Security Completeness: Towards Noninterference in Composed Languages

Andreas Gampe
The University of Texas at San Antonio
agampe@cs.utsa.edu

Jeffery von Ronne
The University of Texas at San Antonio
vonronne@cs.utsa.edu

Abstract

Ensuring that software protects its users' privacy has become an increasingly pressing challenge. Requiring software to be certified with a secure type system is one enforcement mechanism. Protecting privacy with type systems, however, has only been studied for programs written entirely in a single language, whereas software is frequently implemented using multiple languages specialized for different tasks.

This paper presents an approach that facilitates reasoning over composed languages. It outlines sufficient requirements for the component languages to lift privacy guarantees of the component languages to well-typed composed programs, significantly lowering the burden necessary to certify that such composite programs safe. The approach relies on computability and security-level separability. This paper defines completeness with respect to secure computations and formally establishes conditions sufficient for a security-typed language to be complete. We demonstrate the applicability of the results with a case study of three seminal security-typed languages.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; Correctness proofs

General Terms Languages, Security, Verification

Keywords Security; noninterference; type systems; composition

1. Introduction

In contemporary software engineering practice, it is becoming more common to use several languages while implementing large systems. This ranges from using separate languages in different files (e.g., configuration files in XML or libraries coded in different languages) to using code in one language being directly embedded in the code of another language. As an example, one commonly embedded language is SQL, which is used to declaratively retrieve data. Such languages are common even in the mobile app space: both Android and iOS expose bindings for SQL systems (e.g., SQLite) to applications written primarily in Java and Objective C, respectively.

At the same time, privacy is becoming an increasingly important property of software. Mobile platforms like smartphones and tablets, for example, are built around "app stores", where users can download new applications. These applications can gain access to private information that is stored on the device, like contacts, phone logs, location information, etc. In the mobile space, major vendors have chosen two ways to help users: either, a simple form of static permissions, as used on the Android platform, or a review of every application, as in the case of iOS. Neither approach is very satisfying: several cases of apps that violate the privacy of the owner's device have become known recently for both platforms. A formal and sound mechanism that can be applied to formally verify such software against privacy policies would greatly improve this situation. Type systems enforcing noninterference can provide such a mechanism.

Noninterference [5] ensures that any compliant program cannot leak private information to public channels. Type systems are a common method to guarantee noninterference, e.g., [1, 6, 12, 16, 17], for an overview see [15], and the approach has been extended to cover entire distributed systems [9]. In all of this work, however, exactly one language is treated.

The question then is: How can we (statically) guarantee the safety of programs that are composed from elements in different languages such that a client can check a program for compliance with security policies? The solution we propose is to compose security-typed languages into *composed languages*, such that well-typed programs in a composed language can be guaranteed to comply with noninterference. Focusing on the more general end of the language composition spectrum, we will consider the case where a composed language is created by extending a *host language* so that

PLAS'13, June 20, 2013, Seattle, WA, USA.

code from an *embedded language* can replace certain constructs in otherwise valid host language programs.

This paper describes an approach (in Section 3) where, under certain assumptions, it is 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. Our initial conjecture is that all non-trivial security type systems will satisfy this requirement.

Suppose one has a noninterferent function $f(x, y)$ that produces a public output from public input x and secret input y. To obtain a typeable version f' of f, one can first define a function g to be the same function as f, but type both of g's inputs as public, and its output also as public. Intuitively, assuming that the underlying language is Turing complete, and that any function that involves only a single security level is typeable, g can be expressed and typed. Now, one can define $f'(x, y) = g(x, c)$, for some suitable constant c, which is well-typed since, usually, constants are public. A typical noninterference theorem guarantees that $f(x, y_1) = f(x, y_2)$, for all y_1, y_2, which ensures that $f = f'$.

This basic argument may seem trivial; actually showing that it holds for classes of languages providing operations over certain classes of data types (rather than a specific language with integer data) brings up several issues. Our basic approach for establishing this property is first described in Section 4 and formalized in Section 5. We then, in Sections 6 and 7, show how more complex data types and references can be supported, which adds additional requirements on the host language that would generally be expected to be satisfied by languages that support such entities. Note that the current work does not support extending the types of inputs and outputs to functions. Finally, in Section 8, we show that the requirements placed on host languages are reasonable by showing how three languages, the system of [16], FlowML [13], and work in [1] satisfy the requirements.

The contributions of this paper include

- identifying a general framework for leveraging the proofs of the noninterference of security-typed languages to establish the noninterference of their composition,
- showing how arbitrary noninterferent functions can be computed in certain classes of security-typed languages, and
- showing that the classes of languages are reasonable

2. Background

2.1 Language Definition

We assume languages provide values, programs and states, and are associated with a semantics such that a program and a state get reduced to a value and a state. We only consider deterministic languages here, that is, the semantics can be considered a (partial) function. We use p to range over programs, and f, g over functions represented by such programs.

Type systems assign types to values, programs and states in a judgment. If a type system makes a judgment, a certain property is guaranteed. For example, traditionally type systems enforce that programs cannot go wrong, that is, get stuck or end with an error.

Most security-typed languages can be seen as annotated versions of a ground typed language. These annotations may be on programs, values and states, as well as on types. The annotations are used to maintain local security invariants that allow judgments to guarantee security properties for the whole program.

2.2 Security

Our work is in the realm of lattice-based security [2], with security information taken from a lattice of security "levels". The simplest non-trivial lattice is $\{L, H\}$ with the obvious ordering, but many more complicated lattices have been proposed. Flows can be formulated by the partial order of levels in the lattice. Informally, (1) computations that involve private=high information need to be classified private, (2) we cannot allow direct assignments of private information to public storage and (3) we cannot allow indirect leaks, that is, assignments to public storage in a private context. Examples for violations of these cases are

```
(1) d{H} * 3{L} => 15{L}
(2) var tmp{L} = salary{H}
(3) if age{H}>65 then
      tmp{L}=1 else tmp{L}=0 end
```

In the first case, one can deduce that the private input d must have been 5. In the second case, we *declassified* formerly private information and store it in a public field. In the third case, we gain one bit of information about the private age information by case analysis of the value of `tmp`.

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. Approach to Composition

Our ultimate goal is to *prove* safe the composition of practical languages. For example, we are interested in the embedding of SQL into JIF [12], which would allow us to write code like the following, executing a SQL statement and retrieving the result in the host language.

```
int{high} age = SELECT age FROM employees
```

```
WHERE name=$name;
```

In this fragment, the (confidential) age of the employee with the name given in the Java variable `name`, embedded into the SQL statement by `$name`, is retrieved from the database table employees.

We now assume that SQL is secure by use of a security-type system. For example, we could assign security types to each column in each table of the database to trace flows through queries. In the example, this might derive that the result of the query should better be confidential.

Embedding can be formalized by extending the host language with an evaluation construct, maybe named `eval`. We would add evaluation and typing rules that connect the host and embedded language. For example, `eval` could be resolved by computing its parameters, translating them to the embedded language, running the embedded program, and translating the result back to the host. For typing purposes, we need a translation of types between the languages the preserve security properties. This translation needs to be monotonic and round-trip non-decreasing to not leak information. Furthermore, indistinguishable input must be translated to indistinguishable output. With such rules in place, and assuming that the type systems for both JIF and SQL are secure, we would like to prove that the composed language is also secure.

3.1 Limitations of Proof Manipulation

A strawman approach to proving noninterference for well-typed composed programs is by defining a mechanical procedure for directly manipulating the proofs of noninterference of programs in the host and embedded languages. However, this seems to be tied closely to the format and details of the noninterference proof of the host language. For example, if that proof was done syntactically via a subject reduction theorem, as for example in [13], that theorem would need to be extended. Subject reduction is usually shown by some inductive argument, for example over the input typing. As such, we could extend the case analysis of the induction with a specific argument for `eval` that is derived from the typing constraints, and hope to get a well-formed proof for the composed language.

However, the picture is not that simple. While adding a case to subject reduction is simple and (mostly) independent from the other constructs, other cases may use their own nested inductive arguments, auxillary lemmas, or inversions. [13], for example, uses auxillary lemmas for weakening, projection, and substitution, with all lemmas ranging over *all* constructs in the language. Without a detailed knowledge of the proof and the statements necessary, it seems impossible to generically prove correctness by manipulating an (in details) unknown proof.

3.2 Computability

We instead try to reuse the existing proofs completely and without modifications, by focussing on computability in a three-step process:

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

Assume that the embedded language is Turing-computable, that is, every program computes a Turing-computable function. Furthermore assume that the host language is Turing-complete, that is, there is a program for every Turing-computable function. Then the host language is able to *functionally* simulate any embedded language program (fragment). Since the original program was noninterferent - the `eval` forces a typing, which guarantees noninterference - the simulation is also noninterferent. Note that this argument requires compatible notions of noninterference. For example, if the embedded-language fragment was termination-sensitive noninterferent, then it will be termination-sensitive or termination-insensitive noninterferent in the host language.

However, not all termination-insensitive noninterferent programs are termination-sensitive noninterferent. In that case a totally faithful representation is impossible. We sketch the following solution. We can change the semantics of the embedded program to be total by selecting an arbitrary indistinguishable result in the nonterminating cases, which allows representation in the host. The resulting program is an overapproximation, but ensures that the original semantics with nontermination is now termination-*insensitive* noninterferent.

It remains to show that there is a legal program, that is, a *typable* host language program that computes the simulation. If such a program exists *and* the host language allows us to substitute the `eval` with our typed simulation, we end up with a pure host-language program that is typed and computes the same function as the composed-language program. The host language security-type system now guarantees this program to be noninterferent, which means that the functionally equivalent composed-language version is noninterferent, too.

Note that step 3 is specific to each host language, but can be reused for all compositions of that language. In general, a proof of this step is similar to a standard substitution lemma. One has to show that the program remains typed, and retains its meaning.

In this paper, we focus on step 2. We try to answer this key question: if a function is computable (there is a, not necessarily typable, program) and noninterferent, is there

a security-typable program that computes the (exact) same function? This is formulated in the following definition.

Definition 1 (Security Completeness)**.** *A security-typed language $\mathcal{L}$ is security-complete if and only if for every (not necessarily typable) program p that computes a function f, where f is noninterfering with respect to security signature S, there exists a program p' that also computes f and is typable corresponding to S.*

Our working conjecture here is the following.

Conjecture 2. *All non-trivial security-typed languages are security-complete.*

Note that, while the conjecture might seem obviously valid, "the devil is in the details." This statement is rather strong. While it is easy to show that it holds for simple formalizations of noninterference with primitive input and output types (e.g., `int`), it actually does *not* hold for all computations in all non-trivial languages when we consider more complex datastructures - noninterference and indistinguishability do not always provide enough context for a typable program. It remains to be seen what requirements on $\mathcal{L}$, $\mathcal{T}$ and f are sufficient to prove a corresponding theorem.

3.3 Applicability

One might ask if embedding a less powerful language is useful at all, and thus if our approach is realistic in practice. For one, in practice many embedded languages are only powerful in certain domains (i.e., domain specific languages). Most embedded languages show their advantages in the conciseness and expressivity in just this limited domain. For example, SQL is a query language for databases and is (in its basic incarnation) not Turing-complete, but can describe a complex set of relational queries with a relatively small amount of code. The SQL semantics could be simulated precisely in a general-purpose language, albeit with simulation overhead. Thus, SQL can clarify the meaning and intention of some part of a program, improving that and only that part over a general-purpose implementation. Also, most host languages are general-purpose languages that are Turing-complete and thus as powerful as realistically possible.

Second, we would like to stress that any simulation overhead is not relevant in practice. The simulation is a tool for guaranteeing noninterference of the extended semantics of the composed language. Thus, the size of the simulation would not matter, since it would *not* be used in practice.

Another relevant question is whether our requirements are too strong. While we require noninterferent input programs, type systems are still important once one already knows that a function is noninterferent. For example, a typing can act as a certificate for a program, such that remote clients can check for actual noninterference. Also, our overarching goal is to formally establish the safety of a composed language from its components. Typing for a known noninterferent embedded program still needs to be liftable to the overall language, which our approach provides.

4. Approach

The section details our approach in the simplified case of output indistinguishability being output equivalence, as used in the previous section. This is for example the case if the output is just a single integer value that is assumed to be public, which is a common form to formalize noninterference (e.g., [13]). We will formalize this setting in the next section, and the following sections will detail generalizations to more complicated values. A subsection treats the differences between termination-sensitive and termination-insensitive noninterference.

4.1 Basic Approach

If a function f is computable, then there exists a program p that computes f, that is, the output of p agrees with f under the same inputs for the right meaning of inputs and output. Noninterference is a dependency problem, If a program is noninterferent, then the (low) result does not depend on high inputs. This means that, for any high inputs, the low output value will be the same. We are thus able to substitute arbitrary constants for those inputs when computing only low outputs. However, we need to prove the existence or wellformed-ness of said program. We approach this from a computability direction, where constant functions and function composition are guaranteed by primitive recursion.

We prefer a composition requirement over more direct manipulations because it abstracts the exact syntax and semantics of the language involved. Note that we do not need to inspect programs at all, as required by, for example, a slicing approach. Instead, we show the existence of some separate program that is typable and computes an equivalent function. This allows us a generalization that can accept, for example, both imperative and functional languages. Note that it is important to find an *equivalent* function: for our simulation argument, it is not enough to compute correctly up to indistinguishability.

The final step is showing typability of this intuitive construction. We observe that, trivially, every function is noninterferent if all inputs are considered public - noninterference resolves to determinism (or some similar notion in the case of nondeterministic languages). This generalizes to any single security level. It seems reasonable to require that a nontrivial security-type system should be able to type a program with a security typing assigning a single level to everything. This is our first requirement for a security-typed language.

Next, projection and constant functions should be typable at the respective levels. Under projection we understand here functions of multiple inputs that return one of those inputs. For example, projection $\pi_1(x, y) = x$ should be typable as $\ell_x \times \ell_y \mapsto \ell_x$ for any ℓ_x and ℓ_y. The constant function is noninterferent no matter the security typing, since the output is always the same and does not depend on the inputs. Our

$\mathrm{a} := (\mathrm{x} + \mathrm{y}) - (\mathrm{y} - \mathrm{x});$	$[\mathrm{x} : L,\ \mathrm{y} : H,\ \mathrm{a} : L]$
$\mathrm{t}_1^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_1^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_1^{\mathrm{o}} := \mathrm{t}_1^{\mathrm{x}};$	$[\mathrm{t}_1^{\mathrm{x}} : L,\ \mathrm{t}_1^{\mathrm{y}} : H,\ \mathrm{t}_1^{\mathrm{o}} : L]$ $\mathrm{t}_1^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}_2^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_2^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_2^{\mathrm{o}} := 0;$	$[\mathrm{t}_2^{\mathrm{x}} : L,\ \mathrm{t}_2^{\mathrm{y}} : H,\ \mathrm{t}_2^{\mathrm{o}} : L]$ $\mathrm{t}_2^{\mathrm{o}} = c_0(x, y)$
$\mathrm{t}_3^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}}; \mathrm{t}_3^{\mathrm{y}} := \mathrm{t}_2^{\mathrm{o}};$ $\mathrm{t}_3^{\mathrm{o}} := (\mathrm{t}_3^{\mathrm{x}} + \mathrm{t}_3^{\mathrm{y}}) - (\mathrm{t}_3^{\mathrm{y}} - \mathrm{t}_3^{\mathrm{x}});$	$[\mathrm{t}_3^{\mathrm{x}} : L,\ \mathrm{t}_3^{\mathrm{y}} : L,\ \mathrm{t}_3^{\mathrm{o}} : L]$ $\mathrm{t}_3^{\mathrm{o}} = p^L(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_2^{\mathrm{o}})$
$\mathrm{a} := \mathrm{t}_3^{\mathrm{o}};$	

Figure 1. Simple Program with Security Context

requirement is that it can be typed with any result level, including public.

Last, we require that composition is typable if the components are typable and agree on input and output types. By construction, the input types of the composition agree with the security typing for the original program, and the output type with the output type of said program. Then a typing states the same (or extended) noninterference property that we intended for the original program, and the construction guarantees functional equivalence.

A demonstration is shown in Figure 1 for program $p = \mathrm{a} := (\mathrm{x} + \mathrm{y}) - (\mathrm{y} - \mathrm{x})$ computing $f(x, y) = 2x$. The result is shown below the double lines. The first block projects x, the second block is the constant 0 function, the third block represents the original program in a low-typable version, and the fourth block is the final result. The overhead comes from the necessity to rename variables when composing in imperative languages to prevent side effects. The original program is not typable under the signature that assumes x low and y high, because the assignment cannot be typed. The new construction, however can be typed and computes essentially the same function.

4.2 Termination Sensitivity

The development in the previous subsection only holds if we consider termination-sensitive noninterference. In that case, two runs on indistinguishable inputs have to agree on their termination behaviour, that is, the first run terminates if and only if the second run terminates. Termination-insensitive noninterference, on the other hand, only makes a statement over two terminating runs, considering that an attacker might not be able to observe (non-)termination, or at most gain one bit of knowledge from it. In that case, the approach outlined in the previous subsection cannot be guaranteed to simulate correctly only up to termination, because termination may depend on the high input, and thus the choice of constants.

Our solution imposes further requirements that allow the application of a standard technique in computability: dovetailing (interleaving computations). If we require the set of values valid for the high inputs to be recursively enumerable, we can *test* the function on all possible inputs. For this test to succeed, we have to be able to simulate the function in a stepwise manner, e.g., as in a small-step semantics. We will interleave the simulations of the different input values, such that if there is at least one value that forces termination, we will find that case. As an example, assume that $f_k(x)$ denotes a computation of $f(x : \mathbb{N})$ for k steps. Then an interleaving could be $f_1(0)$, $f_2(0)$, $f_1(1)$, $f_3(0)$, $f_2(1)$, $f_1(2)$, ... If for any x, $f(x) = v$ is defined, there is a k such that $f_k(x) = v$, and the interleaving contains this computation.

Such an interleaving will terminate if there is at least one terminating high value. To complete correctness with respect to the original, we compute the original function in a high setting in sequence. This will ensure that the simulation does not terminate when it should not.

We demonstrate this approach in Figure 2. The first three blocks iteratively compute the values of the program for increasing values of y, where sim simulates the given program for t^{n} steps and assigns $\mathrm{t}^{\mathrm{s}} = 0$ if the program finished. If the loop terminates, a result for some y will be in $\mathrm{t}_2^{\mathrm{o}}$. Note that the loop only involves L variables and can thus be typed as L. The following three blocks compute an H version of the program for termination correctness. The last block assigns the result of the low simulation as the overall result.

5. Formalization

In this section we formalize the approach and requirements outlined in the previous section. We start by introducing generic notation for the security-typed language and its security-type system, and formally defining our requirements in the first subsection. In the second subsection, we formally show how these requirements lead to our revised hypothesis.

5.1 Definitions & Requirements

We assume a security-typed language $\mathcal{L}$ and its security-type system $\mathcal{T}$ with associated lattice $\mathcal{S}$. The languages provides a set of values, ranged over by v, a set of programs, ranged over by p, and state or input, ranged over by μ. The language has associated semantics that reduces a program and state to a value and state. We denote the semantics by $(p, \mu) \rightsquigarrow_s (v, \mu')$. We define $\Downarrow$ to include nontermination, such that $(p, \mu) \Downarrow (v, \mu')$ if $(p, \mu) \rightsquigarrow_s (v, \mu')$, and $(p, \mu) \Downarrow (\bot, \bot)$, if there is no such (v, μ').

We connect the semantics to a functional interpretation through two predicates defined by the language. A program p computes function $f(x_1, \ldots, x_n)$, if for all μ such that $in_{p,f}(x_1, \ldots, x_n, \mu)$, and $(p, \mu) \Downarrow (v, \mu')$, we have $f(x_1, \ldots, x_n) = res_{p,f}(v, \mu')$, where in and res abstract how a language defines input and output in program p with respect to function f[1]. We denote this by $comp(p, f)$.

[1] For an example, recall how in the example of Figure 1 the inputs were bound to variables x and y and the output to variable a.

$\mathrm{a} := \mathrm{x}$; while $\mathrm{y} \neq 10$ do $\{\ \mathrm{a} := \mathrm{x}; \mathrm{y} := \mathrm{y} + 1;\ \}$	$[\mathrm{x} : L,\ \mathrm{y} : H,\ \mathrm{a} : L]$
$\mathrm{t}_1^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_1^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_1^{\mathrm{o}} := \mathrm{t}_1^{\mathrm{x}}$	$[\mathrm{t}_1^{\mathrm{x}} : L,\ \mathrm{t}_1^{\mathrm{y}} : H,\ \mathrm{t}_1^{\mathrm{o}} : L]$ $\mathrm{t}_1^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}^{\mathrm{i}} := 0; \mathrm{t}^{\mathrm{s}} := 1;$	$[\mathrm{t}^{\mathrm{i}} : L,\ \mathrm{t}^{\mathrm{s}} : L]$
while $\mathrm{t}^{\mathrm{s}} > 0$ do $\{$	
$\quad \mathrm{cantor}_1(\mathrm{t}^{\mathrm{i}}, \mathrm{t}_2^{\mathrm{y}});$	$[\mathrm{t}_2^{\mathrm{y}} : L,\ \mathrm{t}^{\mathrm{n}} : L]$
$\quad \mathrm{cantor}_2(\mathrm{t}^{\mathrm{i}}, \mathrm{t}^{\mathrm{n}});$	$\mathrm{t}^{\mathrm{i}} \mapsto \langle \mathrm{t}_2^{\mathrm{y}}, \mathrm{t}^{\mathrm{n}} \rangle$
$\quad \mathrm{t}_2^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}};$	$[\mathrm{t}_2^{\mathrm{x}} : L,\ \mathrm{t}_2^{\mathrm{o}} : L]$
$\quad \mathrm{sim}(p^L, \mathrm{t}_2^{\mathrm{x}}, \mathrm{t}_2^{\mathrm{y}}, \mathrm{t}^{\mathrm{n}}, \mathrm{t}_2^{\mathrm{o}}, \mathrm{t}^{\mathrm{s}});$	$p^L(\mathrm{t}_2^{\mathrm{x}}, \mathrm{t}_2^{\mathrm{y}}) \rightarrow^{(\mathrm{t}^{\mathrm{n}})} \mathrm{t}_2^{\mathrm{o}}$?
$\quad \mathrm{t}^{\mathrm{i}} := \mathrm{t}^{\mathrm{i}} + 1;$	
$\}$	
$\mathrm{t}_3^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_3^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_3^{\mathrm{o}} := \mathrm{t}_3^{\mathrm{x}};$	$[\mathrm{t}_3^{\mathrm{x}} : L,\ \mathrm{t}_3^{\mathrm{y}} : H,\ \mathrm{t}_3^{\mathrm{o}} : H]$ $\mathrm{t}_3^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}_4^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_4^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_4^{\mathrm{o}} := \mathrm{t}_4^{\mathrm{y}};$	$[\mathrm{t}_4^{\mathrm{x}} : L,\ \mathrm{t}_4^{\mathrm{y}} : H,\ \mathrm{t}_4^{\mathrm{o}} : H]$ $\mathrm{t}_4^{\mathrm{o}} = \pi_2(x, y)$
$\mathrm{t}_5^{\mathrm{x}} := \mathrm{t}_3^{\mathrm{o}}; \mathrm{t}_5^{\mathrm{y}} := \mathrm{t}_4^{\mathrm{o}};$ $\mathrm{t}_5^{\mathrm{o}} := \mathrm{t}_5^{\mathrm{x}};$ while $\mathrm{t}_5^{\mathrm{y}} \neq 10$ do $\{\ \mathrm{t}_5^{\mathrm{o}} := \mathrm{t}_5^{\mathrm{x}}; \mathrm{t}_5^{\mathrm{y}} := \mathrm{t}_5^{\mathrm{y}} + 1;\ \}$	$[\mathrm{t}_5^{\mathrm{x}} : H,\ \mathrm{t}_5^{\mathrm{y}} : H,\ \mathrm{t}_5^{\mathrm{o}} : H]$ $\mathrm{t}_5^{\mathrm{o}} = p^H(x, y)$
$\mathrm{a} := \mathrm{t}_2^{\mathrm{o}};$	

Figure 2. Example Pair-Result Program

The type system provides a set of types, ranged over by τ, and type judgements of the form $\Gamma \vdash_s p : \tau$, $\Gamma \vdash_s v : \tau$ and $\Gamma \vdash_s \mu$. Note that for our purposes, it is not necessary to explicitly include a program counter in the notation. Security completeness is about whole programs, at which point low side effects are permissible. We can extract a type level $l \in \mathcal{S}$ from a type τ through the function *top*. We use two predicates to connect a type judgment and function signature, similar to the semantic connection. A judgment $\Gamma \vdash_s p : \tau$ is typed according to f with signature $\tau_1 \times \cdots \times \tau_n \mapsto \tau_r$, if $in^t_{p,f}(\Gamma, \tau, \tau_1, \ldots, \tau_n)$ and $\tau_r = res^t_{p,f}(\Gamma, \tau)$. We denote this as $typed(p, f, \Gamma, \tau)$.

We define noninterference ni for a function f with respect to (security) signature $\tau_1 \times \cdots \times \tau_n \mapsto \tau$ in the following way:

$$ni(f, \tau_1 \times \cdots \times \tau_n \mapsto \tau) \iff \left(\begin{array}{l} \forall l \in \mathcal{S}.top(\tau) \sqsubseteq l \implies \\ \quad \forall x_1^1, x_1^2 : \tau_1, x_2^1, x_2^2 : \tau_2, \ldots . \\ \quad\quad (\forall i.top(\tau_i) \sqsubseteq l \implies x_i^1 = x_i^2) \implies \\ \quad\quad\quad f(x_1^1, \ldots) = f(x_1^2, \ldots) \end{array} \right)$$

A program p is noninterferent with respect to f and a signature $\tau_1 \times \cdots \times \tau_n \mapsto \tau$, if p computes f and f is noninterferent. The type system guarantees that a typable program is noninterferent with respect to all functions it computes and is typed accordingly[2]:

$$\Gamma \vdash_s p : \tau \wedge comp(p, f) \wedge \\ typed(p, f : \tau_1 \times \cdots \times \tau_n \mapsto \tau_r, \Gamma, \tau) \implies \\ ni(f, \tau_1 \times \cdots \times \tau_n \mapsto \tau_r)$$

Associated with the security-typed versions we expect ground-typed versions, denoted by a g subscript or by $\lfloor \bullet \rfloor$ (which can be seen as an erasure function removing all security annotations), that is, the security-typed language is based on a regular language and type system with regular soundness guarantees, that is, ground-typed programs do not go wrong.

We require the following manipulation functions for annotations.

Requirement 3 (Erasure & Lift)**.** *There exist an erasure function $\lfloor \bullet \rfloor$ and a lift function $\lceil \bullet \rceil^l$ such that*

$$\forall p, v, \mu \in \mathcal{L}, \Gamma, \tau \in \mathcal{T}.\lfloor p \rfloor \in \lfloor \mathcal{L} \rfloor, \lfloor v \rfloor \in \lfloor \mathcal{L} \rfloor, \ldots \\ \forall p_g, v_g, \mu_g \in \lfloor \mathcal{L} \rfloor, \Gamma_g, \tau_g \in \lfloor \mathcal{T} \rfloor, l \in \mathcal{S}.\lceil p_g \rceil^l \in \mathcal{L}, \lceil v_g \rceil^l \in \mathcal{L}, \ldots \\ \forall \tau_g, l \in \mathcal{S}.top(\lceil \tau_g \rceil^l) = l$$

We define a complete relabeling $[\bullet]^l = \lceil \lfloor \bullet \rfloor \rceil^l$. The identity $\tau = [\tau]^{top(\tau)}$ is required to hold.

We use the requirements on relabeling to form a partial order on types lifted from their security levels.

$$\tau_1 \sqsubseteq \tau_2 \iff \exists \tau, l_1 \sqsubseteq l2.\tau_1 = [\tau]^{l_1} \wedge \tau_2 = [\tau]^{l_2}$$

Security and ground languages are suitably related:

Requirement 4 (Security to Ground.)**.**

$$\forall p, v, \mu, \mu', \Gamma, \tau. \\ \Gamma \vdash_s p/v/\mu : \tau \implies \lfloor \Gamma \rfloor \vdash_g \lfloor p/v/\mu \rfloor : \lfloor \tau \rfloor \\ (p, \mu) \rightsquigarrow_s (v, \mu') \implies (\lfloor p \rfloor, \lfloor \mu \rfloor) \rightsquigarrow_g (\lfloor v \rfloor, \lfloor \mu' \rfloor)$$

Furthermore, it holds that

$$typed(p, f : \tau_1 \times \cdots \times \tau_n \mapsto \tau, \Gamma, \tau) \implies \\ typed(\lfloor p \rfloor, f : \lfloor \tau_1 \rfloor \times \cdots \times \lfloor \tau_n \rfloor \mapsto \lfloor \tau \rfloor, \lfloor \Gamma \rfloor, \lfloor \tau \rfloor)$$

and $comp(p, f) \implies comp(\lfloor p \rfloor, \lfloor f \rfloor)$.

[2] The notion that a program might compute multiple functions might be surprising. But computation here is defined with respect to what parts of the output state are of interest. For example, consider projection $\pi_1(x, y) = x$ in a WHILE language.

Furthermore, we can gain security-type system judgments from ground-type judgments for any security level from $\mathcal{S}$.

Requirement 5 (Single-Level.)**.**

$$\begin{aligned}&\forall l \in \mathcal{S}.\Gamma_g \vdash_g p_g/v_g/\mu_g : \tau_g \implies \\ &\qquad \lceil\Gamma_g\rceil^l \vdash_s \lceil p_g/v_g/\mu_g\rceil^l : \lceil\tau_g\rceil^l \\ &\forall l \in \mathcal{S}.(p_g,\mu_g) \rightsquigarrow_g (v_g,\mu_g') \implies \\ &\qquad (\lceil p_g\rceil^l,\lceil\mu_g\rceil^l) \rightsquigarrow_s (\lceil v_g\rceil^l,\lceil\mu_g'\rceil^l)\end{aligned}$$

Furthermore, it holds that

$$\begin{aligned}&typed(p_g, f_g : \tau_1 \times \cdots \times \tau_n \mapsto \tau, \Gamma_g, \tau_g) \implies \\ &\ typed(\lceil p_g\rceil^l, f_g : \lceil\tau_1\rceil^l \times \cdots \times \lceil\tau_n\rceil^l \mapsto \lceil\tau\rceil^l, \lceil\Gamma_g\rceil^l, \lceil\tau_g\rceil^l)\end{aligned}$$

and $comp(p_g, f_g) \implies comp(\lceil p_g\rceil^l, f_g)$.

We need programs that compute projection and constants.

Requirement 6 (Projection)**.** *Let $\pi_i^n(x_1,\ldots,x_n) = x_i$ be the i-th projection function of n inputs. Let $\pi_i^n : \tau_1 \times \cdots \times \tau_n \mapsto \tau_i$ be a signature of π_i^n. Then there exists a program p_i^n, a Γ and τ such that*

$$\begin{aligned}&\Gamma \vdash_s p_i^n : \tau \wedge comp(p_i^n, \pi_i^n) \wedge \\ &typed(p_i^n, \pi_i^n : \tau_1 \times \cdots \times \tau_n \mapsto \tau_i, \Gamma, \tau)\end{aligned}$$

Requirement 7 (Constant Function)**.** *Let $c_i^{x,l}(x_1,\ldots,x_n) = x$ be the x-constant function of n inputs. We have $\tau_1 \times \cdots \times \tau_n \mapsto \tau_r$ a signature of $c_i^{x,l}$, that is $x : \tau_r$ and $\tau_r = \lfloor\tau_i\rfloor^l$. Then there exists a program p^x, a Γ and τ such that*

$$\begin{aligned}&\Gamma \vdash_s p^x : \tau \wedge comp(p^x, c_x) \wedge \\ &typed(p^x, c_i^{x,l} : \tau_1 \times \cdots \times \tau_n \mapsto \tau_r, \Gamma, \tau)\end{aligned}$$

Note that this requirement is necessary. While it may seem that erasure and lifting of an arbitrary value of τ_i may fulfill the requirements, it does not guarantee the existence of a *program* that creates the value. This is important, since only programs need to be able to be composed. This restriction allows us to easily include imperative languages into the framework.

Finally, we want to compose typed programs. We decided to formulate a general composition requirement, instead of a special-cased one.

Requirement 8 (Composition)**.**

$$\begin{aligned}&\forall p_\bullet, p. \\ &\left(\begin{array}{l}\forall i.\Gamma_i \vdash_s p_i : \tau^i \wedge comp(p_i, f_i) \wedge \\ \quad typed(p_i, f_i : \tau_1 \times \cdots \times \tau_n \mapsto \tau_r^i, \Gamma_i, \tau^i)\end{array}\right) \wedge \\ &\left(\begin{array}{l}\Gamma \vdash p : \tau^p \wedge comp(p, f) \wedge \\ \quad typed(p, f : \tau_r'^1 \times \cdots \times \tau_r'^m \mapsto \tau_r, \Gamma, \tau^p)\end{array}\right) \wedge \\ &\forall i.\tau_r^i \sqsubseteq \tau_r'^i \\ &\implies \\ &\exists p_c, \Gamma_c, \tau_c.\ \begin{array}{l}\Gamma_c \vdash_s p_c : \tau_c \wedge comp(p_c, f \circ \overrightarrow{f_i}) \wedge \\ typed(p_c, f \circ \overrightarrow{f_i} : \tau_1 \times \cdots \times \tau_n \mapsto \tau_r, \Gamma_c, \tau_c)\end{array}\end{aligned}$$

where $(f \circ \overrightarrow{f_i})(x_1,\ldots,x_n) = f(f_1(x_1,\ldots,x_n),\ldots, f_m(x_1,\ldots,x_n))$.

5.2 Revised Theorem & Proof

Theorem 9 (Security-Typability Completeness)**.** *Assume a language $\mathcal{L}$ and corresponding ground language that fulfill the requirements in the previous subsection. Such language is security-complete.*

Proof. Assume p a program that computes noninterferent $f : \tau_1 \times \cdots \times \tau_n \mapsto \tau$. Since p is ground-typable, we have $p_g = \lfloor p\rfloor$ such that there is a ground typing $\Gamma_g \vdash_g p_g : \tau_g$, such that $typed(p_g, f : \lfloor\tau_1\rfloor \times \cdots \times \lfloor\tau_n\rfloor \mapsto \lfloor\tau\rfloor, \Gamma_g, \tau_g)$ by requirement 4. Let $l = top(\tau)$. Define $g_\bullet$ as

$$g_i = \begin{cases}\pi_i^n : \tau_1 \ldots \tau_n \mapsto \tau_i & \text{if } top(\tau_i) \sqsubseteq l \\ c_i^{x_i,l} : \tau_1 \ldots \tau_n \mapsto \lfloor\tau_i\rfloor^l & \text{else, with arbitrary } x_i : \lfloor\tau_i\rfloor^l\end{cases}$$

which exist by requirements 6 and 7. Noninterference of f with respect to signature $\tau_1 \times \cdots \times \tau_n \mapsto \tau$ and level l states that

$$\begin{aligned}&\forall x_1^1, x_1^2 : \tau_1, \ldots, x_n^1, x_n^2 : \tau_n. \\ &\quad (\forall i.top(\tau_i) \sqsubseteq l \implies x_i^1 = x_i^2) \implies \\ &\qquad\qquad f(x_1^1,\ldots) = f(x_1^2,\ldots)\end{aligned}$$

Take any set of inputs $x_\bullet$ for f. Let $y_\bullet$ be defined as $y_i = g_i(x_1,\ldots,x_n)$. Then $\forall i.top(\tau_i) \sqsubseteq l \implies x_i = y_i$ by construction. Now, by noninterference, we have

$$\begin{aligned}f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) = f(&g_1(x_1,\ldots,x_n),\ldots, \\ &g_n(x_1,\ldots,x_n))\end{aligned}$$

By requirements 6 and 7, there exist $p_\bullet$, $\Gamma_\bullet$ and $\tau_\bullet^f$ such that

$$\begin{aligned}&\forall i.\Gamma_i \vdash_s p_i : \tau_i^g \wedge comp(p_i, g_i) \wedge \\ &\quad typed(p_i, g_i : \tau_1 \ldots \tau_n \mapsto \tau^i, \Gamma_i, \tau_i^g).\end{aligned}$$

Furthermore, by construction we have $\forall i.\tau^i \sqsubseteq \lfloor\tau_i\rfloor^l$, by requirements 3 and 7.

By the Single-level requirement, we can lift the typing of p_g such that $\lceil\Gamma_g\rceil^l \vdash_s \lceil p_g\rceil^l : \lceil\tau_g\rceil^l$ and $comp(\lceil p_g\rceil^l, f)$ and $typed(\lceil p_g\rceil^l, f : \lfloor\tau_1\rfloor^l \times \cdots \times \lfloor\tau_n\rfloor^l \mapsto \lfloor\tau\rfloor^l, \lceil\Gamma_g\rceil^l, \lceil\tau_g\rceil^l)$. This allows us to use the Composition requirement, composing $p_\bullet$ into $\lceil p_g\rceil^l$, which is functionally equivalent to composing $g_\bullet$ into f. This results in a program p_c and typing $\Gamma_c \vdash_s p_c : \tau_c$ such that $comp(p_c, f \circ \overrightarrow{g_i})$ and $typed(p_c, f \circ \overrightarrow{g_i} : \tau_1 \times \cdots \times \tau_n \mapsto \lfloor\tau\rfloor^l, \Gamma_c, \tau_c)$. Previous deductions and identity requirement on relabeling permit us to simplify this to $comp(p_c, f)$ and $typed(p_c, f : \tau_1 \times \cdots \times \tau_n \mapsto \tau, \Gamma_c, \tau_c)$. Thus, the program p_c is typable with the required signature and computes f, which concludes the proof. □

5.3 Sufficient vs. Necessary Conditions

Our derivation in the previous subsections concludes that the requirements established are sufficient for a language to be security-complete. However the reverse is not generally true. The leeway that the definition of security completeness allows us, i.e., that *another* equivalent program exists that is typable, makes a reverse deduction impossible in general.

6. Datatypes

We can extend the formalization of the previous section to data types. For security, compound values imply the possibility of more complicated indistinguishability relations, e.g., different parts of a value may have different security levels and need to be treated differently. A statement of noninterference may then use this complex indistinguishability both for inputs and outputs. That is, noninterferent programs create outputs that agree on low parts if the low input parts are equivalent. A simple case for demonstration follows. Assume that the language in questions supports *pairs*. Let $f(x, y) = \langle x, y + 1\rangle$, where x and the first component of the output pair are public, and y and the second component of the output pair are confidential. A sample noninterference statement for this function is

$$\forall x, y_1, y_2, x'_1, x'_2, y'_1, y'_2.$$
$$f(x, y_1) = \langle x'_1, y'_1\rangle \wedge f(x, y_2) = \langle x'_2, y'_2\rangle \implies \langle x'_1, y'_1\rangle \sim \langle x'_2, y'_2\rangle \equiv x'_1 = x'_2.$$

Notice that the construction in the previous subsection required the whole output to be equivalent, whereas now only the public part is. Also, the confidential output may depend on the confidential input, as in the given example function.

6.1 Assumptions

We assume some structure of complex values. First, complex values can be described as algebraic, that is, are of the form $v = c\ v_1, \ldots, v_n$ for an n-ary constructor c. Note that we require values to be made up of sub-values. To ensure termination of our recomputation, we require all treated values to be finite. We require typing to structurally match values: if a value $v = c\ v_1, \ldots, v_n$ can be typed as τ under Γ, then there are types $\tau_1, \ldots, \tau_n$ such that $v_1, \ldots, v_n$ are typed under Γ, and for all $v'_1, \ldots, v'_n$ typable in that way, $c\ v'_1, \ldots, v'_n$ can be typed as τ. This is a standard consequence in rule-based type systems.

Furthermore, we need functions to decompose values to their components. To unify product and variant treatment, we assume a matching construct in the language. Formally, if a type system can type values $v_i = c_i\ v^i_1, \ldots, v^i_{n_i}$ with $c_i \neq c_j$ (for $i \neq j$) as τ, where v^i_j can be typed with τ^i_j, then there exist matching functions with the signature $\mathrm{match} : \tau \mapsto (\tau^1_1 \times \cdots \times \tau^1_{n_1} \mapsto \tau') \mapsto \ldots \mapsto \tau'$ for all τ', with the semantics that $\mathrm{match}(c_i\ v^i_1, \ldots, v^i_{n_i}, f_1, \ldots, f_m) = f_i(v^i_1, \ldots, v^i_{n_i})$. This is standard for pattern-matching languages, and can be simulated in languages without explicit pattern matching (e.g., by branching on tag values encoding variants). We will use the common syntax, that is, "$\lambda x \ldots$" for functions and "$\mathrm{match}\ x\ \mathrm{with} \ldots$" for matching. Note that this existential requirement is very weak: to compute with datatypes, one form or another of matching is required.

We assume that each type τ involved has one immediate security-level annotation, which we denote by $tp(\tau)$. Multiple immediate annotations can be handled by complex security lattices. For matching, we require match to be typable, if $tp(\tau) \sqsubseteq tp(\tau^i_j)$ for all i and j, $tp(\tau) \sqsubseteq tp(\tau')$, and all f_i are typable according to the signature of match. This might seem restrictive, but is powerful enough to capture all cases outside the limitations outlined in the following subsection.

We also need to make minimum requirements on what indistinguishability means for values of a type τ and observer level ϕ. Our single requirement is that if two values v_1 and v_2 of type τ are indistinguishable at level ϕ, and $tp(\tau) \sqsubseteq \phi$, then both values have the same root constructor, and all immediate sub-values are indistinguishable with respect to their corresponding types at level ϕ.

6.2 Limitations

It turns out that Conjecture 2 is not provable in the generalized context anymore. As an example, take a language with pairs which have three security annotations: one for each component and one to signal the security level of the identity of the pair. Now take a pair that has a public and a private component, and is itself private. This leads to the public component not being accessible by an attacker (cf. [1, 12, 13]). Indistinguishability might thus be defined as:

$$\langle x_1, y_1\rangle \sim \langle x_2, y_2\rangle \text{ at } \langle \phi_1, \phi_2\rangle^{\phi_3} \iff \phi_3 = H \vee \begin{pmatrix} (\phi_1 = L \implies x_1 = x_2) \wedge \\ (\phi_2 = L \implies y_1 = y_2) \end{pmatrix}$$

With this, the following computation is noninterferent:

$$\mathtt{if}\ \mathrm{h} > 0\ \mathtt{then}\ \langle 3, 5\rangle\ \mathtt{else}\ \langle 4, 5\rangle\ :\ H \mapsto \langle L, H\rangle^H$$

However, this computation has a dependency between high input h and the low output component. In our companion technical report [4] we formally show that FlowML, a practical non-trivial language, cannot type any program that computes this function. We thus limit the theorems to security types such that levels of sub-types are at least as high as those of enclosing types.

6.3 Approach

The intuition behind our approach is to split computations by output level, allowing a level-separated computation. The final result then needs to be composed from the parts. Separability is a known result for trace-based security. We re-use and extend it to complex datastructures.

In a language-based environment, directly separating by security level is complicated. Since levels are connected to types, which are connected to the structure of values, we instead separate structurally, which implies a level separation. E.g., with the example above, we will find a program that represents $f(x, y)$ as a composition of computations for each pair component:

$$f(x, y) = \mathrm{match}\ f^L(x, y)\ \mathrm{with}\ \langle x_t, y_t\rangle \Rightarrow \langle f^1(x, y), f^2(x, y)\rangle$$

where $f^1(x, y) = \pi_1(f(x, y))$ and $f^2(x, y) = \pi_2(f(x, y))$. Intuitively, the matching will compute a single (sub-)value

if $\mathrm{y} = 0$ then $\mathrm{a} := \langle \mathrm{x}, \mathrm{y} \rangle$ else $\mathrm{a} := \langle \mathrm{x}+0, \mathrm{y}+1 \rangle$;	$[\mathrm{x} : L,\ \mathrm{y} : H,\ \mathrm{a} : L \times H]$
$\mathrm{t}_1^{\mathrm{x}} := \mathrm{x}$; $\mathrm{t}_1^{\mathrm{y}} := \mathrm{y}$; $\mathrm{t}_1^{\mathrm{o}} := \mathrm{t}_1^{\mathrm{x}}$	$[\mathrm{t}_1^{\mathrm{x}} : L,\ \mathrm{t}_1^{\mathrm{y}} : H,\ \mathrm{t}_1^{\mathrm{o}} : L]$ $\mathrm{t}_1^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}_2^{\mathrm{x}} := \mathrm{x}$; $\mathrm{t}_2^{\mathrm{y}} := \mathrm{y}$; $\mathrm{t}_2^{\mathrm{o}} := 0$;	$[\mathrm{t}_2^{\mathrm{x}} : L,\ \mathrm{t}_2^{\mathrm{y}} : H,\ \mathrm{t}_2^{\mathrm{o}} : L]$ $\mathrm{t}_2^{\mathrm{o}} = c_0(x, y)$
$\mathrm{t}_3^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}}$; $\mathrm{t}_3^{\mathrm{y}} := \mathrm{t}_2^{\mathrm{o}}$; if $\mathrm{t}_3^{\mathrm{y}} = 0$ then $\mathrm{t}_3^{\mathrm{o}} := \langle \mathrm{t}_3^{\mathrm{x}}, \mathrm{t}_3^{\mathrm{y}} \rangle$ else $\mathrm{t}_3^{\mathrm{o}} := \langle \mathrm{t}_3^{\mathrm{x}}+0, \mathrm{t}_3^{\mathrm{y}}+1 \rangle$;	$[\mathrm{t}_3^{\mathrm{x}} : L,\ \mathrm{t}_3^{\mathrm{y}} : L,\ \mathrm{t}_3^{\mathrm{o}} : L \times L]$ $\mathrm{t}_3^{\mathrm{o}} = p^L(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_2^{\mathrm{o}})$
$\mathrm{t}_4^{\mathrm{o}} := \pi_1\ \mathrm{t}_3^{\mathrm{o}}$;	$[\mathrm{t}_4^{\mathrm{o}} : L]$
$\mathrm{t}_5^{\mathrm{x}} := \mathrm{x}$; $\mathrm{t}_5^{\mathrm{y}} := \mathrm{y}$; $\mathrm{t}_5^{\mathrm{o}} := \mathrm{t}_5^{\mathrm{y}}$;	$[\mathrm{t}_5^{\mathrm{x}} : L,\ \mathrm{t}_5^{\mathrm{y}} : H,\ \mathrm{t}_5^{\mathrm{o}} : H]$ $\mathrm{t}_5^{\mathrm{o}} = \pi_2(x, y)$
$\mathrm{t}_6^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}}$; $\mathrm{t}_6^{\mathrm{y}} := \mathrm{t}_5^{\mathrm{o}}$; if $\mathrm{t}_6^{\mathrm{y}} = 0$ then $\mathrm{t}_6^{\mathrm{o}} := \langle \mathrm{t}_6^{\mathrm{x}}, \mathrm{t}_6^{\mathrm{y}} \rangle$ else $\mathrm{t}_6^{\mathrm{o}} := \langle \mathrm{t}_6^{\mathrm{x}}+0, \mathrm{t}_6^{\mathrm{y}}+1 \rangle$;	$[\mathrm{t}_6^{\mathrm{x}} : H,\ \mathrm{t}_6^{\mathrm{y}} : H,\ \mathrm{t}_6^{\mathrm{o}} : H \times H]$ $\mathrm{t}_6^{\mathrm{o}} = p^H(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_5^{\mathrm{o}})$
$\mathrm{t}_7^{\mathrm{o}} := \pi_2\ \mathrm{t}_6^{\mathrm{o}}$;	$[\mathrm{t}_7^{\mathrm{o}} : H]$
$\mathrm{a} := \langle \mathrm{t}_6^{\mathrm{o}}, \mathrm{t}_7^{\mathrm{o}} \rangle$;	

Figure 3. Example Pair-Result Program

at the level of the immediate annotation of that type. Noninterference will enforce that at least the variant chosen is correctly computed at this level. The corresponding matched case will re-compute all sub-values, at their correct levels, and reconstruct the correct value in a typable fashion.

An example of this construction is given in Figure 3. The program p on top computes $f(x, y) = \langle x, y+1 \rangle$, but is not typable. In the transformed program, for brevity we use π to extract components of a pair. The first four blocks compute the low component, while the next three blocks compute the high component, and finally the pair is reconstituted. Note the conceptual similarity to [3]. They perform a similar process at runtime to enforce noninterference, compared to our approach of showing typability in the case when noninterference is given.

6.4 Nonrecursive Datatypes

For nonrecursive datatypes, a type τ can be matched statically to any value v it types. We will recompute a (sub-)value corresponding to the structure of its (sub-)type, ensuring typability along the way.

We can recursively generate a function for this whole computation. Note that underlined functions are meta-level functions defining a language-level construct - in a sense they are macros to construct the language-level computation. Assume that f is noninterferent with respect to signature $\tau^i \mapsto \tau$. Here, we assume τ^i is not complex to simplify the presentation. Also, let f^ϕ denote the function that results from single-level typing as outlined in the previous section. We start with the actual computation, modeled with $\underline{\text{match}}$:

$$\underline{\text{match}}(p, \tau) \ = \ \lambda x : \tau^i.\ \underline{\text{extract}}(p, \tau)\ \ f^{tp(\tau')}(x)$$
// if τ' is not a datatype
$$\underline{\text{match}}(p, \tau) \ = \ \lambda x : \tau^i.$$
$$\text{match}\ \ (\underline{\text{extract}}(p, \tau)\ \ f^{tp(\tau')}(x))\ \ \text{with}$$
$$\vdots$$
$$c_i\ t_1^i, \ldots, t_{n_i}^i \ \Rightarrow\ c_i\ (\underline{\text{match}}(p{+}{+}(i, 1), \tau)\ x), \ldots, (\underline{\text{match}}(p{+}{+}(i, n_i), \tau)\ x)$$
// else

Here, p is a path the a sub-value/sub-type, which is encoded by a list of pairs for the choice of constructor and immediate sub-value. We denote the type in τ relative to path p by τ'. Note that the concept of paths is purely meta-level, since nonrecursive types can be fully statically described - we only need it to describe the computation recursively.

The key point of $\underline{\text{match}}$ is the recomputation of f at the level of the currently inspected sub-value denoted by p. To avoid inspection of f, we do a full recomputation, which then requires to extract the sub-value in question - this is the job of $\underline{\text{extract}}$. With the restrictions on τ and indistinguishability, it follows that this recomputation is correct up to the choice of constructor, but not necessarily the sub-values $t_1^i, \ldots, t_{n_i}^i$. We thus recompute the sub-values recursively by extending the path for $\underline{\text{match}}$.

Extraction itself does not need to recompute at each step. To be typable as needed, $\underline{\text{extract}}$ refers to a default value v_{def} for type τ' when the given path does not lead to such a case (we use "default" to stand for the finite number of other cases).

$$\underline{\text{extract}}((), \tau) \ = \ \lambda x : \tau.\ \ x$$
$$\underline{\text{extract}}((i, j) :: p, \tau) \ = \ \lambda x : \tau.\ \ \text{match}\ \ x\ \ \text{with}$$
$$c_i\ t_1^i, \ldots, t_{n_i}^i \ \Rightarrow\ \underline{\text{extract}}(p, \tau_j^i)\ \ t_j^i$$
$$\text{default} \ \Rightarrow\ v_{def}$$

We can now formalize our conjecture for nonrecursive datatypes.

Theorem 10 (Nonrecursive Datatypes)**.** *Assume a language $\mathcal{L}$ and corresponding ground language that fulfill all the requirements of the previous section, as well as the requirements in this section. This language is security-complete with the mentioned restrictions.*

The proof proceeds by induction over the structure of types and paths and can be found in [4].

6.5 Recursive Datatypes

The approach of the previous section can be extended to recursive datatypes, but for space reasons we only sketch our solution. The key difference is that now the path in a value needs to be handled dynamically. We can model the path with a list of integers, which is a recursive type and thus allowed by the language in question. We use μ types to guide the construction of the corresponding code. μ types allow a binding construct $\mu\, x.\tau$, where x may appear in τ.

For each binder a recomputation function is generated that assumes that the current path leads to a place in the value that corresponds to the binder. Furthermore, to minimize typing requirements, e.g., not require polymorphism, we create extraction functions for each start and end binder, e.g., $\mathrm{extract}_{x \mapsto y}$ assumes a value corresponding to binder x, and a path that will lead to binder y.

With this setup we can state a theorem corresponding to Theorem 10. The details of the construction, exact statement and proof are available in [4].

7. References

References introduce additional constructs that need to be handled. This complicates matters and requires further restrictions on languages that our technique can support. For one, most languages with references only allow limited interactions with references. Allocation of new locations can usually not be influenced directly on the language level. This makes exact recomputation impossible. Our technique can thus only simulate correctly up to renaming of heap locations. This also means that computations exhibiting identity behaviour cannot be simulated correctly. We see this as a minor disadvantage. We want to use the simulation to replace an embedded program that is cleanly separated from the host. It seems reasonable to require that any objects returned from the embedded program are independent of the inputs. This is usually the case when the embedded program cannot "call back" into the host program.

Any recomputation in parts will repeatedly invoke the original computation at certain levels. This may create several temporary locations polluting the heap, which, of course, do not appear in the original computation. Our technique is thus only correct up to locations reachable from the result of the function.

Also, computation with references allows side effects. In this case, a side effect may change the input values. We can contain side effects if we can create temporary "clones" of the relevant inputs and use those for computations. This means that the language needs to guarantee that two suitably related inputs, e.g., clones, compute suitably related outputs. We formalize this as the shape of the part of the heap reachable from the inputs, which corresponds to the first restriction. This, however, forbids any reflective language constructs.

7.1 Heaps

We formalize state through the concept of heaps. Heaps, denoted by $\mathcal{H}$, are mappings of locations, denoted by ℓ, to values. Values are extended to include locations. nil is a special value that is not mapped by any heap. Typings may contain a heap typing that assigns types to locations. Reference types are composed of the type of values that can be stored at the location, as well as the security level of location value itself.

7.2 Reachability, Equivalence & Indistinguishability

As outlined above, we restrict our attention to languages that restrict computation to reachable values. We formalize reachability as a set parameterized over a heap and starting location in said heap. The reachable set $\mathcal{R}$ of $\mathcal{H}$ and ℓ is the smallest set closed under the following rules: (1) $\ell \in \mathcal{R}$ and (2) if $\ell \in \mathcal{R}$ and $\mathcal{H}(\ell) = o$ a value of a type τ, then for all location-typed sub-values o' of o we have $o' \in \mathcal{R}$. For our treatment, we require $\mathcal{R}$ to be finite.

We define two pairs of heap and start locations to be equivalent if there exists a bijection ρ between the locations of the reachable sets such that non-reference values in related locations are identical, and reference values are related by the bijection. We denote equivalence by $\equiv$.

If two locations are indistinguishable with respect to bijection ρ and level ϕ and their level is at most ϕ, then they are in relation with respect to ρ. This is standard. Now, along the lines of datatypes we require that if two references to values of type τ are indistinguishable with respect to ϕ and ρ, and the security annotation is at most ϕ, then the potential variants of both objects are identical. We require that indistinguishability for heaps is with respect to its reachable part. That is $(\mathcal{H}_\infty, o_1)$ is indistinguishable to $(\mathcal{H}_\in, o_2)$ at level ϕ if there exists a bijection ρ between the reachable sets and o_1 and o_2 are indistinguishable with respect to ρ.

7.3 Computation

We define as a computation a function from a heap $\mathcal{H}$ and location in the heap, to a result heap $\mathcal{H}'$ and result object. A program computes f if reduction of the main fragment with variables bound to the input object and given heap results in the result heap and result object. As mentioned above, we assume correct executions. Thus we will leave typing constraints implicit here. All definitions are predicated on heaps and input objects correct with respect to f, that is $\mathcal{H}$ etc. range only over valid input states for f.

We formalize our requirements on the treated programs in the following way. A function f is ok, if:

- $\forall \mathcal{H}, \mathcal{H}', o, o'. f(\mathcal{H}, o) = (\mathcal{H}', o') \implies \mathcal{R}_{\mathcal{H},o} \cap \mathcal{R}_{\mathcal{H}',o} = \emptyset$
- $\forall \mathcal{H}_1, o_1, \mathcal{H}_2, o_2. (\mathcal{H}_1, o_1) \equiv (\mathcal{H}_2, o_2) \implies f(\mathcal{H}_1, o_1) \equiv f(\mathcal{H}_2, o_2)$

A simulation g is correct for f, if $\forall \mathcal{H}, o. f(\mathcal{H}, o) \equiv g(\mathcal{H}, o)$.

7.4 Approach

Treatment of references generalizes the approach of recursive data types. In this case, a heap walking approach is used. At each location in the heap, inputs need to be duplicated. Note that this is typable due to the single level necessary for the current computation. Non-location values can be immediately computed when a location is reached. Otherwise, references are resolved recursively, extending the current path from the starting location. Since the reachable part of the heap is required to be finite, the workload is finite.

A complication arises from the potential for cyclic paths in the heap. The process is complicated by the fact that the original result will be recomputed in every step. Our solution is to store a list of objects seen when extracting the current object from the result along the current path. This list can be typed with a single level, since the result object and heap are typed at a single level. A "back edge" is detected when the current extracted object is present in the list. To also detect "cross edges", we must also walk all completed paths again. This can be implemented through a deterministic processing order of fields in objects. To break the cycle, we return the object constructed earlier. This is the main reason why the treatment of datatypes (value is created late) differs from references (value is created early). Retrieval can be encoded with single-level integer lists describing paths to stored objects. If we update locations early, that is, a child adds itself to its parent, and have access to the root, we can use the path to retrieve the recomputed object in a typable way.

For space reasons, we omit a formalization of the construction, which can be found in the technical report, and only state the resulting theorem.

Theorem 11 (References)**.** *Assume a language $\mathcal{L}$ and corresponding ground language that fulfill all the requirements of the previous sections, as well as the requirements in this section. Such a language is security-complete with the mentioned restrictions.*

8. Example Languages

This section briefly describes three case studies which demonstrate that our formalization and requirements permit such different paradigms as imperative, functional and object-oriented languages. A more detailed analysis can be found in the technical report.

8.1 Volpano, Smith & Irvine

VSI [16] is based on a simple WHILE language based on integers. It fits the development in Section 5. Erasure and lifting functions are straightforward for VSI, since only types are annotated. Requirement 5 follows from the polymorphic setup of the type rules and can be formally proved by induction on the ground typing. Variable assignment represents the projection function of requirement 6, typable by the assignment rule. Assignment of an integer literal represents the constant function of requirement 7, and can be typed at the output variable level.

The only complicated requirement is composition. In a WHILE language, composition is basically sequencing the components, with possible assignments to connect the components. Composition actually necessitates renaming of variables and connecting assignments to allow typing, but satisfies the requirements.

8.2 FlowML

FlowML [13] is based on a core functional ML fragment including references, pairs, sums and exceptions. For simplicity of the functional interpretation we do not treat exceptions and references here. FlowML fits the development in Section 6.4. Lifting, erasure and single-level typing follow from the polymorphic setup of the rules. Projection is provided by a simple variable, while constants can be freely formed. Composition is provided by variable substitution, which may be combined with renaming and weakening to fulfill the requirements. Extraction for pairs is provided by typed projection, and a `case` construct allows to distinguish variants. However, basic noninterference cannot be lifted to abstractions, so that we cannot support arrow types (c.f. [6]).

8.3 Banerjee & Naumann

It is easy to extend the work in Section 7 to a class-based setting. We study the work in [1]. Additional treatment over pure references is necessary for encapsulation, which we solve by making all fields accessible through accessor methods. This does not change the computation. Single-level requirements can be ensured by complete copies of all classes and setting all annotations at the requested level. Projection, constant functions, and composition can be handled as in VSI. Furthermore, we need a matching construct to match objects to their respective classes. This can be realized with `instanceof` and dynamic casts provided by the language. Note that these constructs have the same security level as their inputs, so that they are typable as required.

9. Related Work

Completeness To the best of our knowledge, this is the first work to formally investigate completeness in the context of security-typed languages. Kahrs [7, 8] studied completeness for basic type systems, where the question is if all computable functions that are "well-going" can be typed. Kahrs formalized languages and type systems as transition systems and used the product and reachability to define soundness. In comparison, our goal is to permit easy adoption of existing languages, which usually have rule-based designs for both semantics and type systems. As demonstrated, the requirements usually easily follow from the modular nature of judgments, and can be derived for both imperative and functional security-typed 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 certified secure. For an overview we refer to [15]. Our work is orthogonal to such efforts. We show that, under certain constraints formulated

in our requirements, there are always programs that compute a given noninterferent function.

Language Composition To the best of our knowledge, this is the first work to consider composing two different security-typed languages. Composition has been studied in the security field for trace-based systems (e.g., [11]), in the setting of process algebras ([14] for an overview), and security-typed languages ([10] for an overview). In all these, multiple programs (or fragments) of *one* formal system are composed, and the question is if all programs are secure, is the composition secure. In contrast, our fragments are derived from different security-typed languages, exhibiting different semantics that are interfaced through an additional host statement.

Approach Our approach of level separation to establish security completeness is similar to the runtime effort in [3]. Devriese and Piessens enhance the semantics of a language to allow for level-separated computation to ensure noninterference at runtime, potentially changing the meaning of a program. We achieve typing through the use of single-level typing and also separating computations, without changing semantics or meaning of a program. Furthermore, we study the approach also for complex datatypes, nondeterminism and termination-insensitive noninterference, whereas Devriese and Piessens only speculate on the first and avoid the second and third.

10. Conclusion & Future Work

In this paper we have shown an approach, under certain restrictions, to show security of a language composition if the composed components themselves are secure. The approach is based on a simulation of component behavior, and accompanying proof that the noninterference of the component computation guarantees that there exists a typable program for the simulation, thus deducing noninterference for the composed language from noninterference for the host language.

This result allows us to develop separate type systems for languages, and lift the results to compositions. 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.

Acknowledgments We would like to thank our shepherd Nikhil Swamy and the anonymous referees for their helpful comments improving the presentation of this paper.

This work was supported by the National Science Foundation under grants CCF-0846010 and CNS-0964710.

References

[1] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, Mar. 2005.

[2] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976. ISSN 0001-0782.

[3] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[4] A. Gampe and J. Von Ronne. Towards noninterference in composed languages. Technical Report CS-TR-2012-014, Department of Computer Science, The University of Texas at San Antonio, 2012.

[5] J. A. Goguen and J. Meseguer. *Security Policies and Security Models*, volume pages, pages 11–20. IEEE, 1982.

[6] N. Heintze and J. G. Riecke. The slam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM.

[7] S. Kahrs. Limits of ml-definability. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, PLILP '96, pages 17–31, London, UK, UK, 1996. Springer-Verlag.

[8] S. Kahrs. Well-going programs can be typed. In *Proceedings of the 6th international conference on Typed lambda calculi and applications*, TLCA'03, pages 167–179, Berlin, Heidelberg, 2003. Springer-Verlag.

[9] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2009. ACM.

[10] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 218–232, Washington, DC, USA, 2011. IEEE Computer Society.

[11] D. McCullough. Specifications for multi-level security and a hook-up. *Security and Privacy, IEEE Symposium on*, 0:161, 1987.

[12] A. C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[13] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, New York, NY, USA, 2002. ACM.

[14] P. Y. A. Ryan and S. A. Schneider. Process algebra and noninterference. *J. Comput. Secur.*, 9(1-2):75–103, Jan. 2001.

[15] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5 – 19, Jan. 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121.

[16] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[17] S. Zdancewic and A. C. Myers. Secure information flow and cps. In *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 46–61, London, UK, 2001. Springer-Verlag.