in $c[\bullet]$. On the other hand, if a does not terminate, then there won't be a transition from σ', and also $c[a]$ does not terminate. The stuck state simulates this non-termination.

For the second part, we first note that $\mathcal{S}$ is valid by the previous lemma. Further, all attacks ($\mapsto_A$) that were added are noninterfering, high-integrity preserving computations. This satisfies the first two attack properties. Now take two C-indistinguishable states σ_1 and σ_2 so that $A(\sigma_1.l)$. Then its label $l_1 = \sigma_1.l$ denotes a hole in c, and $\mathit{\Gamma}(l_1)$ is low-confidential. To be C-indistinguishable, $l_2 = \sigma_2.l$ must be either $= l_1$ or $\exists k.\ l_1 \in E^k(l_2)$ such that for all indices $i < k$ $\mathit{\Gamma}(l_1) \sqsubset \mathit{\Gamma}(E^i(l_2))$. In the first case, there must be an attack transition. In the second case, if the computation starting with σ_2 terminates, it must reach l_1, at which point there is an attack.

For non-attack transitions, we check each part of step-wise robustness.

- $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_I \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \implies \exists \tau_2'. \forall i < len(\tau_2'). \tau_2'(i) \approx_I \sigma_2 \wedge \tau_2'.e \approx_I \sigma_1'$.
  If the input states are indistinguishable, then either $l_1 = \sigma_1.l = \sigma_2.l = l_2$, or $\exists k_1, k_2.\ E^{k_1}(l_1) = E^{k_2}(l_2)$ where all smaller indices are not in L_I. In the first case, both states denote the same program location. As the program is typed, we can apply to Theorem 2 of [16]. The second case has three subcases. If $l_2 = E(l_1)$, then $\tau = \emptyset$ is the solution. If $l_1 = E^{k_2}(l_2)$, then we can use Lemma 9 to reduce to the first case, and otherwise we apply Lemma 9 twice to reduce to the first case.
- $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_C \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \wedge \neg D(\sigma_1) \implies \exists \tau_2'. \forall i < len(\tau_2'). \tau_2'(i) \approx_C \sigma_2 \wedge \tau_2'.e \approx_C \sigma_1'$
  Analogous to the first case. As $\neg D(\sigma_l)$, the location does not denote a declassification, and we can appeal to Theorem 1 of [16].
- $\forall \sigma_1, \sigma_2.\ \sigma_1 \approx_{I/C} \sigma_2 \wedge D(\sigma_1) \implies \exists \tau_2'. \forall i < len(\tau_2'). \tau_2'(i) \approx_{I/C} \sigma_2 \wedge D(\tau_2'.e)$
  Declassification is required to be in a low-confidentiality, high-integrity context by typing rules. Any equivalent state is either at the same location, or the declassification is the exit. We conclude with Lemma 9.
- $\forall \sigma, \sigma'.\ \sigma \mapsto \sigma' \wedge D(\sigma) \implies \sigma_{LL} = \sigma'_{LL}$
  We declassify high-integrity-high-confidentiality data to high-integrity-low-confidentiality data.

Theorem 5 (Step-wise robust implies robust). *If $\mathcal{S} = (\Sigma, \mapsto, L, D, A, E, \Gamma)$ is step-wise robust with respect to $\mapsto_A$, then $(\Sigma, \mapsto \setminus \mapsto_A)$ is robust with respect to A and $\approx_C$.*

We require some auxilliary notation and lemmas for this proof. Let $\overline{\mathcal{S}}$ be $\mathcal{S}$ without $\mapsto_A$, and $\overline{\mathcal{S}} \cup A'$ be the system when adding $\mapsto_{A'}$ to $\overline{\mathcal{S}}$.

Lemma 11. *If σ_1 and σ_2 are observationally equivalent under $\approx_C$, $\overline{\sigma}_1 \approx_I \sigma_1$, and $\overline{\sigma}_2 \approx_I \sigma_2$, then $\overline{\sigma}_1$ and $\overline{\sigma}_2$ are observationally equivalent under $\approx_C$.*

Proof. Observational equivalence can be stated as $\forall \overline{\tau}_1 \in \mathcal{T}(\mathcal{S}, \overline{\sigma}_1). \exists \overline{\tau}_2 \in \mathcal{T}(\mathcal{S}, \overline{\sigma}_2).\ \overline{\tau}_1 / \approx_C = \overline{\tau}_2 / \approx_C$. Induction on the length of $\overline{\tau}_1$. Assume $\overline{\tau}_1 = \overline{\sigma}_1 \mapsto \overline{\tau}_1'$ and let $\overline{\sigma}_1' = \overline{\tau}_1'(0)$. Case decision on $D(\overline{\sigma}_1)$.

If $\overline{\sigma}_1$ cannot declassify, then by step-wise robustness $\exists \overline{\tau}_2'$ with $\overline{\tau}_2'(0) = \overline{\sigma}_2$ such that $\overline{\tau}_2'$ preserves indistinguishability and $\overline{\tau}_2'.e \approx_C \overline{\sigma}_1'$. By step-wise robustness, $\exists \tau_1, \tau_2$ with $\tau_1(0) = \sigma_1$ and $\tau_2(0) = \sigma_2$, τ_1 and τ_2 preserve C-indistinguishability and $\tau_1.e \approx_I \overline{\sigma}_1'$ and $\tau_2.e \approx_I \overline{\tau}_2'.e$. As σ_1 and σ_2 are observationally indistinguishable, all corresponding reachable states must be, too. Thus $\tau_1.e$ is observationally equivalent to $\tau_2.e$. By induction hypothesis, $\overline{\sigma}_1'$ is observationally equivalent to $\overline{\tau}_2'.e$. Thus for all traces generated by $\overline{\sigma}_1'$ there exists a trace generated by $\overline{\tau}_2'.e$ that is equivalent. Thus, there exists $\overline{\tau}_2''$ starting from $\overline{\tau}_2'.e$ such that $\overline{\tau}_1' / \approx_C = \overline{\tau}_2' / \approx_C$. Concatenation concludes.

Now assume $\overline{\sigma}_1$ can declassify. Then by step-wise robustness exists τ_1 such that $\tau_1(0) = \sigma_1$, τ_1 preserves indistinguishability and $\tau_1.e$ can declassify. Further, there exists σ_1' such that $\tau_1.e \mapsto \sigma_1'$, $\overline{\sigma}_1' \approx_I \sigma_1'$, $\tau_1.e_{LL} = \sigma_{1LL}'$ and $\overline{\sigma}_{1LL} = \overline{\sigma}_{1LL}'$. As $\overline{\sigma}_1 \approx_C \overline{\sigma}_2$, $\overline{\sigma}_2$ can reach a declassification through a trace $\overline{\tau}_2$ preserving indistinguishability on the way, where we let $\overline{\sigma}_2' = \overline{\tau}_2.e$. As before, this can be matched by τ_2 with $\tau_2(0) = \sigma_2$, where we let $\sigma_2' = \tau_2.e$.

Now we have $\overline{\sigma}_1' \approx_I \sigma_1' \approx_C \sigma_2' \approx_I \overline{\sigma}_2'$, and furthermore $\overline{\sigma}_{1LL}' = \overline{\sigma}_{2LL}'$ as result of $\approx_C$ followed by declassification. As $\sigma_1' \approx_C \sigma_2'$ and $\overline{\sigma}_{1/2LH}' = \sigma_{1/2LH}'$, it follows that $\overline{\sigma}_1' \approx_C \overline{\sigma}_2'$. We conclude as in the non-declassifying case.

We can now prove the theorem.

Proof (Of theorem 5). Given two states σ_1 and σ_2 that are observationally equivalent under attack A_1, we are gonna show that these states are also observationally equivalent under attack A_2. Let $\tau^1 \in \mathcal{T}(\overline{S} \cup A_2, \sigma_1)$. Divide τ^1 into segments (states connected by non-attack transitions) connected by attack transitions: $\tau^1 = \tau_1^1 \mapsto_{A_2} \tau_2^1 \mapsto_{A_2} \dots$. We will show that there exists τ_i^2 for each τ_i^1 such that $\tau_i^1/\approx_C = \tau_i^2/\approx_C$ and τ_1^2 starts with σ_2, by induction on the number of segments.

In the base case, there is just one segment, i.e., $\tau^1 = \tau_1^1$. As there are no A_2 transitions, we have that $\tau^1 \in \mathcal{T}(\overline{S} \cup A_1, \sigma_1)$. Since σ_1 and σ_2 are observationally equivalent, there exists $\tau^2 \in \mathcal{T}(\overline{S} \cup A_1, \sigma_2)$ such that $\tau^1/\approx_C = \tau^2/\approx_C$. We can construct τ' a prefix of τ^2 such that $\tau^1/\approx_C = \tau'/\approx_C$ and τ' does not contain A_1 transitions: Assume there are such transitions. Then let τ' be the prefix up to and excluding the first A_1 transition. As it is a prefix of τ^2, and τ^2 is equivalent to τ^1, there must be a prefix of τ^1 that is equivalent to τ'. By the closure rules on attacks, either $\tau' = \tau^1$, or all states up to the end of τ^1 are equivalent to the end of τ'. Thus, $\tau'/\approx_C = \tau^1/\approx_C$. The trace τ' does not contain A_1 transitions. It is thus a trace of $\mathcal{T}(\overline{\mathcal{S}} \cup A_2, \sigma_2)$.

Now assume we have segments $\tau_1^{1/2}, \dots, \tau_i^{1/2}$ such that $\tau_j^1/\approx_C = \tau_j^2/\approx_C$, $\tau_j^{1/2} \mapsto_{A_2} \tau_{j+1}^{1/2}$, $\tau_i^1.e \approx_I \sigma_i^1 \approx_C \sigma_i^2 \approx_I \tau_i^2.e$, where σ_i^1 is observationally equivalent to σ_i^2 in $\overline{\mathcal{S}} \cup A_1$, and $\tau_i^1 \mapsto_{A_2} \tau_{i+1}^1$. We first note that, as $\tau_i^1/\approx_C = \tau_i^2/\approx_C$, if there is no attack transition from $\tau_i^2.e$ directly, then there exists an extension that preserves indistinguishability and ends in such a state. Thus, w.l.o.g., $\exists \overline{\sigma}_{i+1}^2.\tau_i^2.e \mapsto_{A_2} \overline{\sigma}_{i+1}^2$. Let $\overline{\sigma}_{i+1}^1 = \tau_{i+1}^1(0)$. We have that $\tau_i^1.e \approx_I \overline{\sigma}_{i+1}^1 \approx_C \overline{\sigma}_{i+1}^2 \approx_I \tau_i^2.e$ as an attack transition connects them. Thus $\sigma_i^{1/2} \approx_I \overline{\sigma}_{i+1}^{1/2}$. By Lemma 11, $\overline{\sigma}_{i+1}^1$ and $\overline{\sigma}_{i+1}^2$ are observationally equivalent in $\overline{\mathcal{S}} \cup A_1$, and $\tau_{i+1}^1 \in \mathcal{T}(\overline{\mathcal{S}} \cup A_1, \overline{\sigma}_{i+1}^1)$. Analogous to the base case, there is a $\mapsto_{A_1}$-free τ_{i+1}^2 with $\tau_{i+1}^2(0) = \overline{\sigma}_{i+1}^2$ such that $\tau_{i+1}^1/\approx_C = \tau_{i+1}^2/\approx_C$. Thus $\tau_{i+1}^2 \in \mathcal{T}(\overline{\mathcal{S}} \cup A_2, \overline{\sigma}_{i+1}^2)$. The concatenation of the segments concludes the proof.

Theorem 6 (Composition is step-wise robust). *Given the above assumptions, the composition $\mathcal{S}$ is step-wise robust.*

Proof. We have to show validity of the composed system, and the core properties of step-wise robustness.

We first note that `eval` itself is noninterfering, non-declassifying, and by typing, *pc*s increase. We enforce a single-entry single-exit regime for `eval`. It is thus easy to show that the validity requirements hold for systems describing simple `eval` executions. Then composed-system validity follows trivially by the single-entry single-exit nature, deterministic and noninterfering behavior of `eval`.

For the core properties, we note that property 4 does not apply to `eval`, and thus immediately carries over from the constituent systems. For all other properties, whenever $\sigma_1.l$ and $\sigma_2.l$ belong to one language mode, the properties either carry over directly (embedded mode) or are follow immediately from the constituent and the validity of simple `eval` executions. We will detail the arguments for $\sigma_1.l$ and $\sigma_2.l$ not denoting the same language mode on the example of the first core properties.

We have to show that $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_I \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \implies \exists \tau_2'.\forall i < len(\tau_2').\tau_2'(i) \approx_I \sigma_2 \wedge \tau_2'.e \approx_I \sigma_1'$. Note that as σ_1 and σ_2 denote different language states, $l_1 = \sigma_1.l \neq \sigma_2.l = l_2$. Then $\exists k_1, k_2.E^{k_1}(l_1) = E^{k_2}(l_2)$. If $l_1 = E^{k_2}(l_2)$, then by validity l_2 can complete the `eval` and reach l_1, from which a step is possible. If otherwise $E^{k_1>0}(l_1) = E^{k_2}(l_2)$, then the step of l_1 can be matched by a zero-step of l_2.