Environmental Bisimulation for Type-Based Secure Information Flow in λ-Calculus with Declassification

Eijiro Sumii Lin Oshitani* Takeshi Egawa*
Graduate School of Information Sciences
Tohoku University
Sendai, Japan
sumii@tohoku.ac.jp

Abstract

We define a security-typed lambda-calculus with declassification, and develop an environmental bisimulation proof technique for secure information flow in this setting. Unlike traditional security typing that enforces full noninterference, our bisimulation allows proving conditional, intentionally weakened noninterference properties while correctly "leaking" (or publishing) part of the high-security information. Despite the long history of this research area and previous work on declassification, this is, to our knowledge, the first result of such a direct approach to the problem of proving noninterference in a higher-order language with declassification. Our technical development is based on novel treatment of `if`-branches whose conditions are of high secrecy.

Keywords

language-based security, security type system, secrecy/confidentiality and integrity levels, security lattice, noninterference, high-branch closure, λ-calculus

1 Introduction

Secure information flow is a property of systems or programs, that a secret does not leak to an observer or attacker [28]. In a classical model [7], information is classified by security levels, H(igh) and L(ow) in the simplest case, forming a lattice. Static type systems are known to be useful for guaranteeing secure information flow ([47] and a significant amount of subsequent research; see [28] for a general summary of this area).

For instance, consider the conditional expression $\texttt{if}\ s < 42\ \texttt{then}\ 1\ \texttt{else}\ 0$, where s is a secret integer. This expression leaks some information about s, that is, whether it is smaller than 42 or not, and is therefore considered insecure if observed by an attacker. By contrast, the application $(\lambda x.\, 1)s$ of a constant function to a secret argument is trivially secure, as the result is completely independent of the secret.

However, such a security property, called *noninterference* [10], is often too strong a restriction in realistic situations. Indeed, it forbids *any* communication of information between high-security values and low-level observers, as if the former did not exist at all and could even be erased entirely [1].

Declassification (e.g. [31, 48, 49] among others) is a virtual (in the sense that it does not affect the actual contents of information) operation that converts a high-security value into a lower security level. Like type-casts in C, declassification by itself is generally unsafe in that it leaks secrets if unintentionally misused by the programmer—or, worse, if indirectly abused by an attacker through seemingly harmless and legitimate interfaces. Various approaches have been proposed to mitigate this problem of declassification; see [31] for a survey[1].

In the present work, we develop a theory for proving noninterference even for programs with declassification, as a direct, extensional equivalence property like traditional noninterference, albeit with an appropriate condition to deliberately weaken the too strong restriction.

For a (very) simple example, consider the following function

$$f : \texttt{int}_\texttt{H} \to \texttt{int}_\texttt{L} = \lambda s : \texttt{int}_\texttt{H}.\, \texttt{declassify}(\texttt{if}\ s < 42\ \texttt{then}\ 1\ \texttt{else}\ 0)$$

which takes an argument s of type $\texttt{int}_\texttt{H}$ (meaning a high-security integer; for the moment, we only consider secrecy—or, synonymously, confidentiality—and will treat integrity later) and partially leaks the information whether s is smaller than 42 or not.[2] Traditional, full noninterference demands $f(i)$ and $f(j)$ be equal for any integers i and j, which clearly does not hold here. Instead, we prove $f(i) = f(j)$ only for any i and j satisfying $i < 42 \iff j < 42$, that is, $f(i)$ and $f(j)$ are equivalent (or equal, in this simple case of integers) under the condition that i is smaller than 42 if and only if j is. Note that this is an extensional program equivalence property; internal implementation does not matter.[3] For instance, the same property also holds for

$$\lambda s : \texttt{int}_\texttt{H}.\, \texttt{declassify}(\texttt{if}\ s \geq 42\ \texttt{then}\ 0\ \texttt{else}\ 1)$$

or

$$\lambda s : \texttt{int}_\texttt{H}.\, \texttt{if}\ \texttt{declassify}(s < 42)\ \texttt{then}\ 1\ \texttt{else}\ 0.$$

Why do we still need security types even though we directly prove noninterference as extensional program equivalence? The answer is: because types also constrain the attacker or *context* and thereby play an important role in higher-order languages (as most modern programming languages are), where programs can accept other programs as input. Consider, for example, a higher-order function like

$$h : \texttt{int}_\texttt{H} \to_\texttt{L} (\texttt{int}_\texttt{H} \to_\texttt{L} \texttt{int}_\texttt{L}) \to_\texttt{L} \texttt{int}_\texttt{L} = \lambda s : \texttt{int}_\texttt{H}.\, \lambda g : \texttt{int}_\texttt{H} \to_\texttt{L} \texttt{int}_\texttt{L}.\, g(s)$$

*The second and third authors contributed to this work while they were enrolled in the master's program at the graduate school.

[1] or https://dblp.org/search?q=declassification, or even https://dl.acm.org/action/doSearch?AllField=declassification, for more recent and complete lists of publications, which we do not aim to enumerate in this paper

[2] This example is (extremely) simplified for the sake of exposition; a more realistic case would, for instance, compute and publish only partial information—such as an average age—from a collection of private data.

[3] One may argue that internal implementations *do* matter as they reflect programmers' intentions; this is a different—even complementary—approach, in that intensional restrictions do not always imply extensional noninterference, nor vice versa.

where $\rightarrow_{\mathsf{L}}$ means the function itself is (well-typed but) low-security, which we omitted in the previous examples for brevity. Then we have $h(i)(a) = h(j)(a)$ for any integers i, j and any well-typed (but low-level) attacker function a, as our type system forbids low-level contexts from declassifying high-security values (T-Dec in Figure 2 requires $conf(l) \preceq conf(\alpha)$; details in Section 4). Note that assuming attackers conform to static disciplines like well-typedness is not unrealistic in hosting environments or on virtual machines (with a network connection that may leak information), or when a user executes downloaded code after checking or within a sandbox. Moreover, the "attacker" may not necessarily be a malicious villain but could also be a well-intentioned yet flawed program, for which quick static type checking would likely help.[4]

Integrity, that is, whether a value is trustworthy or not, which we have ignored so far, introduces another axis. For instance, consider the pairs

$$\langle i_{\mathsf{HH}}, \lambda s : \mathtt{int}_{\mathsf{HH}}.\, \mathtt{declassify}(\mathtt{if}\ s < 42\ \mathtt{then}\ 1\ \mathtt{else}\ 0)\rangle$$

and

$$\langle j_{\mathsf{HH}}, \lambda s : \mathtt{int}_{\mathsf{HH}}.\, \mathtt{declassify}(\mathtt{if}\ s < 42\ \mathtt{then}\ 0\ \mathtt{else}\ 1)\rangle$$

where the second H in the label HH means high integrity. They are contextually equivalent as far as $i < 42 \iff \neg(j < 42)$ since the arguments to the function $\lambda s : \mathtt{int}_{\mathsf{HH}}.\, \mathtt{declassify}(\mathtt{if}\ s < 42\ \mathtt{then}\ 1\ \mathtt{else}\ 0)$ must be high-integrity, for which only i or j is possible, respectively, on the left- or right-hand side of the equivalence. Obviously, this equivalence would not hold if the integrity of s were allowed to be low, i.e., if it could be an arbitrary integer created by an attacker—or if the attacker could create "high-integrity" integers. The latter case illustrates again that security depends on the "level" of the context as we formally define later. It should also be noted that attackers can always create HL (high-secrecy but low-integrity) values by "classification" via "upcasts," namely, subtyping.

Interested readers can find more examples in Section 7, ignoring the proofs for the moment.

To prove those conditional noninterference properties as program equivalence, we adapt *environmental bisimulation* ([33, 35, 43–46] along with a number of later adaptations), which is a sound and complete (though inevitably non-automatic, as the problem itself is generally undecidable) proof technique for contextual equivalence in higher-order languages. Environmental bisimulation fits the present purpose as it has been known to work well with various forms of information hiding (such as "sealing" [43, 45]—also known as perfect/symbolic/formal encryption [3, 8]—and type abstraction [44, 46]) and higher-order functions or processes [33, 35]. Although outside the direct scope of the present paper, it also scales to various language features such as state [14, 33, 35, 40], objects [13], concurrency [33, 35, 38], probability [36, 37], and distribution [22, 23]—all of which are higher-order—while logical relations [25, 27, 41, 42] and applicative bisimulation [4] are not readily applicable to concurrency and information hiding [12], respectively.

To the best of our knowledge, this is the first result that directly formulates and proves conditional noninterference as extensional program equivalence in a higher-order language; see also Section 2.

The rest of this paper proceeds as follows. Section 2 discusses (direct) related work. Section 3 introduces the syntax of our language, with (subtyping and) typing rules in Section 4 and operational semantics (and definition of equivalence) in Section 5. Section 6 develops our bisimulation, Section 7 presents some examples, and Section 8 concludes.

[4] As a rather different approach, dynamic labeling as in [5] may be a possible alternative.

2 Related Work

As already mentioned, research in this field has a long history, dating back to the 1970s at least. Leaving more thorough surveys to other publications [28, 31][6, Chapter 6], we here focus only on previous work that is more directly related to ours.

Delimited information release [29] formalized a form of conditional [29, Definition 2] noninterference for a first-order imperative language (syntax on [29, p. 3]). It crucially depended on the language being first-order (e.g. the definition of equivalence [29, p. 4]).

Relaxed noninterference [6, 16, 20] guarantees noninterference while allowing declassification via "downgrading policies," which are a set of functions from higher-secrecy values to lower-secrecy, predefined by the programmer. Each declassification operation is allowed if and only if it conforms to the corresponding downgrading policy. As such, checking noninterference is much easier in many cases (though still undecidable in general) but a program which is semantically secure as a whole (like the previous examples) is rejected if it does not use the downgrading policies in the syntactically restricted manner.

Partial equivalence relations (PERs) [30] are useful as models of partial information leaks induced by declassification. They provide elegant definitions, though not a proof technique by themselves, for (conditional) noninterference.

Logical relations (another historical area, going back at least to [25]; see, e.g., [18, Chapter 8] for a textbook) are relations on λ-terms defined by induction on types (at least traditionally, though variations exist). They are also mathematically elegant, and useful (among other purposes) for proving program equivalence, in particular with information hiding as in [27, 42] to name just a few examples. However, no sound (and complete) logical relations seem to be known for general (as opposed to syntactically restricted [6, 16, 20]) declassification, perhaps since such declassification is somewhat like type-casts and challenging for the type-directed nature of logical relations. Note that, for example, high-secrecy integers 123_{H} and 45_{H} *by themselves* are contextually equivalent (to low-level attackers), and the function $\lambda x{:}\mathtt{int}_{\mathsf{H}}.\, \mathtt{declassify}(x)$ *per se* is also contextually equivalent to itself, but their pair $\langle 123_{\mathsf{H}}, \lambda x : \mathtt{int}_{\mathsf{H}}.\, \mathtt{declassify}(x)\rangle$ is *not* contextually equivalent to $\langle 45_{\mathsf{H}}, \lambda x{:}\mathtt{int}_{\mathsf{H}}.\, \mathtt{declassify}(x)\rangle$. While one may consider assigning some "special" relations to high-security types, they would differ for each program as above. In addition, integrity would require more considerations since attackers can create arbitrary low-integrity values, even using high-secrecy data, though the results would again be high-secrecy. Although interesting, we leave these challenges to other work[5] and here adopt environmental bisimulation instead, which is arguably more straightforward (and would even scale to concurrency).

[5] Recently and independently (after early versions of our work [9, 21]), Rajani, Coleman, and Kanabar [26] developed a higher-order language and sound (though not claimed to be complete) logical relations with modalities for a form of declassification inspired by delimited information release [29], relaxed noninterference [16], and modal types for security levels [1].

Menz et al. [17] defined logical relations for a different, intensional *where*-declassification [31]. Our conditional noninterference can be regarded as a form of extensional *what*-declassification [31]. The latter, especially if extended with state, might be able to simulate some other variants of declassification as equivalence between programs with and without dynamic assertions that abort the program when security checks fail. [31] is an extensive survey of various kinds of declassification as of the time it was published.

Robust declassification [48, 49] is now a classical approach that prevents low-integrity attackers from declassifying high-secrecy data. We partially adopt it (rule T-Dec in Section 4). However, unlike standard robust declassification, we allow declassification by high-security programs even after branching on low-integrity conditions (α in T-IF remains the same among its premises and conclusion). We instead guarantee the security of programs as (appropriately conditional) noninterference via environmental bisimulations.

3 Syntax

Our language is a security-typed λ-calculus with declassification. Its syntax is given in Figure 1.

Security levels are elements of some (bounded) security lattice $\mathcal{L}$ with partial order $\preceq$ and join $\sqcup$ and meet $\sqcap$ operations, and are used for designating confidentiality c and integrity e. We write H and L for the top and bottom levels, respectively.

Security *labels* l, k, and α are pairs of confidentiality and integrity. The pairs (c, e) are often written just ce, like HL for (H, L). As usual, we define $(c_1, e_1) \preceq (c_2, e_2) \iff c_1 \preceq c_2$ and $e_2 \preceq e_1$. Also, we will use $\mathit{conf}(c, e) = c$ and $\mathit{integ}(c, e) = e$. Integrity or even entire labels may be omitted when unimportant, while we never omit only confidentiality.

Types are mutually defined by bare types τ and labeled types s, t (which are τ_l). We sometimes write $s_1 \times_l s_2$ for $(s_1 \times s_2)_l$ and $s_1 \to_l s_2$ for $(s_1 \to s_2)_l$.

We allow arbitrary nesting of security types of any levels. In standard security type systems, some of them are not useful: for instance, $\mathtt{int}_{\mathsf{L}} \times_{\mathsf{H}} \mathtt{int}_{\mathsf{L}}$ is not very different from $\mathtt{int}_{\mathsf{H}} \times_{\mathsf{H}} \mathtt{int}_{\mathsf{H}}$ since their projections would anyway be given the type $\mathtt{int}_{\mathsf{H}}$. However, they *are* crucially different in our system where the pair itself can be declassified while its elements are not. Note also that the combination of the paring itself, rather than individual elements, may convey secret information; we will see this by Example 7.3 in Section 7.

Our term language is standard simply typed λ-calculus except for the security labels, protection, and declassification. We use terms as possibly multi-hole contexts (for closed values only, which suffices for our purpose) with free variables as holes and substitutions as (capture-avoiding, which does not matter for *closed* values) hole filling. We often use the syntactic sugar $\mathtt{let}\ x : s = M\ \mathtt{in}\ N$ for $(\lambda x : s.\, N)M$. Tuples with more than two elements can be represented by nested pairs of the same security level. We assume some primitive operations on integers, including equality. We define $U_l \sqcup l' = U_{l \sqcup l'}$. Again, labels (as well as type annotations) are omitted when unimportant or clear from the context (e.g. via externally specified types).

Throughout the paper, we sometimes use _ as a placeholder when something is omitted. Also, we use overbars to abbreviate sequences, like $\overline{x}$ for $x_1, \ldots, x_n$ and $(\overline{V}, \overline{V}')$ for $(V_1, V_1'), \ldots, (V_m, V_m')$.

4 Typing

The (subtyping and) typing rules are given in Figure 2. It is clear by induction that subtyping is reflexive and transitive. Type judgments are of the form $\Gamma \vdash_\alpha M : s$ where Γ is a standard type environment (omitted when empty) and α is, intuitively, the "power" or "capabilities" of the term M, in particular when it is used as a context: as we will define in Section 5, contexts who observe the (in)equivalence of terms are restricted so that they cannot create high-integrity (nor declassify high-confidentiality) values; this restriction is imposed by the conditions $\mathit{integ}(l) \preceq \mathit{integ}(\alpha)$ and $e \preceq \mathit{integ}(\alpha)$ in T-Bool, T-Int, T-Pair, T-Fun, and T-Op (and by $\mathit{conf}(l) \preceq \mathit{conf}(\alpha)$ in T-Dec). Note that the integrity e in T-Op is also restricted since integer operations do not just consume but also produce values: for instance, the equality $i = i$ for an integer i always returns $\mathtt{true}$, hence the same integrity restriction as in T-Bool.

The other conditions are standard. Note that low-secrecy contexts can also apply high-secrecy functions, project from high-secrecy pairs, branch over high-secrecy Booleans, etc. although the results will still be of high secrecy.

For brevity, we write the $\Gamma \vdash M : s$ for $\Gamma \vdash_{\mathsf{HH}} M : s$. The following lemma guarantees that upgrading confidentiality or integrity gives no less capabilities to a term (which often acts as a context).

Lemma 4.1 (monotonicity of typing with respect to levels in α). *If $\Gamma \vdash_{ce} M : s$, then $\Gamma \vdash_{c'e'} M : s$ for any $c' \succeq c$ and $e' \succeq e$; in particular, $\Gamma \vdash M : s$.*

Proof. By induction on the derivation of $\Gamma \vdash_{ce} M : s$. Note that all the conditions concerning α in the typing rules are monotone for α. □

One may wonder why the order on integrity in this lemma is *not* reversed as is usual for labels on values. This is because α in $\vdash_\alpha$ represents the capabilities of the term (or context) being typed—as already explained above—rather than the security of a value; to clarify this difference, we could instead write, say, $\vdash_{c;e}$, which we do not, just favoring simplicity.

The next lemma justifies, in terms of typing, filling the holes of contexts as in the rest of this paper. Again, we use overbars to abbreviate sequences, like $\vdash \overline{V} : \overline{s}$ for $\vdash V_1 : s_1, \ldots, \vdash V_m : s_m$ and $\overline{x} : \overline{s}$ for $x_1 : s_1, \ldots, x_n : s_n$. We also write $[\overline{V}/\overline{x}]C$ for simultaneous substitutions of $x_1, \ldots, x_n$ with $V_1, \ldots, V_n$ in C.

Lemma 4.2. *If $\vdash \overline{V} : \overline{s}$ and $\overline{x} : \overline{s} \vdash_{\alpha'} C : t$, then $\vdash [\overline{V}/\overline{x}]C : t$.*

Proof. By induction on the derivation of $\overline{x} : \overline{s} \vdash_{\alpha'} C : t$, noting that all the conditions on α while deriving $\vdash [\overline{V}/\overline{x}]C : t$ are trivially satisfied as $\alpha = \mathsf{HH}$ (recall it is just omitted for brevity). □

5 Operational Semantics

The operational semantics of our language is defined by reduction $\to$ as in Figure 3, using *evaluation contexts* E where the *hole* $[\,]$ is a special variable, *hole filling* $E[M]$ is a (capture-avoiding, which again makes no difference for closed terms) substitution of $[\,]$ with

$c, e \in \mathcal{L}$	security levels	$M, N, C ::=$	terms and (term) contexts
		$\texttt{true}_l,\ \texttt{false}_l$	Booleans
$l, k, \alpha ::= (c, e)$	security labels	$\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2$	branchings
		i_l	integers
$\tau ::=$	bare types	$op_e(M_1, M_2)$	integer operations
$\texttt{bool}$	Boolean type	$\langle M_1, M_2 \rangle_l$	parings
$\texttt{int}$	integer type	$\#_1(M),\ \#_2(M)$	projections
$s_1 \times s_2$	pair types	$(\lambda x : s.\, M)_l$	functions
$s_1 \to s_2$	function types	MN	function applications
		x	variables
$s, t ::= \tau_l$	labeled types	$\texttt{protect}_l(M)$	protections
		$\texttt{declassify}_c(M)$	declassifications
$T, U ::=$	bare values		
$\texttt{true},\ \texttt{false}$	Booleans	$V, W, D ::=$	values and value contexts
i	integers	U_l	labeled values
$\langle V_1, V_2 \rangle$	pairs	x	holes of contexts
$\lambda x : s.\, M$	functions		

Figure 1: Syntax

$$\frac{}{\texttt{bool} \le \texttt{bool}}\ \text{S-Bool} \qquad \frac{}{\texttt{int} \le \texttt{int}}\ \text{S-Int} \qquad \frac{s_1 \le s_1' \quad s_2 \le s_2'}{s_1 \times s_2 \le s_1' \times s_2'}\ \text{S-Pair}$$

$$\frac{s_1' \le s_1 \quad s_2 \le s_2'}{s_1 \to s_2 \le s_1' \to s_2'}\ \text{S-Fun} \qquad \frac{l \le l' \quad \tau \le \tau'}{\tau_l \le \tau_{l'}'}\ \text{S-Label} \qquad \frac{\Gamma \vdash_\alpha M : s \quad s \le s'}{\Gamma \vdash_\alpha M : s'}\ \text{T-Sub}$$

$$\frac{b \in \{\texttt{true}, \texttt{false}\} \quad \mathit{integ}(l) \le \mathit{integ}(\alpha)}{\Gamma \vdash_\alpha b_l : \texttt{bool}_l}\ \text{T-Bool} \qquad \frac{\Gamma \vdash_\alpha M : \texttt{bool}_l \quad \Gamma \vdash_\alpha N_i : s,\ i = 1, 2}{\Gamma \vdash_\alpha \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 : s \sqcup l}\ \text{T-If}$$

$$\frac{\mathit{integ}(l) \le \mathit{integ}(\alpha)}{\Gamma \vdash_\alpha i_l : \texttt{int}_l}\ \text{T-Int} \qquad \frac{\begin{array}{c}op \text{ is a binary operator on } \texttt{int} \text{ to } \tau \in \{\texttt{int}, \texttt{bool}\} \\ \Gamma \vdash_\alpha M_i : \texttt{int}_{l_i},\ i = 1, 2 \quad e \le \mathit{integ}(\alpha)\end{array}}{\Gamma \vdash_\alpha op_e(M_1, M_2) : \tau_{l_1 \sqcup l_2 \sqcup (\mathsf{L}, e)}}\ \text{T-Op}$$

$$\frac{\Gamma \vdash_\alpha M_i : s_i,\ i = 1, 2 \quad \mathit{integ}(l) \le \mathit{integ}(\alpha)}{\Gamma \vdash_\alpha \langle M_1, M_2 \rangle_l : (s_1 \times s_2)_l}\ \text{T-Pair} \qquad \frac{\Gamma \vdash_\alpha M : (s_1 \times s_2)_l \quad i \in \{1, 2\}}{\Gamma \vdash_\alpha \#_i(M) : s_i \sqcup l}\ \text{T-Proj}$$

$$\frac{\Gamma, x : s_1 \vdash_\alpha M : s_2 \quad \mathit{integ}(l) \le \mathit{integ}(\alpha)}{\Gamma \vdash_\alpha (\lambda x : s_1.\, M)_l : (s_1 \to s_2)_l}\ \text{T-Fun} \qquad \frac{\Gamma \vdash_\alpha M : (s_1 \to s_2)_l \quad \Gamma \vdash_\alpha N : s_1}{\Gamma \vdash_\alpha MN : s_2 \sqcup l}\ \text{T-App} \qquad \frac{}{\Gamma \vdash_\alpha x : \Gamma(x)}\ \text{T-Var}$$

$$\frac{\Gamma \vdash_\alpha M : s}{\Gamma \vdash_\alpha \texttt{protect}_l(M) : s \sqcup l}\ \text{T-Prot} \qquad \frac{\Gamma \vdash_\alpha M : \tau_l \quad c \le \mathit{conf}(l) \le \mathit{conf}(\alpha)}{\Gamma \vdash_\alpha \texttt{declassify}_c(M) : \tau_{(c, \mathit{integ}(l))}}\ \text{T-Dec}$$

Figure 2: Subtyping and Typing Rules

M, and $\Gamma \vdash_\alpha E[t] : s \overset{\text{def}}{\iff} \Gamma, z : t \vdash_\alpha E[z] : s$ for fresh variable z. It is a standard call-by-value, left-to-right, small-step evaluation. Declassification (E-Dec) only changes the confidentiality level; endorsement (upgrade of integrity) could be treated similarly but is omitted in the present paper. We write $\twoheadrightarrow$ for the reflexive transitive closure of $\to$.

Theorem 5.1 (progress and preservation). *For any α, M, and s, if $\vdash_\alpha M : s$, then either M is a value, or else $M \to N$ for some N with $\vdash_\alpha N : s$.*

Proof. Standard induction on the derivation of $\vdash_\alpha M : s$, with an also standard substitution lemma (with the same α). □

The next definition materializes the intuition given in Section 4 about the role of α in $\vdash_\alpha$ as the level of contexts:

Definition 5.2 (α-contextual equivalence). Values $\overline{V}$ and $\overline{V}'$ with $\vdash \overline{V} : \overline{s}$ and $\vdash \overline{V}' : \overline{s}$ are called *α-contextually equivalent* at type $\overline{s}$ when: $[\overline{V}/\overline{x}]C \twoheadrightarrow b_k \iff [\overline{V}'/\overline{x}]C \twoheadrightarrow b_{k'}$ for any b, k, k' and $\overline{x} : \overline{s} \vdash_\alpha C : \texttt{bool}_l$ with $\mathit{conf}(l) \le \mathit{conf}(\alpha)$.

$$\frac{}{\texttt{if true}_l \texttt{ then } N_1 \texttt{ else } N_2 \longrightarrow \texttt{protect}_l(N_1)}\ \text{E-IfTrue}$$

$$\frac{}{\texttt{if false}_l \texttt{ then } N_1 \texttt{ else } N_2 \longrightarrow \texttt{protect}_l(N_2)}\ \text{E-IfFalse}$$

$$\frac{op(i,j) = U \in \mathbf{Int} \cup \{\texttt{true}, \texttt{false}\}}{op_e(i_l, j_k) \longrightarrow U_{l \sqcup k \sqcup (\mathsf{L}, e)}}\ \text{E-Op} \qquad \frac{i \in \{1,2\}}{\#i(\langle V_1, V_2\rangle_l) \longrightarrow \texttt{protect}_l(V_i)}\ \text{E-Proj}$$

$$\frac{}{(\lambda x : s.\, M)_l V \longrightarrow \texttt{protect}_l([V/x]M)}\ \text{E-App} \qquad \frac{}{\texttt{protect}_l(V) \longrightarrow V \sqcup l}\ \text{E-Prot}$$

$$\frac{}{\texttt{declassify}_c(U_l) \longrightarrow U_{(c, integ(l))}}\ \text{E-Dec} \qquad \frac{M \longrightarrow N}{E[M] \longrightarrow E[N]}\ \text{E-Context}$$

$E ::=$ evaluation contexts
$[\,] \mid \texttt{if } E \texttt{ then } N_1 \texttt{ else } N_2 \mid op_e(E, M) \mid op_e(V, E) \mid \langle E, M\rangle_l \mid \langle V, E\rangle_l \mid \#_1(E) \mid \#_2(E)$
$EN \mid VE \mid \texttt{protect}_l(E) \mid \texttt{declassify}_c(E)$

Figure 3: Reduction and Evaluation Contexts

Contextual equivalence is a classical and standard definition in programming language theory [19] and non-interference of security-typed λ-calculi. (e.g. [11, Theorem 2.3], [2, Theorem 4.2], and [39, Section 2.5]). Our α-contextual equivalence is a generalization parameterized by the capabilities α of the contexts as attackers.

Our contextual equivalence is defined simultaneously on multiple values for the sake of simpler correspondence with the typed value relations

$$\{(\overline{V}, \overline{V}', \overline{s}) \mid \overline{V} \text{ and } \overline{V}' \text{ are } \alpha\text{-contextually equivalent at type } \overline{s}\}$$

indexed by α, which are particularly convenient for the completeness proof of environmental bisimulation later. Standard contextual equivalence (for closed values, while possibly open terms can be translated into closed values by λ-abstractions) under low-level attackers can be thought as HH-contextual equivalence for single V_1 and V_1'. Note also that, even without declassification, (in)equivalences obviously depend on the levels of contexts; for instance, the functions $\lambda x : \texttt{int}_{\mathsf{LH}}.\, x + 1$ and $\lambda x : \texttt{int}_{\mathsf{LH}}.\, x - 1$ *by themselves* are LL-contextually equivalent at type $\texttt{int}_{\mathsf{LH}} \to_{\mathsf{LH}} \texttt{int}_{\mathsf{LH}}$ since the context cannot obtain *any* high-integrity integers and, as a result, can *never* apply these functions, while HH- or even LH-contexts can easily distinguish them by simply creating a high-integrity integer. For an even simpler example, for any integrity level _, the high-secrecy integers $42_{\mathsf{H\_}}$ and $24_{\mathsf{H\_}}$ at type $\texttt{int}_{\mathsf{H\_}}$ are L_-contextually equivalent but not H_-contextually equivalent.

6 Our Environmental Bisimulation

The following two definitions are standard in (typed) environmental bisimulations [40, 46]. Intuitively, an *environment* $R = \{(\overline{V}, \overline{V}', \overline{s})\}$ represents the knowledge of a context (which acts as an attacker) about the corresponding values V_i and V_i' on the left- and right-hand sides of the equivalence, while each (R, M, M', s) in an *environmental relation* X represents a configuration where M and M' (of type s) are respectively running on the two sides.

Definition 6.1. A *typed value relation* or *environment* R (or S) is a set of triples of the form (V, V', s) with $\vdash V : s$ and $\vdash V' : s$.

Lemma 6.2. *For each element $(U_l, U_{l'}', \tau_{l''})$ of typed value relations, we have $l \preceq l''$ and $l' \preceq l''$.*

Proof. By induction (and inversion) on typing (and subtyping) derivations. □

Definition 6.3. An *environmental relation* X is a set where each element is either a typed value relation R or a quadruple of the form (S, M, M', s) with $\vdash M : s$ and $\vdash M' : s$.

As above, an environmental relation can also include a set of multiple R's since the "attacker's knowledge" represented by each R may "evolve over time," that is, increase by reductions. In many actual use cases, however, it can be taken just as the singleton of the "limit" or an (not necessarily the least) upper bound of such R's, as in Section 7. Besides the examples, see Definition 6.11 or previous work [40, 46] to observe how these definitions work in detail.

We then give an auxiliary definition of upward closures, which captures the intuition that an "upcast" (i.e., upgrading security levels) is always possible and can (even should, for ease of proofs including soundness) be included in the relations.

Definition 6.4 (upward closure). Define $(U_k, U_{k'}', \tau_{k''}) \geq (U_l, U_{l'}', \tau_{l''}) \iff k \geq l, k' \geq l', k'' \geq l'', k \preceq k''$, and $k' \preceq k''$. Then $R^{\mathrm{up}} = \{\, (V, V', s) \mid (V, V', s) \geq (W, W', t) \in R \,\}$.

Lemma 6.5. *If R is a typed value relation, then R^{up} is also a typed value relation.*

Proof. By T-Sub and S-Label. □

We omit similar lemmas (and straightforward proofs) for other definitions of closures that follow.

Obviously, upward closure is idempotent, that is, $(R^{\mathrm{up}})^{\mathrm{up}} = R^{\mathrm{up}}$ for any R since $\preceq$ is transitive. Also $R^{\mathrm{up}} \supseteq R$ since $\preceq$ is reflexive.

The next definition introduces contexts (or attackers) into the relations as in previous environmental bisimulations. This is crucial for higher-order languages such as λ-calculus, where the contexts or "attackers" can create (and supply to the system) their own functions, which may contain previously given high-security values.

Definition 6.6 (α-context closure). $R^{\mathrm{ctx}}_\alpha = S^{\mathrm{up}}$ where

$$S = \{\ (\ [\overline{V}/\overline{x}]D,\ [\overline{V'}/\overline{x}]D,\ s\ )\ \mid\ \overline{x} : \overline{s} \vdash_\alpha D : s,\ (\overline{V}, \overline{V'}, \overline{s}) \in R^{\mathrm{up}}\ \}.$$

Note in particular that, by taking $D = x_1$, we have $R^{\mathrm{ctx}}_\alpha \supseteq R^{\mathrm{up}}$ ($\supseteq R$) for any α. By the idempotence of upward closure, we also have $(R^{\mathrm{up}})^{\mathrm{ctx}}_\alpha = (R^{\mathrm{ctx}}_\alpha)^{\mathrm{up}} = R^{\mathrm{ctx}}_\alpha$. Furthermore:

Lemma 6.7 (idempotence of context closure). $(R^{\mathrm{ctx}}_\alpha)^{\mathrm{ctx}}_\alpha = R^{\mathrm{ctx}}_\alpha$ *for any α and R.*

Proof. By composing contexts, like $[\overline{D}/\overline{x}]D_0$. □

In the above definition, the upward closures appear twice since, intuitively, contexts can upcast (via `protect()`) any values they know, almost anywhere and anytime. Alternatively, we could implicitly impose all environmental relations to be upward-closed in the first place, which in effect would require similar arguments throughout our developments, even where unnecessary; moreover, extreme care would be needed if the upward closures are implicit, especially when inverting such definitions for various environmental relations. We avoid these difficulties and explicitly take upward closures instead.

A key definition below in our development accounts for high-secrecy branches (or just "high branches" for short), such as `if` h `then` $f(1)$ `else` $g(2) + 3$ when h has type $\mathtt{bool}_{\mathsf{H}}$ (note that such h can be created anytime using $\mathtt{protect}_{\mathsf{H}}(\_)$ even by the attacker, albeit with low integrity). The point here is that the `then` and `else` branches can execute completely different computations (although their results will be marked H). Our bisimulation must take such splitting into account. However, naively pursuing each branch separately would lead to a mostly duplicated set of conditions and (rather redundant) proofs, which we avoid by separately increasing only the environments.

Definition 6.8 (high-branch closure for environment). $R^{\mathrm{if}}_\alpha = S^{\mathrm{ctx}}_\alpha$ for

$$\begin{array}{ll} S = R \cup \{\ (\ \overline{W} \sqcup \overline{l},\ \overline{W'} \sqcup \overline{l'},\ \overline{t} \sqcup \overline{l''}\ )\ \mid & \textbf{(a)} \\ (\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in R^{\mathrm{up}},\ \overline{U} \neq \overline{U'},\ \mathit{conf}(\overline{l''}) \not\leq \mathit{conf}(\alpha), & \textbf{(b)} \\ (\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in R^{\mathrm{ctx}}_\alpha\ \}. & \textbf{(c)} \end{array}$$

Or [6] just unfolding the outer context closure S^{ctx}_α, we have $R^{\mathrm{if}}_\alpha = Q^{\mathrm{up}}$ for

$$\begin{array}{l} Q = \{\ (\ [\overline{V}, \overline{W_1}/\overline{x}, \overline{y}]D,\ [\overline{V'}, \overline{W_1'}/\overline{x}, \overline{y}]D,\ s\ )\ \mid \\ \overline{x} : \overline{s}, \overline{y} : \overline{t_1} \vdash_\alpha D : s,\ (\overline{V}, \overline{V'}, \overline{s}) \in R^{\mathrm{up}}, \\ (\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in R^{\mathrm{up}},\ \overline{U} \neq \overline{U'},\ \mathit{conf}(\overline{l''}) \not\leq \mathit{conf}(\alpha), \\ (\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in R^{\mathrm{ctx}}_\alpha, \\ (\overline{W_1}, \overline{W_1'}, \overline{t_1}) \geq (\ \overline{W} \sqcup \overline{l},\ \overline{W'} \sqcup \overline{l'},\ \overline{t} \sqcup \overline{l''}\ )\ \}. \end{array}$$

[6] This alternative and equivalent definition is shown here only for the sake of later references and can be ignored for the moment.

Intuitively, line (**b**) takes different Booleans $\overline{U}$ and $\overline{U}'$ of levels l and l', respectively, with type $\mathtt{bool}_{\overline{l''}}$, which is higher than (strictly speaking, neither less than nor equal to) α. Line (**c**) chooses values $\overline{W}$ and $\overline{W}'$ separately from the left- and right-hand sides of R^{ctx}_α, and then line (**a**) upgrades them with the levels of the Booleans, as will happen in high-secrecy branches.

Again, note $R^{\mathrm{if}}_\alpha \supseteq R^{\mathrm{ctx}}_\alpha$ ($\supseteq R^{\mathrm{up}} \supseteq R$) by taking $\overline{W}$ and $\overline{W'}$ to be empty.

One may wonder why the context closures are taken twice in the above definition. The reason is contexts like `let` h' = (`if` h `then` C_1 `else` C_1') `in` C_0, where the inner contexts C_1 and C_1' (separate for the `then` and `else` clauses) need to be upgraded with the level of h before each of them is put into the outer, common context C_0.

Like the definition above, the following lemma is a key in our development—hence a detailed proof in the Appendix, which may seem syntactically complicated but is essentially straightforward given the definition and intuition above; it can be skipped, however, for reading the rest of this paper.

Lemma 6.9 (idempotence of high-branch closure). $(R^{\mathrm{if}}_\alpha)^{\mathrm{if}}_\alpha = R^{\mathrm{if}}_\alpha$.

Proof. Unfolding the definitions (with idempotence of upward and context closures), we have $(R^{\mathrm{if}}_\alpha)^{\mathrm{if}}_\alpha = Q^{\mathrm{up}}$ and $R^{\mathrm{if}}_\alpha = S^{\mathrm{ctx}}_\alpha$ for

$$\begin{array}{l} Q = \{\ (\ [\overline{V}, \overline{W_1}/\overline{x}, \overline{y}]D,\ [\overline{V'}, \overline{W_1'}/\overline{x}, \overline{y}]D,\ s\ )\ \mid \\ \overline{x} : \overline{s}, \overline{y} : \overline{t_1} \vdash_\alpha D : s,\ (\overline{V}, \overline{V'}, \overline{s}) \in (R^{\mathrm{if}}_\alpha)^{\mathrm{up}}\ (= S^{\mathrm{ctx}}_\alpha), \\ (\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in (R^{\mathrm{if}}_\alpha)^{\mathrm{up}},\ \overline{U} \neq \overline{U'},\ \mathit{conf}(\overline{l''}) \not\leq \mathit{conf}(\alpha), \\ (\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in (R^{\mathrm{if}}_\alpha)^{\mathrm{ctx}}_\alpha\ (= S^{\mathrm{ctx}}_\alpha), \\ (\overline{W_1}, \overline{W_1'}, \overline{t_1}) \geq (\ \overline{W} \sqcup \overline{l},\ \overline{W'} \sqcup \overline{l'},\ \overline{t} \sqcup \overline{l''}\ )\ \}. \end{array}$$

and

$$\begin{array}{l} S = R \cup \{\ (\ \overline{W} \sqcup \overline{l},\ \overline{W'} \sqcup \overline{l'},\ \overline{t} \sqcup \overline{l''}\ )\ \mid \\ (\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in R^{\mathrm{up}},\ \overline{U} \neq \overline{U'},\ \mathit{conf}(\overline{l''}) \not\leq \mathit{conf}(\alpha), \\ (\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in R^{\mathrm{ctx}}_\alpha\ \}. \end{array}$$

We aim to prove $(R^{\mathrm{if}}_\alpha)^{\mathrm{if}}_\alpha = R^{\mathrm{if}}_\alpha$ by, in the right-hand side of the above Q, replacing

- $(\overline{V}, \overline{V'}, \overline{s}) \in S^{\mathrm{ctx}}_\alpha$,
- $(\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in S^{\mathrm{ctx}}_\alpha$, and
- $(\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in S^{\mathrm{ctx}}_\alpha$

with

- $(\overline{V}, \overline{V'}, \overline{s}) \in R^{\mathrm{up}}$,
- $(\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in R^{\mathrm{up}}$, and
- $(\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in R^{\mathrm{ctx}}_\alpha$

respectively, without changing the resulting set Q^{up} as a whole, as follows:

- $(\overline{V}, \overline{V'}, \overline{s}) \in S^{\mathrm{ctx}}_\alpha$ can be replaced with $(\overline{V}, \overline{V}', \overline{s}) \in R^{\mathrm{up}}$ by composing the context for S^{ctx}_α into D and replacing the elements from S^{up} with $(\overline{V}, \overline{V'}, \overline{s}) \in R^{\mathrm{up}}$ and $(\overline{W_1}, \overline{W_1'}, \overline{t_1}) \geq (\ \overline{W} \sqcup \overline{l},\ \overline{W'} \sqcup \overline{l'},\ \overline{t} \sqcup \overline{l''}\ )$ with $(\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in R^{\mathrm{ctx}}_\alpha$.
- $(\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in S^{\mathrm{ctx}}_\alpha$ can be replaced with $(\overline{U}_{\overline{l}}, \overline{U'}_{\overline{l'}}, \mathtt{bool}_{\overline{l''}}) \in R^{\mathrm{up}}$ by the following case analysis on each element $(U_{i\,l_i}, U'_{i\,l'_i}, \mathtt{bool}_{l''_i})$ of S^{ctx}_α.

- If the element comes from the common context for S_α^{ctx} itself (i.e., without using any values from S^{up}), then $U_i = U_i'$, which contradicts with another condition $\overline{U} \neq \overline{U}'$.
- If the element comes from S^{up} itself (i.e., the context for S_α^{ctx} is just a variable), then it comes from either R^{up} or the other part of S^{up}. We are done in the former case. In the latter case, we have $(U_{i\,l_i}, U'_{i\,l'_i}, \mathtt{bool}_{l''_i}) \geq (\ W_j \sqcup l_j,\ \overline{W'}_j \sqcup l'_j,\ t_j \sqcup l''_j\ )$ with $(U_{j\,l_j}, U'_{j\,l'_j}, \mathtt{bool}_{l''_j}) \in R^{\mathrm{up}}$, $U_j \neq U'_j$, $\mathit{conf}(l''_j) \not\leq \mathit{conf}(\alpha)$, and $(W_j, \_, t_j)$, $(\_, W'_j, t_j) \in R_\alpha^{\mathrm{ctx}}$. Since $l_i \geq l_j$, $l'_i \geq l'_j$, and $l''_i \geq l''_j$, we are done with $(U_{j\,l_i}, U'_{j\,l'_i}, \mathtt{bool}_{l''_i}) \in R^{\mathrm{up}}$.
- The other cases are impossible since U_i and U'_i are values of type `bool`, that is, either `true` or `false`.

- $(\overline{W}, \_, \overline{t})$, $(\_, \overline{W'}, \overline{t}) \in S_\alpha^{\mathrm{ctx}}$ can be replaced with $(\overline{W}, \_, \overline{t})$, $(\_, \overline{W'}, \overline{t}) \in R_\alpha^{\mathrm{ctx}}$ as follows. Take $(W_i, \_, t_i) \in S_\alpha^{\mathrm{ctx}}$ (the case for $(\_, W'_i, t_i) \in S_\alpha^{\mathrm{ctx}}$ is similar). It is constructed from elements of S^{up}. As for elements coming from R^{up}, we are done. Elements coming from the other part of S^{up} comes from R_α^{ctx}. We are done by composing the context for R_α^{ctx} into the context for S_α^{ctx}.

□

Each of the above definitions of closures also incorporates previous closures in order to combine all of them, so "high-branch closure" for instance should perhaps be called "upward and α-context and high-branch closure" but we abbreviate them for conciseness.

We then extend the definition of high-branch closure from environments to environmental relations:

Definition 6.10 (high-branch closure for environmental relation). We define by induction

$$
\begin{aligned}
&X_\alpha^{\mathrm{if}} = X \cup \\
&\{\ R \mid S \in X,\ R \subseteq S_\alpha^{\mathrm{if}}\ \} \cup && \textbf{(i)}\\
&\{\ (R, M, M', s) \mid S \in X,\ R \subseteq S_\alpha^{\mathrm{if}},\ (M, M', s) \in S_\alpha^{\mathrm{if}}[\,]\ \} \cup && \textbf{(ii)}\\
&\{\ (R, E[M], E'[M'], s) \mid && \textbf{(iii)}\\
&\quad (S, M, M', t) \in (X_\alpha^{\mathrm{if}})^{\rightarrow},\ R \subseteq S_\alpha^{\mathrm{if}},\ (E, E', s) \in S_\alpha^{\mathrm{if}}[t]\ \} \cup \\
&\{\ (R,\ E[\mathtt{protect}_l(M)],\ E'[\mathtt{protect}_{l'}(M')],\ s) \mid && \textbf{(iv)}\\
&\quad (U_l, U'_{l'}, \mathtt{bool}_{l''}) \in S^{\mathrm{up}},\ U \neq U',\ \mathit{conf}(l'') \not\leq \mathit{conf}(\alpha),\\
&\quad (S, M, \_, t), (S, \_, M', t) \in (X_\alpha^{\mathrm{if}})^{\rightarrow},\\
&\quad R \subseteq S_\alpha^{\mathrm{if}},\ (E, E', s) \in S_\alpha^{\mathrm{if}}[t \sqcup l'']\ \} \cup \\
&\{\ (R,\ E[\mathtt{protect}_{k \sqcup l}(M)],\ E'[\mathtt{protect}_{k' \sqcup l'}(M')],\ s) \mid && \textbf{(v)}\\
&\quad (U_l, U'_{l'}, \mathtt{bool}_{l''}) \in S^{\mathrm{up}},\ U \neq U',\ \mathit{conf}(l'') \not\leq \mathit{conf}(\alpha),\\
&\quad (S, \mathtt{protect}_k(M), \_, t), (S, \_, \mathtt{protect}_{k'}(M'), t) \in (X_\alpha^{\mathrm{if}})^{\rightarrow},\\
&\quad R \subseteq S_\alpha^{\mathrm{if}},\ (E, E', s) \in S_\alpha^{\mathrm{if}}[t \sqcup l'']\ \}
\end{aligned}
$$

where

$$
\begin{aligned}
X^{\rightarrow} \quad = \quad & \{(R, M, M', s) \mid \vdash M : s,\ \vdash M' : s,\\
& \qquad M \twoheadrightarrow N,\ M' \twoheadrightarrow N',\ (R, N, N', s) \in X\}\\
\cup \quad & \{(R, M, M', s) \mid \vdash M : s,\ \vdash M' : s,\\
& \qquad M \not\twoheadrightarrow V,\ M' \not\twoheadrightarrow V' \text{ for any } V, V'\}\\
\cup \quad & \{(R, M, M', s) \mid \vdash M : s,\ \vdash M' : s,\\
& \qquad M \twoheadrightarrow V,\ M' \twoheadrightarrow V',\ R \cup \{(V, V', s)\} \in X\}
\end{aligned}
$$

and $S_\alpha^{\mathrm{if}}[\,]$ (resp. $S_\alpha^{\mathrm{if}}[t]$) is defined by replacing D (and omitting the outer upward closure) with a term context C (resp. an evaluation context with a hole of type t) in Definition 6.8.

The rationales for each case are as follows:

- Case (**i**) takes an arbitrary subset R of the high-branch closure S_α^{if} of each environment S in X. Taking a subset—called "up-to environment" in [35]—is convenient (or, in fact, even necessary for the congruence proof later) as the conditions of bisimulation add elements only one-by-one to the environments.
- Case (**ii**) takes (besides up-to environment) terms M and M' (not necessarily values) of type s from S_α^{if} and "runs" them on the left- and right-hand sides of the configuration (R, M, M', s), as required for functions in the proof of congruence of bisimulations.
- Case (**iii**) inductively takes terms M and M' from $(X_\alpha^{\mathrm{if}})^{\rightarrow}$ (and evaluation contexts E and E' from S_α^{if}), as required for the condition on reductions.
- Case (**iv**) is similar to Definition 6.8 but M and M' are (separately) taken from $(X_\alpha^{\mathrm{if}})^{\rightarrow}$ as in case (**iii**).
- Case (**v**) is a variation of case (**iv**) for condition 2e, where the terms are already (somewhat) protected.

Note that $X \subseteq X_\alpha^{\mathrm{if}}$ by definition, hence $X_\alpha^{\mathrm{if}} \subseteq (X_\alpha^{\mathrm{if}})_\alpha^{\mathrm{if}} \subseteq ((X_\alpha^{\mathrm{if}})_\alpha^{\mathrm{if}})_\alpha^{\mathrm{if}} \subseteq \cdots = X_\alpha^{\mathrm{if}}$ thanks to the monotonicity of X_α^{if} for X and by a standard argument on inductive definition. Note also that $X \subseteq X^{\rightarrow}$.

We now define our environmental bisimulation. For brevity, we ab initio introduce (weak) environmental bisimulations up to context, up to reduction (see [40] for separate expositions of these up-to techniques for environmental bisimulations), and, in particular, with our key treatment of high-secrecy branches, calling the whole result just "environmental bisimulations."

Definition 6.11 (environmental bisimulation). An environmental relation X is an *α-environmental simulation* if it satisfies all of the following conditions:

(1) Every $(R, M, M', s) \in X$ satisfies $(R, M, M', s) \in \mathit{step}(X_\alpha^{\mathrm{if}})$ where

$$
\begin{aligned}
&\mathit{step}(Y) =\\
&\quad \{(R, M, M', s) \mid \vdash M : s,\ \vdash M' : s,\\
&\qquad M \rightarrow N_1 \twoheadrightarrow N,\ M' \twoheadrightarrow N',\ (R, N, N', s) \in Y\} \cup\\
&\quad \{(R, M, M', s) \mid \vdash M : s,\ \vdash M' : s,\\
&\qquad M \twoheadrightarrow V \text{ implies } M' \twoheadrightarrow V' \text{ and } R \cup \{(V, V', s)\} \in Y\}.
\end{aligned}
$$

(2) Every $R \in X$ satisfies:
   (a) Every $(U_l, U'_{l'}, \mathtt{bool}_{l''}) \in R$ with $\mathit{conf}(l'') \leq \mathit{conf}(\alpha)$ satisfies $U = U'$.
   (b) Every $(i_l, i'_{l'}, \mathtt{int}_{l''}), (j_k, j'_{k'}, \mathtt{int}_{k''}) \in R_\alpha^{\mathrm{if}}$ satisfies $(U_{l \sqcup k \sqcup (\mathsf{L}, e)}, U'_{l' \sqcup k' \sqcup (\mathsf{L}, e)}, \mathtt{int}_{l'' \sqcup k'' \sqcup (\mathsf{L}, e)}) \in R_\alpha^{\mathrm{if}}$ for any $e \preceq \mathit{integ}(\alpha)$, $U = \mathit{op}(i, j)$, and $U' = \mathit{op}(i', j')$.
   (c) Every $(U_l, U'_{l'}, \tau_{l''}) \in R$ with $\mathit{conf}(l'') \leq \mathit{conf}(\alpha)$ satisfies $(U_{(c, \mathit{integ}(l))}, U'_{(c, \mathit{integ}(l'))}, \tau_{(c, \mathit{integ}(l''))}) \in R_\alpha^{\mathrm{if}}$ for any $c \preceq \mathit{conf}(l'')$.
   (d) Every $(\langle V_1, V_2\rangle_l, \langle V'_1, V'_2\rangle_{l'}, (s_1 \times s_2)_{l''}) \in R$ satisfies $(V_i \sqcup l, V'_i \sqcup l', s_i \sqcup l'') \in R_\alpha^{\mathrm{if}}$ for $i \in \{1, 2\}$.
   (e) Every $((\lambda x.\, M)_l, (\lambda x.\, M')_{l'}, (s_1 \rightarrow s_2)_{l''}) \in R$ satisfies $(R, \mathtt{protect}_l([W/x]M), \mathtt{protect}_{l'}([W'/x]M'), s_2) \in (X_\alpha^{\mathrm{if}})^{\rightarrow}$ for any $(W, W', s_1) \in R_\alpha^{\mathrm{if}}$.

X is an *α-environmental bisimulation* if both X and

$$X^{-1} = \{(R, M', M, s) \mid (R, M, M', s) \in X\} \cup \{R^{-1} \mid R \in X\}$$

where $R^{-1} = \{(V', V, s) \mid (V, V', s) \in R\}$ are α-environmental simulations.

The intuitions are as follows:

- Condition 1 is standard and requires the relation is preserved by reduction or evaluation (reduction to values).
- Condition 2a says Boolean values that are visible to the attacker $conf(l'') \preceq conf(\alpha)$ must be equal on the left- and right-hand sides.
- Condition 2b requires integers—visible or not—to be preserved by operations on them, though the result is of lower (or equal) integrity than the context itself. Assuming an operation like zero-test on one of the arguments, this, along with Condition 2a, also requires the equality of visible integers.
- Condition 2c accounts for declassification of (visible) values.
- Condition 2d projects elements of pairs.
- Condition 2e applies functions to arguments that can be composed R^{if}_α from the attacker's knowledge R.

We now show the core lemma of our development:

Lemma 6.12 (congruence of environmental (bi)simulation). *If X is an α-environmental simulation, then so is X^{if}_α.*

Proof. By checking each condition of Definition 6.11 for every element of X^{if}_α by case analysis according to Definition 6.10.

The important cases proceed as follows:

Case (**i**) $R \in X^{\mathrm{if}}_\alpha$ *with* $S \in X$ *and* $R \subseteq S^{\mathrm{if}}_\alpha$. We check conditions 2a–2e. Consider the most complex one: condition 2e. Take $(V, V', (s_1 \to s_2)_{l''}) \in R \subseteq S^{\mathrm{if}}_\alpha$ with $l'' \preceq \alpha$ and $S \in X$. By Definition 6.8, $S^{\mathrm{if}}_\alpha = Q^{\mathrm{up}}$ for

$$\begin{array}{l} Q = \{\ (\ [\overline{V}, \overline{W_1}/\overline{x}, \overline{y}]D,\ [\overline{V'}, \overline{W_1'}/\overline{x}, \overline{y}]D,\ s\ )\ \mid \\ \overline{x} : \overline{s}, \overline{y} : \overline{t_1} \vdash_\alpha D : s,\ (\overline{V}, \overline{V'}, \overline{s}) \in S^{\mathrm{up}}, \\ (\overline{U_l}, \overline{U'_{l'}}, \mathtt{bool}_{\overline{l''}}) \in S^{\mathrm{up}},\ \overline{U} \neq \overline{U'},\ conf(\overline{l''}) \not\preceq conf(\alpha), \\ (\overline{W}, \_, \overline{t}),\ (\_, \overline{W'}, \overline{t}) \in S^{\mathrm{ctx}}_\alpha, \\ (\overline{W_1}, \overline{W_1'}, \overline{t_1}) \succeq (\ \overline{W} \sqcup \overline{l},\ \overline{W'} \sqcup \overline{l'},\ \overline{t} \sqcup \overline{l''}\ )\ \}. \end{array}$$

We then have $(V, V', (s_1 \to s_2)_{l''}) = (\ [\overline{V}, \overline{W_1}/\overline{x}, \overline{y}]D,\ [\overline{V'}, \overline{W_1'}/\overline{x}, \overline{y}]D,\ s\ )$ with the corresponding conditions. Since V and V' are values of a function type—namely λ-abstractions—either D itself is of the form $(\lambda x : s_1.\, C_0)_l$ with $l \preceq l''$, or else D is a variable x_i or y_j. In the former case, we have $VW \to \mathtt{protect}_l([\overline{V'}, W'/\overline{x}, x]C_0)$ and $V'W' \to \mathtt{protect}_l([\overline{V'}, W'/\overline{x}, x]C_0)$, so we are done with case (**ii**) for $(R,\ [\overline{V}, W/\overline{x}, x]\mathtt{protect}_l(C_0),\ [\overline{V'}, W'/\overline{x}, x]\mathtt{protect}_l(C_0),\ s_2 \sqcup l'') \in X^{\mathrm{if}}_\alpha$. In the latter case, if $D = x_i$, then $(V, V', (s_1 \to s_2)_{l''}) = (V_i, V_i', s_i) \in S^{\mathrm{up}}$, so the condition required for V and V' follows from the corresponding condition for V_i and V_i', with $R^{\mathrm{if}}_\alpha \subseteq (S^{\mathrm{if}}_\alpha)^{\mathrm{if}}_\alpha = S^{\mathrm{if}}_\alpha$ by Lemma 6.9 for constructing the function arguments W and W'. On the other hand, if $D = y_j$,[7] we have $W_{1j} \succeq W_j \sqcup \ell_j$ and $W'_{1j} \succeq W'_j \sqcup \ell'_j$ with $(W_j, \_, t_j) \in S^{\mathrm{ctx}}_\alpha$ and $(\_, W'_j, t_j) \in S^{\mathrm{ctx}}_\alpha$, so $(\mathtt{protect}_k([W/x]M), \_, s_2) \in (X^{\mathrm{if}}_\alpha)^{\to}$ and $(\_, \mathtt{protect}_{k'}([W'/x]M'), s_2) \in (X^{\mathrm{if}}_\alpha)^{\to}$ by a similar (albeit separate for W_j and W'_j) analysis on the context of S^{ctx}_α as in D for the previous case, hence, by case (**v**),[8] $(\mathtt{protect}_{k \sqcup l}([W/x]M), \mathtt{protect}_{k' \sqcup l'}([W'/x]M'), s_2) \in (X^{\mathrm{if}}_\alpha)^{\mathrm{if}}_\alpha \subseteq ((X^{\mathrm{if}}_\alpha)^{\mathrm{if}}_\alpha)^{\to}$ as required.

Case (**ii**). We check condition 1 by analyzing the term context C in Definition 6.10 for $S^{\mathrm{if}}_\alpha[\,]$. If C is not of the form $E[z]$ for any $z \in \{\overline{x}, \overline{y}\}$, then C itself reduces and we are done with the same case (**ii**). So suppose $C = E[z]$ for some E and $z \in \{\overline{x}, \overline{y}\}$.

If E is of the form $E_0[\mathtt{protect}_l([\,])]$ or $E_0[\mathtt{declassify}_c([\,])]$, then the required condition follows from Definition 6.4 (of upward closure) or condition 2c (for declassification), respectively.

Otherwise consider, for example, the subcase where the type of z in $\overline{x} : \overline{s}, \overline{y} : \overline{t}_1$ is a function type. If $C \neq E_0[zC_0]$, then again C itself reduces. Otherwise, $C = E_0[zC_0]$ and we follow a similar analysis as in case (**i**), albeit with the evaluation context E_0 and ending up with case (**iii**).

Another interesting and new case is when the type is $\mathtt{bool}_{l''}$ with $conf(l'') \not\preceq conf(\alpha)$ and $C = E_0[\mathtt{if}\ z\ \mathtt{then}\ C_0\ \mathtt{else}\ C'_0]$, which is covered by case (**iv**).

Cases (**iii**), (**iv**), *and* (**v**). Condition 1 follows by induction on the definition of X^{if}_α. □

Then the soundness (and completeness) of our environmental bisimulation follows:

Theorem 6.13 (soundness and completeness of environmental bisimulation w.r.t. contextual equivalence). *V and V' are α-contextually equivalent at type s if and only if $(V, V', s) \in R \in X$ for some R and α-environmental bisimulation X.*

Proof. The "if" direction (soundness) is an easy corollary of Lemma 6.12, just as congruence—namely, preservation of a relation under contexts—of bisimulations generally implies soundness as a corollary on a special case. For "only if" (completeness), take $X = \{S\}^{\mathrm{if}}_\alpha$ for

$$S = \{(\overline{V}, \overline{V}', \overline{s}) \mid \overline{V} \text{ and } \overline{V}' \text{ are } \alpha\text{-contextually equivalent at type } \overline{s}\}$$

and show the conditions of α-environmental bisimulation for each element of X by induction on the definition of $\{S\}^{\mathrm{if}}_\alpha$, constructing contexts to utilize the α-contextual equivalence of $\overline{V}$ and $\overline{V}'$.

We first check conditions 2a–2e in Definition 6.11 for each $R \in \{S\}^{\mathrm{if}}_\alpha$. Consider Definition 6.10 for $\{S\}^{\mathrm{if}}_\alpha$.

- As for the first line (before case (**i**)), R comes from $\{S\}$, that is, $R = S$ itself. Then condition 2a is immediate from Definition 5.2 for an empty context $C = x_1$. Conditions 2b–2e respectively follow from contexts $C = op_e(x_1, x_2)$, $\mathtt{declassify}_c(x_1)$, $\#_i(x_1)$, and x_1C_0 where C_0 constructs the function arguments W and W' using high-secrecy branches as required from Definition 6.8 for R^{ctx}_α.
- For case (**i**), we again construct contexts so that they emulate Definition 6.8 for S^{ctx}_α using high-secrecy branches and then check each of conditions 2a–2e as in the previous case.
- Cases (**ii**)–(**v**) do not apply for conditions 2a–2e.

[7] new case due to high-branch closure

[8] In fact, the case (**v**) is introduced into Definition 6.10 for this purpose, that is, for contexts like $(\mathtt{if}\ b_{\mathrm{H}}\ \mathtt{then}\ f\ \mathtt{else}\ g)x$.

We then check condition 1 of Definition 6.11—using only "big-step evaluation," that is, the second subset of $step(Y)$—for cases (**iii**)–(**v**) in Definition 6.10. Again, it is straightforward to construct contexts that include each (evaluation) context in these conditions, "emulating" high-branch closures by actual high-secrecy branches. Where $\{S\}^{\mathrm{if}}_{\alpha}$ is inductively defined, we, naturally, use the inductive hypothesis—that $\{S\}^{\mathrm{if}}_{\alpha}$ satisfies the conditions of Definition 6.11. □

7 Examples

Example 7.1. The pairs

$$p \quad = \quad \langle\, i,\ \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(\mathtt{if}\ x < 42\ \mathtt{then}\ 1\ \mathtt{else}\ 0)\,\rangle$$

and

$$p' \quad = \quad \langle\, j,\ \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(\mathtt{if}\ x < 42\ \mathtt{then}\ 1\ \mathtt{else}\ 0)\,\rangle$$

are LL-contextually equivalent at type

$$s = \mathtt{int}_{\mathsf{HH}} \times_{\mathsf{LH}} (\mathtt{int}_{\mathsf{HH}} \to_{\mathsf{LH}} \mathtt{int}_{\mathsf{LH}})$$

for any integers i and j with $(i < 42) \iff (j < 42)$. To show that, consider $X = \{R\}$ where:

$$\begin{aligned} R \quad = \quad & \{\, (p_{\mathsf{LH}}, p'_{\mathsf{LH}}, s),\ (i_{\mathsf{HH}}, j_{\mathsf{HH}}, \mathtt{int}_{\mathsf{HH}}), \\ & ((\lambda x.\, \mathtt{declassify}_{\mathsf{L}}(\mathtt{if}\ x < 42\ \mathtt{then}\ 1\ \mathtt{else}\ 0))_{\mathsf{LH}}, \\ & \ (\lambda x.\, \mathtt{declassify}_{\mathsf{L}}(\mathtt{if}\ x < 42\ \mathtt{then}\ 1\ \mathtt{else}\ 0))_{\mathsf{LH}}, \\ & \ \mathtt{int}_{\mathsf{HH}} \to_{\mathsf{LH}} \mathtt{int}_{\mathsf{LH}}), \\ & (1_{\mathsf{LH}}, 1_{\mathsf{LH}}, \mathtt{int}_{\mathsf{LH}}),\ (0_{\mathsf{LH}}, 0_{\mathsf{LH}}, \mathtt{int}_{\mathsf{LH}}),\ (b_{\mathsf{HL}}, b'_{\mathsf{HL}}, \mathtt{bool}_{\mathsf{HL}}) \\ & \mid b, b' \in \{\mathtt{true}, \mathtt{false}\}\ \} \end{aligned}$$

To see X is an LL-environmental bisimulation, we check the conditions in Definition 6.11, most of which are trivial in this case. A little caution is needed for condition 2e, where the HH arguments W and W' for x can be 1_{HH} and 1_{HH} or 0_{HH} and 0_{HH}, in addition to i_{HH} and j_{HH}, because of upward closure. Note that, for the same reason,

$$\langle\, 0,\ \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(\mathtt{if}\ x < 42\ \mathtt{then}\ 100\ \mathtt{else}\ 0)\,\rangle$$

and

$$\langle\, 50,\ \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(\mathtt{if}\ x > 42\ \mathtt{then}\ 100\ \mathtt{else}\ 0)\,\rangle$$

are *not* LL-contextually equivalent at the same type s, which could be overlooked without such careful consideration as the environmental bisimulation proof above.

Note also that, in the above R, we may not need *both* $(1_{\mathsf{LH}}, 1_{\mathsf{LH}}, \mathtt{int}_{\mathsf{LH}})$ and $(0_{\mathsf{LH}}, 0_{\mathsf{LH}}, \mathtt{int}_{\mathsf{LH}})$ if $i, j < 42$, but having them in the same R does no "harm" and is convenient in fact. Such "larger than necessary" R is common in (environmental) bisimulations. Meanwhile, if $i, j \geq 42$, we must consider the return values 0 and 0 at type $\mathtt{int}_{\mathsf{LH}}$, which can be upgraded to $\mathtt{int}_{\mathsf{HH}}$ and then given to the same function, yielding 1 and 1 as well.

Moreover, comparison of the high-secrecy integers i and j by contexts such as $[\,] = c$ where c is some integer constant which may happen to be i or j, may result in possibly different high-secrecy Booleans b and b' if $i \neq j$, though further consequences of this difference can be absorbed thanks to high-branch closures, and does not break the equivalence thanks to their low integrity. This inclusion of b and b' in X is not necessary if $i = j$, but again "does no harm" even in the latter scenario.

Consequently, it is often the case that X is a singleton set because one big environment R suffices, as in the present example.

Example 7.2. Continuing the previous example, $g = \lambda f.\, f\, p$ and $g' = \lambda f.\, f\, p'$ are LL-contextually equivalent at type

$$t \quad = \quad (s \to_{\mathsf{LL}} \mathtt{bool}_{\mathsf{LL}}) \to_{\mathsf{LL}} \mathtt{bool}_{\mathsf{LL}}$$

again provided that $(i < 42) \iff (j < 42)$. In fact, we can show this equivalence by the same X as above, since $(g, g', t) \in R^{\mathrm{ctx}}_{\mathsf{LL}}\ (\subseteq R^{\mathrm{if}}_{\mathsf{LL}})$ for $D = \lambda f.\, f\, x_1$. This trick does not work if g and g' themselves are of high integrity (so the low-level attacker cannot compose the common context D) like

$$t \quad = \quad (s \to_{\mathsf{LL}} \mathtt{bool}_{\mathsf{LL}}) \to_{\mathsf{LH}} \mathtt{bool}_{\mathsf{LL}}$$

in which case we apply them to arguments W and W' from $R^{\mathrm{if}}_{\mathsf{LL}}$ to obtain $W\, p$ and $W\, p'$, which are again in $R^{\mathrm{if}}_{\mathsf{LL}}$ so we are done.

Example 7.3. The pairs of functions

$$q \quad = \quad \langle\, \lambda x.\, \langle x, x\rangle,\ \lambda p.\, \mathtt{declassify}_{\mathsf{L}}(\#_1(p) + \#_2(p))\,\rangle$$

and

$$q' \quad = \quad \langle\, \lambda x.\, \langle x+1, x-1\rangle,\ \lambda p.\, \mathtt{declassify}_{\mathsf{L}}(\#_1(p) + \#_2(p))\,\rangle$$

are LL-contextually equivalent at type

$$s_1 \quad = \quad (\mathtt{int}_{\mathsf{LL}} \to_{\mathsf{LH}} s_0) \times_{\mathsf{LH}} (s_0 \to_{\mathsf{LH}} \mathtt{int}_{\mathsf{LL}})$$

where $s_0 = \mathtt{int}_{\mathsf{LL}} \times_{\mathsf{HH}} \mathtt{int}_{\mathsf{LL}}$. The point here is that the integer x is of low security (both low-secrecy and low-integrity) but the pairs $\langle x, x\rangle$ and $\langle x+1, x-1\rangle$ are HH, for which the only applicable function $\lambda p.\, \mathtt{declassify}_{\mathsf{L}}(\#_1(p) + \#_2(p))$ returns the same low-secrecy integer $2x$. Note that q and q' cannot be equivalent if the pair type s_0 were $\mathtt{int}_{\mathsf{HL}} \times_{\mathsf{LH}} \mathtt{int}_{\mathsf{HL}}$ and x were of type $\mathtt{int}_{\mathsf{HL}}$, as attackers could then create arbitrary pairs of type s_0 by freely recomposing them.

To see the equivalence of q and q' at type s_1, take $X = \{R\}$ where:

$$\begin{aligned} R \quad = \quad & \{\, (\ q,\ q',\ s_1\ ), \\ & (\ \lambda x.\, \langle x, x\rangle,\ \lambda x.\, \langle x+1, x-1\rangle,\ \mathtt{int}_{\mathsf{LL}} \to_{\mathsf{LH}} s_0\ ), \\ & (\ \lambda p.\, \mathtt{declassify}_{\mathsf{L}}(\#_1(p) + \#_2(p)), \\ & \ \ \lambda p.\, \mathtt{declassify}_{\mathsf{L}}(\#_1(p) + \#_2(p)), \\ & \ \ s_0 \to_{\mathsf{LH}} \mathtt{int}_{\mathsf{LL}}\ ), \\ & (\ \langle i, i\rangle,\ \langle i+1, i-1\rangle,\ s_0\ ),\ (\ b,\ b',\ \mathtt{bool}_{\mathsf{HL}}\ ) \\ & \mid i \in \mathbf{Int},\ b, b' \in \{\mathtt{true}, \mathtt{false}\}\ \} \end{aligned}$$

Checking the conditions of LL-environmental bisimulation is straightforward by construction. In particular, it is crucial that attackers (contexts) cannot create high-integrity pairs (of type s_0) of arbitrary integers.

However, comparison of the first and second elements of the high-security pairs $\langle i, i\rangle$ and $\langle i+1, i-1\rangle$ by contexts such as

$$\mathtt{let}\ r = [\,]\ \mathtt{in}\ \mathtt{let}\ f = \#_1(r)\ \mathtt{in}\ \#_1(f\ 42) = \#_2(f\ 42)$$

for q and q', yields different high-secrecy Booleans, though further consequences of this difference can be absorbed thanks to high-branch closures.

Changing the pair type s_0 from $\mathtt{int}_{\mathsf{LL}} \times_{\mathsf{HH}} \mathtt{int}_{\mathsf{LL}}$ to $\mathtt{int}_{\mathsf{HL}} \times_{\mathsf{HH}} \mathtt{int}_{\mathsf{HL}}$ by itself makes no difference in this example, thanks to the high integrity. However, changing also the type $\mathtt{int}_{\mathsf{LL}}$ of the integer parameter x to $\mathtt{int}_{\mathsf{HL}}$ breaks the equivalence for contexts like:

$$\begin{aligned} & \mathtt{let}\ r = [\,]\ \mathtt{in}\ \mathtt{let}\ f = \#_1(r)\ \mathtt{in}\ \mathtt{let}\ g = \#_2(r)\ \mathtt{in} \\ & g(f(\mathtt{if}\ \#_1(f\ 42) = \#_2(f\ 42)\ \mathtt{then}\ 1\ \mathtt{else}\ 0)) \end{aligned}$$

It may be surprising that changing the secrecy *from low to high* leads to *more* information leakage in such an indirect way via a high-secrecy conditional branch, even though it can be explained abstractly as an instance of general contravariance of the argument part of function types.

Note also that without the first change, the second would be ill-typed and thereby prevented, trying to "downcast" $\mathtt{int}_{\mathsf{HL}}$ to $\mathtt{int}_{\mathsf{LL}}$ with no explicit declassification (while the first change without the second performs an "upcast," which is well-typed by T-Sub and S-Label).

Example 7.4. This example essentializes the point above on information leakage caused by changing low secrecy to high: the pairs

$$q \quad = \quad \langle\, \mathtt{true}_{\mathsf{H}\_},\ \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(x) \,\rangle$$

and

$$q' \quad = \quad \langle\, \mathtt{false}_{\mathsf{H}\_},\ \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(x) \,\rangle$$

(where _ means "don't-care," i.e., either H or L will do as long as all terms are well-typed) are LL-contextually equivalent at type

$$s \quad = \quad \mathtt{bool}_{\mathsf{H}\_} \times (\mathtt{int}_{\mathsf{LL}} \to \mathtt{int}_{\mathsf{LL}})$$

(where the "declassification" does nothing for the moment) with $X = \{R\}$ and

$$\begin{aligned} R \quad = \quad & \{\, (\, q,\ q',\ s\,),\ (\, \mathtt{true}_{\mathsf{H}\_},\ \mathtt{false}_{\mathsf{H}\_},\ \mathtt{bool}_{\mathsf{H}\_}\,), \\ & (\, \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(x), \\ & \ \lambda x.\, \mathtt{declassify}_{\mathsf{L}}(x), \\ & \ \mathtt{int}_{\mathsf{LL}} \to \mathtt{int}_{\mathsf{LL}}\,)\,\}. \end{aligned}$$

However, they are *in*equivalent at another type $\mathtt{bool}_{\mathsf{H}\_} \times (\mathtt{int}_{\mathsf{HL}} \to \mathtt{int}_{\mathsf{LL}})$ for contexts such as

$$\mathtt{let}\ p = [\,]\ \mathtt{in}\ \#_2(p)(\mathtt{if}\ \#_1(p)\ \mathtt{then}\ 1\ \mathtt{else}\ 0).$$

Again, "upgrading" $\mathtt{int}_{\mathsf{LL}}$ to $\mathtt{int}_{\mathsf{HL}}$ broke the equivalence via a conditional branch over high-secrecy Booleans. This time, however, there would be less surprise, given such obvious information leakage via declassification.

Further changing $\mathtt{int}_{\mathsf{HL}}$ to $\mathtt{int}_{\mathsf{HH}}$ prevents the leak since the context cannot create a value of high integrity, showing the role of integrity even for ensuring secrecy only.

Technically, such subtle but crucial differences are captured by condition 2e of Definition 6.11, where the function arguments W and W' are composed according to Definition 6.8.

8 Conclusion

We have developed a theory of environmental bisimulation for directly proving noninterference as equivalences of security-typed λ-terms with declassification, and presented examples including non-trivial interaction between security types, declassification, and high-secrecy conditional branches.

An obvious possible extension would be *endorsement* for integrity, often considered the dual of secrecy and declassification.

For the sake of simpler exposition, our present language does not have recursion and is therefore strongly normalizing. It is fundamentally straightforward (though notationally tedious) to extend it with, for instance, a fixed-point operator with appropriate security levels [21]. It is also straightforward to make our theory termination-*in*sensitive as in traditional security type systems for imperative languages (e.g., [47, Theorem 6.8]), by adapting Definition 5.2 as "…when: $[\overline{V}/\overline{x}]C \twoheadrightarrow b_k$ and $[\overline{V'}/\overline{x}]C \twoheadrightarrow b'_{k'}$ imply $b = b'$ for any b_k, $b'_{k'}$, and…" and the last lines of $X^{\rightarrow}$ in Definition 6.10 and $\mathit{step}(Y)$ in Definition 6.11 as "$M \twoheadrightarrow V$ and $M' \twoheadrightarrow V'$ imply $R \cup \{(V, V', s)\} \in \cdots$," with most of the proofs unchanged—only small portions that use these conditions need to be adapted accordingly (in fact, almost automatically).

Since our language did not have state or generative names [12, 24], the environmental bisimulations could probably be reduced to *logical bisimulations* [12, 34]; indeed, all the X's in Section 7 were singletons! The sole element R of X can "easily" be constructed "just" by including all (possibly infinite—recall the problem is undecidable in general) pairs that can be reached from the initial programs according to each condition (which represents an attacker's operation) of environmental bisimulations. We nevertheless adopted the more general, environmental bisimulations since they are arguably not much harder from a technical viewpoint, and naturally scale to languages with state [14, 33, 35, 40] and even concurrency [33, 35, 38].

We guaranteed security of programs directly as noninterference (with suitable conditions) via environmental bisimulations, while the security type system was necessary for constraining the attackers as well as programs' interactions with them, e.g. to prevent a program from applying an attacker's function of type $\mathtt{int}^{\mathsf{L}} \to \ldots$ to an argument of type $\mathtt{int}^{\mathsf{H}}$. Pushing this approach to an extreme, it may be possible to allow even "internally untyped" programs. We leave this as interesting future work.

One might wonder what is so special about Booleans in λ-calculus, where they can anyway be encoded into mere functions in the style of Church. Consider the standard Church encoding of $\mathtt{true}_{\mathsf{H}}$ and $\mathtt{false}_{\mathsf{H}}$, that is, $\mathit{ctrue}_{\mathsf{H}} = (\lambda x.\, \lambda y.\, x)_{\mathsf{H}}$ and $\mathit{cfalse}_{\mathsf{H}} = (\lambda x.\, \lambda y.\, y)_{\mathsf{H}}$. Applying each of them to any known values V and V' yields, of course, V_{H} and V'_{H}, respectively. It might be interesting to think of a less ad hoc method to handle them without having to add all such pairs $(V_{\mathsf{H}}, V'_{\mathsf{H}})$ into the environment.

Practically, it is natural to consider (necessarily incomplete—since equivalence is undecidable—when the language is extended with recursion, but as useful as possible) automation, or mechanization with proof assistants, of our environmental bisimulation proofs, on both of which work is in progress. In the former, it would help to introduce some symbolic representations (e.g. as in open or normal form bisimulations [15, 32]) of the function arguments W and W' taken from the infinite sets R^{if}_{α} in condition 2e of Definition 6.11.

Acknowledgments

This work was supported by the Acquisition, Technology & Logistics Agency (ATLA), Japan, under the Innovative Science and Technology Initiative for Security program, Grant Number JPJ013268 and in part by JSPS KAKENHI Grant Numbers 22K19766 and 23K20379 (20H04161).

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 147–160. doi:10.1145/292540.292555

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 147–160. doi:10.1145/292540.292555

[3] Martín Abadi and Phillip Rogaway. 2002. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). *J. Cryptol.* 15, 2 (2002), 103–127. doi:10.1007/S00145-001-0014-7

[4] Samson Abramsky and C.-H. Luke Ong. 1993. Full Abstraction in the Lazy Lambda Calculus. *Inf. Comput.* 105, 2 (1993), 159–267. doi:10.1006/INCO.1993.1044

[5] Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, Stephen Chong and David A. Naumann (Eds.). ACM, 113–124. doi:10.1145/1554339.1554353

[6] Raimil Cruz, Tamara Rezk, Bernard P. Serpette, and Éric Tanter. 2017. Type Abstraction for Relaxed Noninterference. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:27. doi:10.4230/LIPICS.ECOOP.2017.7

[7] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243. doi:10.1145/360051.360056

[8] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory* 29, 2 (1983), 198–207. doi:10.1109/TIT.1983.1056650

[9] Takeshi Egawa. 2020. *Verification and Improvement of Environmental Bisimulation for Security-Typed λ-calculus.* Master's thesis. Graduate School of Information Sciences, Tohoku University. In Japanese.

[10] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 11–20. doi:10.1109/SP.1982.10014

[11] Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 365–377. doi:10.1145/268946.268976

[12] Vasileios Koutavas, Paul Blain Levy, and Eijiro Sumii. 2011. From Applicative to Environmental Bisimulation. In *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011 (Electronic Notes in Theoretical Computer Science, Vol. 276)*, Michael W. Mislove and Joël Ouaknine (Eds.). Elsevier, 215–235. doi:10.1016/J.ENTCS.2011.09.023

[13] Vasileios Koutavas and Mitchell Wand. 2006. Bisimulations for Untyped Imperative Objects. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 146–161. doi:10.1007/11693024_11

[14] Vasileios Koutavas and Mitchell Wand. 2006. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 141–152. doi:10.1145/1111037.1111050

[15] Søren B. Lassen. 2005. Eager Normal Form Bisimulation. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 345–354. doi:10.1109/LICS.2005.15

[16] Peng Li and Steve Zdancewic. 2005. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 158–170. doi:10.1145/1040305.1040319

[17] Jan Menz, Andrew K. Hirsch, Peixuan Li, and Deepak Garg. 2023. Compositional Security Definitions for Higher-Order Where Declassification. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 406–433. doi:10.1145/3586041

[18] John C. Mitchell. 1996. *Foundations for Programming Languages.* MIT Press.

[19] James H. Morris, Jr. 1968. *Lambda-Calculus Models of Programming Languages.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[20] Minh Ngo, David A. Naumann, and Tamara Rezk. 2020. Type-Based Declassification for Free. In *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12531)*, Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony (Eds.). Springer, 181–197. doi:10.1007/978-3-030-63406-3_11

[21] Lin Oshitani. 2018. *Environmental Bisimulation for Information Flow Analysis.* Master's thesis. Graduate School of Information Sciences, Tohoku University. In Japanese.

[22] Adrien Piérard and Eijiro Sumii. 2011. Sound Bisimulations for Higher-Order Distributed Process Calculus. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6604)*, Martin Hofmann (Ed.). Springer, 123–137. doi:10.1007/978-3-642-19805-2_9

[23] Adrien Piérard and Eijiro Sumii. 2012. A Higher-Order Distributed Calculus with Name Creation. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 531–540. doi:10.1109/LICS.2012.63

[24] Andrew M. Pitts and Ian David Bede Stark. 1993. Observable Properties of Higher Order Functions that Dynamically Create Local Names, or What's new?. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 711)*, Andrzej M. Borzyszkowski and Stefan Sokolowski (Eds.). Springer, 122–141. doi:10.1007/3-540-57182-5_8

[25] Gordon Plotkin. 1973. *Lambda definability and logical relations.* Technical Report SAI-RM-4. School of Artificial Intelligence, University of Edinburgh.

[26] Vineet Rajani, Alex Coleman, and Hrutvik Kanabar. 2025. A Graded Modal Approach to Relaxed Semantic Declassification. In *38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025*. IEEE, 268–283. doi:10.1109/CSF64896.2025.00032

[27] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

[28] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. doi:10.1109/JSAC.2002.806121

[29] Andrei Sabelfeld and Andrew C. Myers. 2003. A Model for Delimited Information Release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers (Lecture Notes in Computer Science, Vol. 3233)*, Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki (Eds.). Springer, 174–191. doi:10.1007/978-3-540-37621-7_9

[30] Andrei Sabelfeld and David Sands. 2001. A Per Model of Secure Information Flow in Sequential Programs. *High. Order Symb. Comput.* 14, 1 (2001), 59–91. doi:10.1023/A:1011553200337

[31] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *J. Comput. Secur.* 17, 5 (2009), 517–548. doi:10.3233/JCS-2009-0352

[32] Davide Sangiorgi. 1994. The Lazy Lambda Calculus in a Concurrency Scenario. *Inf. Comput.* 111, 1 (1994), 120–153. doi:10.1006/INCO.1994.1042

[33] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2007. Environmental Bisimulations for Higher-Order Languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 293–302. doi:10.1109/LICS.2007.17

[34] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2007. Logical Bisimulations and Functional Languages. In *International Symposium on Fundamentals of Software Engineering, International Symposium, FSEN 2007, Tehran, Iran, April 17-19, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4767)*, Farhad Arbab and Marjan Sirjani (Eds.). Springer, 364–379. doi:10.1007/978-3-540-75698-9_24

[35] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2011. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.* 33, 1 (2011), 5:1–5:69. doi:10.1145/1889997.1890002

[36] Davide Sangiorgi and Valeria Vignudelli. 2016. Environmental bisimulations for probabilistic higher-order languages. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 595–607. doi:10.1145/2837614.2837651

[37] Davide Sangiorgi and Valeria Vignudelli. 2019. Environmental Bisimulations for Probabilistic Higher-order Languages. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 22:1–22:64. doi:10.1145/3350618

[38] Nobuyuki Sato and Eijiro Sumii. 2009. The Higher-Order, Call-by-Value Applied Pi-Calculus. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 311–326. doi:10.1007/978-3-642-10672-9_22

[39] Naokata Shikuma and Atsushi Igarashi. 2008. Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus. *Log. Methods Comput. Sci.* 4, 3 (2008). doi:10.2168/LMCS-4(3:10)2008

[40] Eijiro Sumii. 2009. A Complete Characterization of Observational Equivalence in Polymorphic *lambda*-Calculus with General References. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5771)*, Erich Grädel and Reinhard Kahle (Eds.). Springer, 455–469. doi:10.1007/978-3-642-04027-6_33

[41] Eijiro Sumii and Benjamin C. Pierce. 2001. Logical Relations for Encryption. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13*

June 2001, Cape Breton, Nova Scotia, Canada. IEEE Computer Society, 256–269. doi:10.1109/CSFW.2001.930151

[42] Eijiro Sumii and Benjamin C. Pierce. 2003. Logical Relations for Encryption. *J. Comput. Secur.* 11, 4 (2003), 521–554. doi:10.3233/JCS-2003-11403

[43] Eijiro Sumii and Benjamin C. Pierce. 2004. A bisimulation for dynamic sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 161–172. doi:10.1145/964001.964015

[44] Eijiro Sumii and Benjamin C. Pierce. 2005. A bisimulation for type abstraction and recursion. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 63–74. doi:10.1145/1040305.1040311

[45] Eijiro Sumii and Benjamin C. Pierce. 2007. A bisimulation for dynamic sealing. *Theor. Comput. Sci.* 375, 1-3 (2007), 169–192. doi:10.1016/J.TCS.2006.12.032

[46] Eijiro Sumii and Benjamin C. Pierce. 2007. A bisimulation for type abstraction and recursion. *J. ACM* 54, 5 (2007), 26. doi:10.1145/1284320.1284325

[47] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. doi:10.3233/JCS-1996-42-304

[48] Steve Zdancewic. 2003. A Type System for Robust Declassification. In *Proceedings of 19th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2003, Université de Montréal, QC, Canada, March 19-22, 2003 (Electronic Notes in Theoretical Computer Science, Vol. 83)*, Stephen D. Brookes and Prakash Panangaden (Eds.). Elsevier, 263–277. doi:10.1016/S1571-0661(03)50014-7

[49] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society, 15–23. doi:10.1109/CSFW.2001.930133