Gradual Security Types and Gradual Guarantees

Abhishek Bichhawat
Carnegie Mellon University
Pittsburgh, USA
abichhaw@andrew.cmu.edu

McKenna McCall
Carnegie Mellon University
Pittsburgh, USA
mckennak@andrew.cmu.edu

Limin Jia
Carnegie Mellon University
Pittsburgh, USA
liminjia@andrew.cmu.edu

2021 IEEE 34th Computer Security Foundations Symposium (CSF) | 978-1-7281-7607-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSF51468.2021.00015

***Abstract*—Information flow type systems enforce the security property of noninterference by detecting unauthorized data flows at compile-time. However, they require precise type annotations, making them difficult to use in practice as much of the legacy infrastructure is written in untyped or dynamically-typed languages. Gradual typing seamlessly integrates static and dynamic typing, providing the best of both approaches, and has been applied to information flow control, where information flow monitors are derived from gradual security types. Prior work on gradual information flow typing uncovered tensions between noninterference and the dynamic gradual guarantee—the property that less precise security type annotations in a program should not cause more runtime errors.**

This paper re-examines the connection between gradual information flow types and information flow monitors to identify the root cause of the tension between the gradual guarantees and noninterference. We develop runtime semantics for a simple imperative language with gradual information flow types that provides both noninterference and gradual guarantees. We leverage a proof technique developed for FlowML and reduce noninterference proofs to preservation proofs.

***Index Terms*—Information flow control, noninterference, gradual typing, gradual guarantees**

I. Introduction

Information flow type systems combine types and security labels to ensure that well-typed programs do not leak secrets to attackers at compile-time [1]. However, purely statically-typed languages face significant adoption challenges. Most programmers are unfamiliar with and may be unwilling to use complex information flow type systems. Moreover, much of the legacy infrastructure is written in untyped and dynamically-typed languages without precise security type annotations.

Gradual typing is one promising technique to address these challenges [2]; it aims to seamlessly integrate statically-typed programs with dynamically-typed programs. At a high-level, gradual type systems introduce a dynamic type, often written as ?, to accommodate untyped portions of the program. The type system allows any program to be typed under ?. The type system enforces type safety on statically typed parts and the runtime semantics of gradual type systems monitor the interactions between parts typed as ? and statically typed parts to ensure type preservation.

Gradual typing has been applied to information flow types [3]–[7], where certain expressions have a dynamic security label ? (and are typed as, e.g., $\mathsf{int}^?$), which is determined at runtime. Information flow monitors are then derived from the runtime semantics of gradual information flow types. Earlier adoptions of gradual typing to information flow types have developed ad hoc approaches to treat "gradual types". For instance, Disney and Flanagan did not include a dynamic security label [3]. Instead, the programmer would insert type casts, which are checked at runtime, and are used to "gradually" make the programs more secure. Later, ML-GS [4] and LJGS [5] included the dynamic security label ? and a runtime monitor that performed checks on the dynamically typed parts of the program. Programmers still need to write annotations and casts in ML-GS and label constraints in LJGS. The dynamic label is instantiated as a single security label at run time in both ML-GS and LJGS.

At the same time, interests in the formal foundations of gradual types grew significantly. Formal properties related to gradual typing such as the *gradual guarantee* [8] were introduced, which says that loosening policies should not cause more type errors or runtime failures. Roughly, the gradual guarantee ensures that programs which type-check and run to completion with precise type annotations will also type-check and run to completion with less precise types, i.e., ?. This property ensures that programmers are not punished for not specifying type annotations if the program is safe. Such a guarantee is important for information flow type systems, as security annotations have been a road block for adoption. Without the gradual guarantee, the programmers' burden of providing (unnecessary) security annotations is increased.

Garcia et al. developed the abstracting gradual typing (AGT) framework which provides a formal interpretation of gradual type systems [9]. In AGT, a principled interpretation of the dynamic type is that it represents the set of all possible types that are refined by the monitor to preserve type safety at runtime. By that interpretation, the semantics of the dynamic information flow label ? is the set of all possible labels. Early work on gradual information flow typing all instantiate the dynamic label as a single label at runtime [4], [5]. Recent work by Toro et al., $\mathsf{GSL}_{\mathsf{Ref}}$, aims to apply the AGT framework to information flow types [6]; however, it has to give up the dynamic gradual guarantee in favor of *noninterference*, the key information flow security property [10], when dealing with mutable references.

In this paper, we re-examine the connection between gradual information flow types and information flow monitors (c.f. [11]–[13]). We aim to identify the root cause of the tension between the dynamic gradual guarantee and security in systems that refine the set of possible labels for dynamically labeled programs at runtime. To this end, we focus on a simple imperative language with first-order stores, which has been widely used to design information flow control systems [1], [13]–[17]. While simple, this language includes all the features to illustrate the problem of refining dynamic labels at runtime. We develop runtime semantics for this language with gradual information flow types that enjoy both noninterference and the dynamic gradual guarantee. We draw ideas from abstracting gradual typing, which advocates deriving runtime semantics for gradual types via the preservation proof [9].

We observe that as dynamic labels are updated, the semantics that only gradually refine the possible security labels during program execution *resemble* a naive flow-sensitive monitor and therefore inherit the problems with implicit leaks of flow-sensitive monitors [11]. To enforce noninterference and remove the implicit leaks caused by insecure writes in branches, the runtime semantics needs to take into consideration the variable and channel writes in the untaken branch [13]. Pure guessing which ignores information about the untaken branch like $\mathsf{GSL}_{\mathsf{Ref}}$'s runtime yields rigid semantics that break the gradual guarantee (more in Section III-D). The no-sensitive-upgrade (NSU) check [11] also doesn't solve the problem. Instead, a "hybrid" approach [13], [17] that leverages static analysis to obtain the write effects of the untaken branch and upgrade relevant references for both branches can be used to remove the implicit leaks and provide the gradual guarantees.

We leverage a proof technique developed for FlowML, which reduces noninterference proofs to preservation proofs [18]. The main idea is to extend the language with *pairs* of expressions and commands, representing two executions with different secrets in one program. Noninterference follows from preservation. This proof technique clearly illustrates the problem with purely dynamic flow-sensitive monitors and naturally suggests the hybrid approach [13], [17].

To summarize, we study the connection between gradual security types and information flow monitors and identify the conservative handling of implicit flows in $\mathsf{GSL}_{\mathsf{Ref}}$ as the reason that it gives up dynamic gradual guarantee in favor of noninterference. Additionally, we show that the dynamic gradual guarantee can be recovered by using a hybrid approach that leverages the static phase to generate a list of variables that are written to in both the branches. Due to space constraints, we omit detailed definitions and proofs, which can be found in the full version of the paper [19].

II. Overview of Information Flow Control

In information flow control systems, variables are annotated with a label from a security lattice, which have a partial-ordering ($\preccurlyeq$) and a well-defined join and meet operation. $\ell_1 \preccurlyeq \ell_2$ means information can flow from ℓ_1 to ℓ_2. Consider a two-point security lattice with labels $\{L, H\}$ with $L \preccurlyeq H$ where L represents public and H represents secret. A variable x having type int^H contains a sensitive integer value.

Information flows can be broadly classified as *explicit* or *implicit* [10], [20]. Explicit flows arise from variable assignments. For instance, the statement $x = y + z$ causes an explicit flow of values from y and z to x. Implicit flows arise from control structures in the program. For example, in the program $l = \mathsf{false};\ \mathsf{if}(h)\{l = \mathsf{true};\}$, there is an implicit flow of information from h to the final value of l (it is true iff h is true). Implicit flows are handled by maintaining a pc (program-context) label, which is an upper bound on the labels of all the predicates that have influenced the control flow thus far. In the example, the pc inside the branch is the label of h.

Information flow control systems aim to prevent leaks through these flows by either enforcing information flow typing rules and ruling out insecure programs at compile-time or dynamically monitoring programs and aborting the execution of insecure programs. In both systems, assignment to a variable is disallowed if either the pc label or the join of the label of the operands is not less than or equal to the label of the variable being assigned [1], [11]. Thus, in the above examples, if either the label of y or z is greater than the label of x or the label of h is greater than the label of l, the assignment does not type-check or the execution aborts at runtime. This guarantees a variant of noninterference, known as termination-insensitive noninterference [1], which we prove for our gradual type system. We assume that an adversary cannot observe or gain any information if a program's execution diverges or aborts and can only observe "public" outputs by the program.

We consider a flow-insensitive, fixed-label system in this paper and prove termination-insensitive noninterference for it.

III. Gradual Security Typing

Static information flow type systems do not scale up to scenarios where the security levels of some of the variables are not known at compile-time, while pure monitoring approaches cannot reject obviously insecure programs at compile-time. Gradual typing extends the reach of type-system based analysis by adding an imprecise (or dynamic) label, ?, for variables whose labels are not known at compile-time. The runtime semantics then ensures that no information is leaked due to the relaxation of the type-system's handling of ? labels.

A. Imprecise Security Label: Interpretations and Operations

The label ? is not an actual element of the security lattice and its meaning is not universally agreed upon. Differences will manifest in the runtime monitoring semantics and proof of noninterference. For illustration, consider a variable x of type $\mathsf{int}^?$. Semantically, in the literature ? has meant one of the following. (1) x's label is dynamic (flow-sensitive) and can change at runtime (e.g., from ? representing L to representing H when x is assigned a secret). (2) the set of possible labels for x is refined at runtime and the set in a future state will be a subset of its the current state. Since we build on a flow-insensitive type system, we opt for the second meaning of ?. Our runtime monitor will keep track of the set of possible labels for x. Note that a typical flow-sensitive IFC monitor (e.g. [11], [13]) takes an approach aligned with (1).

In the initial program state, ? could be interpreted as: (1) x could contain a secret or be observable to an adversary.

1 $y := \mathsf{false}^H$
2 $\mathsf{if}\ (x)\ \mathsf{then}\ y := \mathsf{true}^H$

Listing 1. Statically typed

1 $y := \mathsf{false}^?$
2 $\mathsf{if}\ (x)\ \mathsf{then}\ y := \mathsf{true}^?$

Listing 2. Dynamically typed

1 $y := \mathsf{false}^?$
2 $z := \mathsf{false}^L$
3 $\mathsf{if}\ (x)\ \mathsf{then}\ y := \mathsf{true}^?$
4 $\mathsf{if}\ (y)\ \mathsf{then}\ z := \mathsf{true}^L$
5 $\mathsf{output}(L,\ z)$

Listing 3. NSU

1 $y := \mathsf{true}^?$
2 $z := \mathsf{true}^L$
3 $\mathsf{if}\ (x)\ \mathsf{then}\ y := \mathsf{false}^?$
4 $\mathsf{if}\ (y)\ \mathsf{then}\ z := \mathsf{false}^L$
5 $\mathsf{output}(L,\ z)$

Listing 4. Implicit flows

Therefore, we should treat x conservatively as if it is both secret and public. (2) ? indicates indifference; the data x contains in the initial state is of no security value; otherwise, x should have been given the label H. We choose (2) again, as it is a cleaner interpretation. Note that this is only for the initial state. At runtime, the monitor maintains enough state to concretely know whether x contains a secret or not.

B. Gradually Refined Security Policy via Examples

Next, we describe how security labels can be gradually refined at runtime. Consider the program in Listing 1 and its variant with dynamic types in Listing 2. Suppose the lattice contains four elements $\{\bot, L, H, \top\}$ such that $\bot \preccurlyeq L \preccurlyeq H \preccurlyeq \top$. Assume that the initial type of x is bool^H in both examples while the type of y is bool^H in Listing 1 and $\mathsf{bool}^?$ in Listing 2[1]. The program in Listing 1 does not leak any information. With gradual typing, its variant in Listing 2 is also accepted by the type system. The runtime refines the set of possible labels for y as the program runs. With $x : \texttt{bool}^H$, y cannot be $\bot$ or L as that would result in an implicit flow. Thus, the possible labels of y are refined to the set $\{H, \top\}$ when $x = \mathsf{true}$. If, suppose, $x : \texttt{bool}^L$, then y is labeled $\{L, H, \top\}$ after executing line 2.

Suppose the program in Listing 2 is extended with another branch as shown in Listing 3 with $z : \texttt{bool}^L$. When $x : \texttt{bool}^L$, the possible labels for y on line 4 are $\{L, H, \top\}$. This allows the assignment on line 4 to succeed as the assignment is to a variable that has a label contained in the set of possible pc labels. When x has the type $\texttt{bool}^H$, then the possible labels for y on line 4 are $\{H, \top\}$. The assignment on line 4 is aborted as L is not equal or higher than any of the possible pc labels ($\{H, \top\}$). In this case, the monitor enforces NSU and prevents the implicit leak.

C. Gradual Guarantees

Desirable formal properties for gradual type systems are the *gradual guarantees*, proposed by [8]. The gradual guarantees relate programs that differ only in the precision of the type annotations. They state that changes that make the annotations of a gradually typed program less precise should not change the static or dynamic behavior of the program. In other words, if a program with more precise type annotations is well-typed in the static type system, and terminates in the runtime semantics, then the same program with less precise terms is also well-typed, and terminates, respectively.

For illustration, consider the previous example from Listing 1. Assume that the program is well-typed under a gradual type system with $x : \texttt{bool}^H$ and $y : \texttt{bool}^H$ as secret variables. When x is `true`, the branch on line 2 is taken and y is assigned the value `true` and has the label H. When x is `false`, y remains `false`. This program is accepted by the security type system and dynamic information flow monitor, and runs to completion in all possible executions. As per the *static gradual guarantee*, the program should also be well-typed if, for instance, y had an imprecise ? security level as shown in Listing 2. By the *dynamic gradual guarantee*, the program in Listing 2 should run to completion at runtime, even with the imprecise label for y in all executions of the program.

The gradual guarantees are important in the context of information flow systems to show that the gradual security type system is strictly more permissive than the static security type system and the dynamic IFC monitor, while providing the same guarantees. They mean that programmers need not worry about insufficient annotations causing their safe programs to be rejected by the type system, or worse, at runtime, which may lead to undesirable behavior. Concerning programmers with unnecessary annotations defeats the purpose of gradual typing, which is meant to alleviate the burden of annotation.

D. Implicit Flows vs. Dynamic Gradual Guarantee

The example in Listing 4 illustrates how gradual typing semantics handle implicit flows. Assume that $x : \mathsf{bool}^H$ is a secret variable while the security types of $y : \mathsf{bool}^?$ and $z : \mathsf{bool}^?$ are unknown at compile-time, and the security lattice is $\bot \preccurlyeq L \preccurlyeq H \preccurlyeq \top$. Consider two runs of the program with different initial values of x. When x is true, the branch on line 3 is taken and y is assigned the value false. With gradual typing, the labels of y are refined to $\{H, \top\}$. As y is false, the branch on line 4 is not taken and z remains true with the set of labels $\{\bot, L, H, \top\}$. As z's value is visible at L, the output on line 5 succeeds. When x is false, the branch on line 3 is not taken and y remains true with the set of labels $\{\bot, L, H, \top\}$. As the pc on line 4 contains the set of labels $\{\bot, L, H, \top\}$, the assignment on line 4 succeeds, and z becomes false while the set of labels remains $\{\bot, L, H, \top\}$. Again, as z's value is visible at L, the output on line 5 succeeds. Thus, in the two runs of the program, different values of z are output for different values of x, thereby leaking x to the adversary at level L. Here, the NSU mechanism does not apply, as the assignment to y on line 3 is merely refining, not "upgrading", the label of y. If y's label had been L, this program would have been rejected.

This program can be rejected by deploying a special monitoring rule that preemptively aborts an assignment statement if there is a possibility that the pc is not lower than or equal to the variable's label, as deployed by $\mathsf{GSL}_{\mathsf{Ref}}$ [6]. In the above example, the assignment on line 3 will be aborted, because the pc is H, and y's label could be $\bot$ or L, which might leak information. This ensures noninterference, but unfortunately,

[1]We will write x^ℓ to indicate that variable x has the type τ^ℓ.

```
y := true^?
if (x) then y := false^?
output(H, y)
```

Listing 5. Secure program violating gradual guarantee

the extra check does not retain the dynamic gradual guarantee. That is, enlarging the set of possible labels for the dynamic security type of y will cause the monitor to abort, which contradicts the dynamic gradual guarantee.

Consider the program in Listing 5 with the same security lattice as before such that the variable x is labeled H and y's label is not specified at compile-time. As the pc on line 2 is H and the possible set of labels of y on line 2 is $\{\bot, L, H, \top\}$, the monitor aborts the execution of the program when x is true to satisfy noninterference. However, if y was labeled H or $\top$ at compile-time instead of being ?, the execution would have proceeded and output the value of y to H. In other words, the larger set of possible labels for y on line 2 (because of the unknown label) causes the monitor to abort while the precisely typed version of the program with $y : \mathsf{bool}^H$ is accepted by the monitor, which violates the dynamic gradual guarantee.

To tackle this problem, we leverage the static phase of the gradual type system to determine the set of variables being written to in different branches and loops, and *refine* their possible security labels to implement a monitoring strategy that preserves the dynamic gradual guarantee. At the branch on line 2 in Listing 5, we know that y may be written to inside the branch, therefore, we narrow the possibility of the labels for y to $\{H, \top\}$ as the first step of executing the if statement, regardless of whether x is true or false. This is very similar to how hybrid monitors stop implicit leaks [13], [17], [21], [22]. We will discuss this further in Section IV-C.

IV. A Language with Gradual Security Types

The syntax and typing rules for the language with gradual security types ($\mathbb{WHILE}^{\mathsf{G}}$) are standard and provided in the full version of the paper [19]. In this section, we present the syntax and typing rules for our language with gradual information flow types and evidence ($\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$), define the translation from $\mathbb{WHILE}^{\mathsf{G}}$ to $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$, and explain the operational semantics for our monitor.

A. $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$

The syntax of $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$ is shown in Fig. 1. Gradual types, U, consist of a type (bool, or int) and a gradual security label, g. This label is either a static security label, denoted ℓ; or an imprecise dynamic label, denoted ?. As is standard, ℓ is drawn from $Labs$, a set of labels, which is a part of a security lattice $\mathcal{L} = (Labs, \preccurlyeq)$. Here $\preccurlyeq$ is a partial order between labels in $Labs$. We commonly use the label H to indicate secret, L to indicate public data, and $L \preccurlyeq H$.

The partial-ordering ($\preccurlyeq$) and join operation ($\curlyvee$) on security labels (ℓ) extends to consistent ordering ($\preccurlyeq_c$) and consistent-join ($\curlyvee_c$) to account for ?, as shown in Fig. 2. The consistent subtyping relation is written as $\tau^{g_1} \leq_c \tau^{g_2}$.

Labels	ℓ	::=	$L \mid \ldots \mid H$
Label-intervals	ι	::=	$[\ell_{low}, \ell_{high}]$
Raw values	u	::=	$n \mid \mathsf{true} \mid \mathsf{false}$
Values	v	::=	$(\iota\ u)^g$
Types	τ	::=	$\mathsf{bool} \mid \mathsf{int}$
Gradual labels	g	::=	$? \mid \ell$
Gradual types	U	::=	τ^g
Typing Context	Γ	::=	$\cdot \mid \Gamma, x : U$
Cast evidence	E	::=	(ι_1, ι_2)
Expressions	e	::=	$x \mid v \mid e_1\ \mathsf{bop}\ e_2 \mid E^g\ e$
Variable Set	X	::=	$\{x_1, \ldots, x_n\}$
Commands	c	::=	$\mathsf{skip} \mid c_1; c_2 \mid x := e \mid \mathsf{output}(\ell, e)$
		$\mid$	$\mathsf{if}^X\, e\, \mathsf{then}\, c_1\, \mathsf{else}\, c_2 \mid \mathsf{while}^X\, e\, \mathsf{do}\, c$

Fig. 1. Syntax for the language $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$

$$\frac{\ell_1 \preccurlyeq \ell_2}{\ell_1 \preccurlyeq_c \ell_2} \qquad \frac{}{? \preccurlyeq_c g} \qquad \frac{}{g \preccurlyeq_c ?} \qquad \frac{g_1 \preccurlyeq_c g_2}{\tau^{g_1} \leq_c \tau^{g_2}}$$

$$\frac{}{\ell_1 \curlyvee_c \ell_2 = \ell_1 \curlyvee \ell_2} \qquad \frac{g \neq \top}{? \curlyvee_c g = ?} \qquad \frac{g \neq \top}{g \curlyvee_c ? = ?}$$

$$\frac{}{? \curlyvee_c \top = \top} \qquad \frac{}{\top \curlyvee_c ? = \top}$$

Fig. 2. Operations on gradual labels and types

Recall that examples in Section III-A use a set of possible security labels for preventing information leaks. This is *evidence* attesting to the validity of gradual labels. We use an interval of labels, representing the lowest and the highest possible label, as the refinement only narrows the interval, similar to $\mathsf{GSL}_{\mathsf{Ref}}$ [6].

There are two types of evidence: a label-interval, ι, that justifies the dynamic label ?; and a pair of intervals or the cast evidence, $E{=}(\iota_1, \iota_2)$, that justifies the consistent subtyping relation between two gradual types used in casts. Intuitively, ι represents the range of possible static labels that would allow the program to type-check. For static labels ℓ, the evidence is $[\ell, \ell]$. An interval $[\ell_l, \ell_r]$ is valid iff $\ell_l \preccurlyeq \ell_r$. The rest of the paper only considers valid intervals. Operations leading to an invalid interval are aborted.

A value in $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$ is a raw value with an interval of possible security labels for the gradual label. Raw values are integer constants n, or boolean values. Expressions include values, variables, casts, and binary operations on expressions. An explicit type cast is written $E^g\ e$, where E is the evidence justifying the type cast and g is the label of the resulting type.

Commands include skip, sequencing, assignments, branches, loops, and outputs. This language does not have higher-order stores, so the left-hand side of the assignment is always a global variable. The output command outputs a value at a fixed security label ℓ. We include this command mainly to have a clear statement of the system's observable behavior, so we do not allow output at the imprecise label ?. To prevent implicit leaks, we include a *write-set* of variables, X, which takes into account variable writes in both conditional branches and the loop body.

Label-interval operations: We first define functions and op-

$$\frac{}{\gamma(?) = [\bot, \top]} \qquad \frac{}{\gamma(\ell) = [\ell, \ell]} \qquad \frac{\ell_l \preccurlyeq \ell_r}{\mathsf{valid}([\ell_l, \ell_r])} \qquad \frac{\ell_2 \preccurlyeq \ell_1 \qquad \ell_1' \preccurlyeq \ell_2'}{[\ell_1, \ell_1'] \sqsubseteq [\ell_2, \ell_2']} \qquad \frac{\ell_1' \preccurlyeq \ell_2}{[\ell_1, \ell_1'] \preccurlyeq [\ell_2, \ell_2']}$$

$$\frac{\ell_{1l} \preccurlyeq \ell_{1r} \curlywedge \ell_{2r} \qquad \ell_{2l} \curlyvee \ell_{1l} \preccurlyeq \ell_{2r}}{\mathit{refine}([\ell_{1l}, \ell_{1r}], [\ell_{2l}, \ell_{2r}]) = ([\ell_{1l}, \ell_{1r} \curlywedge \ell_{2r}], [\ell_{2l} \curlyvee \ell_{1l}, \ell_{2r}])} \qquad \frac{(\ell_{1l} \not\preccurlyeq \ell_{1r} \curlywedge \ell_{2r}) \vee (\ell_{2l} \curlyvee \ell_{1l} \not\preccurlyeq \ell_{2r})}{\mathit{refine}([\ell_{1l}, \ell_{1r}], [\ell_{2l}, \ell_{2r}]) = \mathtt{undef}}$$

$$\frac{\iota_1 = [\ell_1, \ell_1'] \qquad \iota_2 = [\ell_2, \ell_2']}{\iota_1 \curlyvee \iota_2 = [\ell_1 \curlyvee \ell_2, \ell_1' \curlyvee \ell_2']} \qquad \frac{\iota_1 \sqsubseteq \gamma(g_1) \qquad \iota_2 \sqsubseteq \gamma(g_2) \qquad g_1 \preccurlyeq_c g_2}{(\iota_1, \iota_2) \vdash \tau^{g_1} \leq_c \tau^{g_2}}$$

$$\frac{\iota_1 = [\ell_{1l}, \ell_{1r}] \qquad \iota_2 = [\ell_{2l}, \ell_{2r}]}{\iota_1 \bowtie \iota_2 = [\ell_{1l} \curlyvee \ell_{2l}, \ell_{1r} \curlywedge \ell_{2r}]} \qquad \frac{\iota'' = (\iota' \bowtie \iota) \qquad \iota'' \sqsubseteq \gamma(g)}{\iota' \bowtie (\iota\ u)^g = (\iota''\ u)^g} \qquad \frac{\mathit{refine}(\iota_1 \bowtie \iota, \iota_2) = (\iota_1', \iota_2')}{\iota \bowtie (\iota_1, \iota_2) = \iota_2'} \qquad \frac{\mathit{refine}(\iota_1 \bowtie \iota, \iota_2) = \mathtt{undef}}{\iota \bowtie (\iota_1, \iota_2) = \mathtt{undef}}$$

Fig. 3. Label and label-interval operations

erations on the label-intervals that are used by the typing rules and operational semantics (shown in Fig. 3). The function $\gamma(g)$ returns the maximum possible label-interval for the gradual label g, assuming $\bot$ and $\top$ are the least and the greatest element in the lattice, respectively. Label-intervals form a lattice with the partial ordering defined as $\iota_1 \sqsubseteq \iota_2$. Here, ι_1 is said to be more precise than ι_2. The label-intervals are refined throughout the execution of the program; i.e., they get more precise. Consider the example in Listing 3. Assume that the security lattice contains two elements L and H such that $L \preccurlyeq H$. Initially, y has a label ? with the evidence $[L, H]$ indicating that any of the two labels are possible. If $x : \mathsf{bool}^H$, then the only possible label for y that allows assignment on line 3 is H. Thus, the evidence for the label on y is refined to $[H, H]$, which makes the program-context's evidence on line 4 $[H, H]$, disallowing assignments to L-labeled variables.

We define $\iota_1 \preccurlyeq \iota_2$ to mean for every security label in ι_1, all labels in ι_2 are at higher or equal positions in the security lattice; and for every security label in ι_2, all labels in ι_1 are at lower or equal positions in the security lattice. Even though this relation is not used in our typing rules or operational semantics, it is an invariant that must hold on the results of the binary label-interval operations used by the noninterference proofs. The function *refine*(ι_1, ι_2) returns the largest sub-intervals of ι_1 and ι_2 (ι_1' and ι_2', respectively) such that $\iota_1' \preccurlyeq \iota_2'$. If the relation does not hold between ι_1' and ι_2', the function returns `undef`.

The join of label-intervals is defined as $\iota_1 \curlyvee \iota_2$. Note that this is not to be confused with the join operation in the lattice that the intervals form. The join of the label-intervals computes the interval corresponding to all possible joins of security labels in those intervals. The operation $\iota_1 \bowtie \iota_2$ computes the intersection of the intervals ι_1 and ι_2. $\iota' \bowtie (\iota\ u)^g$ merges the labels for a value. $\iota \bowtie (\iota_1, \iota_2)$ refines ι_2 based on the intersection of ι and ι_1.

Evidence-based consistent subtyping: Next, we define consistent subtyping relations for both gradual labels and types as supported by label-intervals. The consistent subtyping relation between two gradual types is written as $(\iota_1, \iota_2) \vdash \tau^{g_1} \leq_c \tau^{g_2}$ (defined in Fig. 3). In this relation, ι_1, resp. ι_2 represents the set of possible labels for g_1, resp. g_2, and $g_1 \preccurlyeq_c g_2$. The evidence (ι_1, ι_2) is to justify the consistent security label partial ordering relation between the labels of the gradual types. Note that we do not have $\iota_1 \preccurlyeq \iota_2$ in the premise. The reason is that $(\iota_1, \iota_2) \vdash \tau^{g_1} \leq_c \tau^{g_2}$ is used to type runtime terms; even though $\iota_1 \preccurlyeq \iota_2$ holds initially, as label-intervals are refined from ι_i to ι_i', we cannot guarantee that $\iota_1' \preccurlyeq \iota_2'$ hold. This will break preservation proofs. It is not the gradual type system's job to ensure all execution paths are secure. Instead, the runtime semantics will refine the intervals and abort the computation if necessary when a term is evaluated.

Typing rules: Expressions and commands with evidence are typed using rules shown in Fig. 4.

G-CAST casts an expression of type U_1 to U_2, if the cast evidence E shows that U_1 is a consistent subtype of U_2.

We augment the command typing with an interval for the gradual *pc* label; ι_{pc} is the range of possible static labels for the g_{pc}. The rules are similar to the ones in the original type-system except for the use of evidence for consistent ordering between the gradual labels. An exception is the use of *WtSet*(c) that returns the set of variables being updated or assigned to in the command c. *WtSet* is straightforwardly inductively defined over the structure of c and is shown below:

$$\begin{array}{ll} \mathit{WtSet}(\mathsf{skip}) = \emptyset & \mathit{WtSet}(\mathsf{output}(\ell, e)) = \emptyset \\ \mathit{WtSet}(x := e) = \{x\} & \mathit{WtSet}(\mathsf{while}\ e\ \mathsf{do}\ c) = \mathit{WtSet}(c) \\ \multicolumn{2}{l}{\mathit{WtSet}(c_1; c_2) = \mathit{WtSet}(c_1) \cup \mathit{WtSet}(c_2)} \\ \multicolumn{2}{l}{\mathit{WtSet}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) = \mathit{WtSet}(c_1) \cup \mathit{WtSet}(c_2)} \end{array}$$

Further, G-ASSIGN and G-OUT do not consider expression subtyping and instead rely on the casts inserted by translation.

B. From $\mathbb{WHILE}^{\mathsf{G}}$ *to* $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$

The programs are written in $\mathbb{WHILE}^{\mathsf{G}}$, the language without evidence, which is then translated to $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$. The explicit casts for ASSIGN and OUT are automatically inserted to account for the subtyping of expressions. We show the interesting rules for translating $\mathbb{WHILE}^{\mathsf{G}}$ expressions and commands to $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$ with evidence insertion in Fig. 5. T-ASSIGN inserts a cast for the expression e to have the same label as that of x. For example, $x^L := y^?$ is rewritten to $x := (\mathit{refine}([\bot, \top], [L, L])^L)y$. Similarly, T-OUT casts the expression e to the channel level ℓ.

$$\boxed{\Gamma \vdash e : U}$$

$$\dfrac{\iota \sqsubseteq \gamma(g)}{\Gamma \vdash (\iota\ u)^g : \Gamma(u)^g}\ \textsc{G-Const} \qquad \dfrac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{G-Var} \qquad \dfrac{\forall i \in \{1,2\},\ \Gamma \vdash e_i : \tau^{g_i} \quad g = g_1 \curlyvee_c g_2}{\Gamma \vdash e_1 \ \mathsf{bop}\ e_2 : \tau^g}\ \textsc{G-Bop} \qquad \dfrac{\Gamma \vdash e : \tau^{g_1} \quad E \vdash \tau^{g_1} \leq_c \tau^g}{\Gamma \vdash E^g\ e : \tau^g}\ \textsc{G-Cast}$$

$$\boxed{\Gamma; \iota_{pc}\ g_{pc} \vdash c}$$

$$\dfrac{}{\Gamma; \iota_{pc}\ g_{pc} \vdash \mathsf{skip}}\ \textsc{G-Skip} \qquad \dfrac{\Gamma \vdash e : \mathsf{bool}^g \quad X = \mathit{WtSet}(c) \quad \iota_c = \gamma(g) \quad \Gamma; \iota_{pc} \curlyvee \iota_c\ g_{pc} \curlyvee_c g \vdash c}{\Gamma; \iota_{pc}\ g_{pc} \vdash \mathsf{while}^X\ e\ \mathsf{do}\ c}\ \textsc{G-While}$$

$$\dfrac{\Gamma; \iota_{pc}\ g_{pc} \vdash c_1 \quad \Gamma; \iota_{pc}\ g_{pc} \vdash c_2}{\Gamma; \iota_{pc}\ g_{pc} \vdash c_1; c_2}\ \textsc{G-Seq} \qquad \dfrac{\Gamma \vdash x : \tau^g \quad \Gamma \vdash e : \tau^g \quad \iota_{pc} \sqsubseteq \gamma(g_{pc}) \quad g_{pc} \preccurlyeq_c g}{\Gamma; \iota_{pc}\ g_{pc} \vdash x := e}\ \textsc{G-Assign} \qquad \dfrac{\Gamma \vdash e : \tau^\ell \quad \iota_{pc} \sqsubseteq \gamma(g_{pc}) \quad g_{pc} \preccurlyeq_c \ell}{\Gamma; \iota_{pc}\ g_{pc} \vdash \mathsf{output}(\ell, e)}\ \textsc{G-Out}$$

$$\dfrac{\Gamma \vdash e : \mathsf{bool}^g \quad \iota_c = \gamma(g) \quad X = \mathit{WtSet}(c_1) \cup \mathit{WtSet}(c_2) \quad \forall i = \{1,2\},\ \Gamma; \iota_{pc} \curlyvee \iota_c\ g_{pc} \curlyvee_c g \vdash c_i}{\Gamma; \iota_{pc}\ g_{pc} \vdash \mathsf{if}^X\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}\ \textsc{G-If}$$

Fig. 4. Typing rules for $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$. $\Gamma(u)$ maps a constant to its type (e.g. n to int, and true to bool)

.

$$\boxed{\Gamma \vdash e \rightsquigarrow e' : U}$$

$$\dfrac{\iota = \gamma(g)}{\Gamma \vdash b^g \rightsquigarrow (\iota\ b)^g : \mathsf{bool}^g}\ \textsc{T-Bool} \qquad \dfrac{\iota = \gamma(g)}{\Gamma \vdash n^g \rightsquigarrow (\iota\ n)^g : \mathsf{int}^g}\ \textsc{T-Int} \qquad \dfrac{\Gamma(x) = \tau^g}{\Gamma \vdash x \rightsquigarrow x : \tau^g}\ \textsc{T-Var}$$

$$\dfrac{\forall i \in \{1,2\},\ \Gamma \vdash e_i \rightsquigarrow e_i' : \tau^{g_i} \quad g = g_1 \curlyvee_c g_2}{\Gamma \vdash e_1\ \mathsf{bop}\ e_2 \rightsquigarrow e_1'\ \mathsf{bop}\ e_2' : \tau^g}\ \textsc{T-Bop} \qquad \dfrac{\Gamma \vdash e \rightsquigarrow e' : \tau^{g_1} \quad g_1 \preccurlyeq_c g \quad (\iota_1, \iota_2) = \mathit{refine}(\gamma(g_1), \gamma(g))}{\Gamma \vdash (e :: \tau^g) \rightsquigarrow (\iota_1, \iota_2)^g\ e' : \tau^g}\ \textsc{T-Cast}$$

$$\boxed{\Gamma;\ g_{pc} \vdash c \rightsquigarrow c'}$$

$$\dfrac{\Gamma(x) = \tau^g \quad \Gamma \vdash e \rightsquigarrow e' : \tau^{g'} \quad g' \preccurlyeq_c g \quad (\iota_1, \iota_2) = \mathit{refine}(\gamma(g'), \gamma(g))}{\Gamma;\ g_{pc} \vdash x := e \rightsquigarrow x := (\iota_1, \iota_2)^g\ e'}\ \textsc{T-Assign} \qquad \dfrac{\Gamma \vdash e \rightsquigarrow e' : \tau^g \quad g \preccurlyeq_c \ell \quad (\iota_1, \iota_2) = \mathit{refine}(\gamma(g), \gamma(\ell))}{\Gamma;\ g_{pc} \vdash \mathsf{output}(\ell, e) \rightsquigarrow \mathsf{output}(\ell, (\iota_1, \iota_2)^g\ e')}\ \textsc{T-Out}$$

$$\dfrac{\Gamma \vdash e \rightsquigarrow e' : \mathsf{bool}^g \quad \Gamma;\ g_{pc} \curlyvee_c g \vdash c \rightsquigarrow c' \quad X = \mathit{WtSet}(c')}{\Gamma;\ g_{pc} \vdash \mathsf{while}\ e\ \mathsf{do}\ c \rightsquigarrow \mathsf{while}^X\ e'\ \mathsf{do}\ c'}\ \textsc{T-While} \qquad \dfrac{\Gamma \vdash e \rightsquigarrow e' : \mathsf{bool}^g \quad \Gamma;\ g_{pc} \curlyvee_c g \vdash c_1 \rightsquigarrow c_1' \quad \Gamma;\ g_{pc} \curlyvee_c g \vdash c_2 \rightsquigarrow c_2' \quad X = \mathit{WtSet}(c_1') \cup \mathit{WtSet}(c_2')}{\Gamma;\ g_{pc} \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rightsquigarrow \mathsf{if}^X\ e'\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'}\ \textsc{T-If}$$

Fig. 5. Translation from $\mathbb{WHILE}^{\mathsf{G}}$ to $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$

The other interesting translation rules are T-IF and T-WHILE, which insert a write-set X that includes the set of all variables that might be written to in both the branches and the loop body. We prove that any well-typed term in $\mathbb{WHILE}^{\mathsf{G}}$ is translated to another well-typed term in $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$.

C. Operational Semantics

Runtime constructs: We define additional runtime constructs for our semantics, shown below. The store, δ, maps variables to values with their gradual labels and intervals. The gradual labels of the variables are suffixed on the values for the purpose of evaluation. κ is a stack of pc labels, each of which is a gradual label, g_{pc}, with the corresponding interval, ι_{pc}.

$$\begin{array}{llcl}
\mathit{Store} & \delta & ::= & \cdot \mid \delta, x \mapsto v \\
\mathit{PC\ Stack} & \kappa & ::= & \emptyset \mid (\iota_{pc}\, g_{pc}) \mid \kappa_1 \rhd \kappa_2 \\
\mathit{Actions} & \alpha & ::= & \cdot \mid (\ell, v) \\
\mathit{Commands} & c & ::= & \cdots \mid \{c\} \mid \mathsf{if}\ e\, \mathsf{then}\, c_1\, \mathsf{else}\, c_2
\end{array}$$

The stack is used for evaluating nested if statements. The operation $\kappa_1 \rhd \kappa_2$ indicates that κ_1 is on top of κ_2 in the stack. α is an action, which may be silent or a labeled output. We add two runtime commands. $\{c\}$ is used in evaluating if statements. The curly braces help the monitor keep track of

$\boxed{\delta \;/\; e \Downarrow v}$

$$\frac{}{\delta \;/\; (\iota\, u)^g \Downarrow (\iota\, u)^g}\ \textsc{M-Const} \qquad \frac{}{\delta \;/\; x \Downarrow \delta(x)}\ \textsc{M-Var}$$

$$\frac{\begin{array}{c}\forall i \in \{1,2\},\ \delta \;/\; e_i \Downarrow (\iota_i\, u_i)^{g_i} \\ \iota = \iota_1 \curlyvee \iota_2 \qquad g = g_1 \curlyvee_c g_2 \qquad u = u_1 \;\mathsf{bop}\; u_2\end{array}}{\delta \;/\; e_1 \;\mathsf{bop}\; e_2 \Downarrow (\iota\; u)^g}\ \textsc{M-Bop}$$

$$\frac{\delta \;/\; e \Downarrow (\iota\, u)^g \qquad \iota' = \iota \bowtie E}{\delta \;/\; E^{g'}\; e \Downarrow (\iota'\; u)^{g'}}\ \textsc{M-Cast}$$

$$\frac{\delta \;/\; e \Downarrow (\iota\, u)^g \qquad \iota \bowtie E = \mathtt{undef}}{\delta \;/\; E^{g'}\; e \Downarrow \mathsf{abort}}\ \textsc{M-Cast-Err}$$

Fig. 6. Monitor semantics for expressions

the scope of a branch. The if statement without the write set is used in an intermediate evaluation state.

Expression monitoring semantics: Our monitoring semantics for expressions is of the form $\delta \;/\; e \Downarrow e'$ as shown in Fig. 6. Rules M-Const and M-Var are standard. To perform a binary operation on two values, the operation is performed on the raw values, and the join of their associated intervals and gradual labels is assigned to the computed value. M-Cast refines a value's interval according to the cast evidence. If the refinement is not valid, the execution aborts (M-Cast-Err). Note that none of these operations modify the gradual label of the variable (the type of store locations remain the same); the operations only refine the intervals of the gradual label.

Commands monitoring semantics: Our monitoring semantics for commands is summarized in Fig. 7 and has the form $\kappa, \delta \;/\; c \xrightarrow{\alpha} \kappa', \delta' \;/\; c'$. Additional set of rules where the monitor aborts can be found in Fig. 16 in the Appendix. Rules M-Pc and M-Pop manage commands running in branches or loops. M-Pop pops the top-most *pc* label from the stack, indicating the end of the branch or loop. We use braces around a command, $\{c\}$ to indicate that c is executing in a branch or loop. Such a command is run taking into account only the specific branch's *pc* stack. When the command execution finishes, the braces are removed and the current *pc* label is popped off the stack.

Rule M-Assign updates the label-interval of the value being assigned based on the assignment's context ι_{pc} to prevent information leaks. The resulting label-interval is further restricted using the existing label-interval of the variable to ensure that we only refine the set of possible labels. The function $\mathit{intvl}\,(v)$ returns the label-interval of v. Formally:

$$\mathit{intvl}\,((\iota\; u)^g) = \iota$$

The function *refineLB* refines the lower bound of a value's label-interval and raises it based on the interval ι_{pc}. The updL function is defined in Fig. 8 and uses the interval *restrictLB* operation. We raise the lower bound of the existing interval

$\boxed{\kappa, \delta \;/\; c \xrightarrow{\alpha} \kappa', \delta' \;/\; c'}$

$$\frac{\kappa, \delta \;/\; c_1 \xrightarrow{\alpha} \kappa', \delta' \;/\; c_1'}{\kappa, \delta \;/\; c_1; c_2 \xrightarrow{\alpha} \kappa', \delta' \;/\; c_1'; c_2}\ \textsc{M-Seq}$$

$$\frac{\kappa, \delta \;/\; c \xrightarrow{\alpha} \kappa', \delta' \;/\; c'}{\kappa \rhd \iota_{pc}\; g_{pc}, \delta \;/\; \{c\} \xrightarrow{\alpha} \kappa' \rhd \iota_{pc}\; g_{pc}, \delta' \;/\; \{c'\}}\ \textsc{M-Pc}$$

$$\frac{}{\iota_{pc}\; g_{pc} \rhd \kappa, \delta \;/\; \{\mathsf{skip}\} \longrightarrow \kappa, \delta \;/\; \mathsf{skip}}\ \textsc{M-Pop}$$

$$\frac{}{\kappa, \delta \;/\; \mathsf{skip}; c \longrightarrow \kappa, \delta \;/\; c}\ \textsc{M-Skip}$$

$$\frac{\begin{array}{c}\delta \;/\; e \Downarrow v \qquad v' = \mathit{refineLB}(\iota_{pc}, v) \\ v'' = \mathsf{updL}(\mathit{intvl}\,(\delta(x)), v')\end{array}}{\iota_{pc}\; g_{pc}, \delta \;/\; x := e \longrightarrow \iota_{pc}\; g_{pc}, \delta[x \mapsto v''] \;/\; \mathsf{skip}}\ \textsc{M-Assign}$$

$$\frac{\begin{array}{c}\delta \;/\; e \Downarrow v \qquad v' = \mathit{refineLB}(\iota_{pc}, v) \\ v'' = \mathsf{updL}(\mathit{intvl}\,(\delta(x)), v')\end{array}}{\iota_{pc}\; g_{pc}, \delta \;/\; \mathsf{output}(\ell, e) \xrightarrow{(\ell, v'')} \iota_{pc}\; g_{pc}, \delta \;/\; \mathsf{skip}}\ \textsc{M-Out}$$

$$\frac{\begin{array}{c}\iota_{pc}' = \iota_{pc} \curlyvee \iota \qquad g_{pc}' = g_{pc} \curlyvee_c g \\ c_i = c_1 \text{ if } b = \mathsf{true} \qquad c_i = c_2 \text{ if } b = \mathsf{false}\end{array}}{\begin{array}{c}\iota_{pc}\; g_{pc}, \delta \;/\; \mathsf{if}\; (\iota\; b)^g \;\mathsf{then}\; c_1 \;\mathsf{else}\; c_2 \longrightarrow \\ \iota_{pc}'\; g_{pc}' \rhd \iota_{pc}\; g_{pc}, \delta \;/\; \{c_i\}\end{array}}\ \textsc{M-If}$$

$$\frac{\delta \;/\; e \Downarrow v \qquad \delta' = \mathit{rfL}(\delta, X, \iota_{pc} \curlyvee \mathit{intvl}\,(v))}{\begin{array}{c}\iota_{pc}\; g_{pc}, \delta \;/\; \mathsf{if}^X\; e \;\mathsf{then}\; c_1 \;\mathsf{else}\; c_2 \longrightarrow \\ \iota_{pc}\; g_{pc}, \delta' \;/\; \mathsf{if}\; v \;\mathsf{then}\; c_1 \;\mathsf{else}\; c_2\end{array}}\ \textsc{M-If-Refine}$$

$$\frac{}{\begin{array}{c}\iota_{pc}\; g_{pc}, \delta \;/\; \mathsf{while}^X\; e \;\mathsf{do}\; c \longrightarrow \iota_{pc}\; g_{pc}, \delta \;/ \\ \mathsf{if}^X\; e \;\mathsf{then}\; (c; \mathsf{while}^X\; e \;\mathsf{do}\; c) \;\mathsf{else}\; \mathsf{skip}\end{array}}\ \textsc{M-While}$$

Fig. 7. Monitor semantics for commands

based on the interval of the newly computed value. Note that *restrictLB* differs from *refine* in the upper-bound of the computed interval. If either of these functions return an invalid interval, the execution aborts. The interval need not be checked against the reference's label-interval because the inserted cast would have ruled out unsafe programs earlier. Consider the following program, where $\delta = [x \mapsto ([L, L]\, 3)^L, y \mapsto ([H, H]\, 5)^H]$

$$x := ([\bot, \top], [L, L])\, ([H, H], [\bot, \top])\, y$$

This program tries to cast an H value to ?, then back to L, which is accepted by the type system. The expression to be assigned to x is first evaluated to $([\bot, \top], [L, L])\, ([H, H], [\bot, \top])\, ([H, H]\, 5)^H$, then to $([\bot, \top], [L, L])\, ([H, H]\, 5)^H$, then aborts, because $[H, H] \bowtie ([\bot, \top], [L, L])$ evaluates to $\mathit{refine}([H, H], [L, L]) = \mathtt{undef}$.

$$\frac{\mathit{refine}(\iota_c, \iota) = (\_, \iota')}{\mathit{refineLB}(\iota_c, (\iota\ u)^g) = (\iota'\ u)^g} \qquad \frac{}{\mathit{rfL}(\delta, \cdot, \iota) = \delta}$$

$$\frac{\delta' = \mathit{rfL}(\delta, X, \iota) \qquad v' = \mathit{refineLB}(\iota, v) \neq \texttt{undef}}{\mathit{rfL}((\delta, x \mapsto v), (X, x), \iota) = \delta', x \mapsto v'}$$

$$\frac{\ell_{1l} \curlyvee \ell_{2l} \preccurlyeq \ell_{1r}}{\mathit{restrictLB}([\ell_{1l}, \ell_{1r}], [\ell_{2l}, \ell_{2r}]) = [\ell_{1l} \curlyvee \ell_{2l}, \ell_{1r}]}$$

$$\frac{}{\mathsf{updL}(\iota_o, (\iota_n\, u_n)^g) = (\mathit{restrictLB}(\iota_o, \iota_n) u_n)^g}$$

Fig. 8. Label-interval operations for the monitor

We explain the assignment rule via examples. Consider a three-point lattice $L \preccurlyeq M \preccurlyeq H$, the following command $x := [H, H]\ 5^?$, and two stores $\delta_1 = x \mapsto [M, H]\ 1^?$ and $\delta_2 = x \mapsto [L, M]\ 2^?$. Assume the following current pc-interval $\iota_{pc} = [L, H]$. Here, $v = [H, H]\ 5^?$. The second premise further refines the interval of v to make sure that the pc context is lower than or equal to the interval of the value to be written. This is to prevent low assignments in a high context. For this example, $\mathit{refine}([L, H], [H, H]) = ([L, H], [H, H])$, so the intervals remain the same. Next, we narrow down the possible label set using the existing value's interval. This is to adhere to our design choice that we do not change the type of the variables and only narrow down the label choices (Section III-A). For δ_1, $v'' = [H, H]\ 5^?$, so now x stores a secret value with label H. For store δ_2, $\mathit{restrictLB}([L, M], [H, H])$ is not defined and the monitor aborts.

When the old value in the store has a label-interval that is lower than the label-interval of the new value to be stored, the monitor aborts; for instance, under the store where $x \mapsto ([L, L]0)^?$, the monitor aborts the execution of $x := ([H, H]1)^?$, as $\mathit{restrictLB}([L, L], [H, H])$ is not defined. This is also consistent with our design choice to only refine the label set, not update it.

M-OUT makes similar comparisons as M-ASSIGN to ensure that the output is permitted. Rule M-IF is standard. The pc label is determined by joining the current pc with the gradual label and interval of the branch-predicate's value. Here, the pc stack grows and the branch is placed in the scoping braces. The rule for `while` reduces it to `if`. Rule M-IF-REFINE is the key for preventing implicit leaks. We refine the intervals for variables in both branches according to the write set, X, which contains the set of all variables being updated in either one of the two branches. Fig. 8 includes the auxiliary definitions for refining the intervals of variables in a write set. The function *rfL* refines the label-interval of values in the store and is defined inductively. Here, it is used to refine the intervals of the variables in the write set to be at least as high as the lower label in the interval of the current pc. When the functions *rfL* return `undef`, the execution aborts.

Example: Below, we define two initial memories, δ_t maps x to true and δ_f maps x to false. Both y and z store true initially with y's label being ? and z being L such that $L \sqsubseteq H$.

$$\begin{array}{lcl}
\delta_y & = & y \mapsto [L, H]\mathsf{true}^?\\
\delta_z & = & z \mapsto [L, L]\mathsf{true}^L\\
\delta_t & = & x \mapsto [H, H]\mathsf{true}^H\\
\delta_f & = & x \mapsto [H, H]\mathsf{false}^H\\
c_1 & = & \mathsf{if}^{\{y\}}\ x\ \mathsf{then}\ y := [L, H]\mathsf{false}^?\ \mathsf{else\ skip}\\
c_2 & = & \mathsf{if}^{\{z\}}\ y\ \mathsf{then}\ z := [L, L]\mathsf{false}^L\ \mathsf{else\ skip}
\end{array}$$

Below is the execution starting from the state where x is true.

$$\begin{array}{l}
\ [L, L]\ L,\ (\delta_t, \delta_z, \delta_y)\ /\ c_1; c_2\\
\longrightarrow [L, L]\ L,\ (\delta_t, \delta_z, y \mapsto [H, H]\mathsf{true}^?)\ /\\
\qquad \mathsf{if}\ x\ \mathsf{then}\ y :=\ [L, H]\mathsf{false}^?\ \mathsf{else\ skip}; c_2\\
\longrightarrow [H, H]\ H \triangleright [L, L]\ L,\ (\delta_t, \delta_z, y \mapsto [H, H]\mathsf{true}^?)\ /\\
\qquad \{y := [L, H]\mathsf{false}^?\}; c_2\\
\longrightarrow [H, H]\ H \triangleright [L, L]\ L,\ (\delta_t, \delta_z, y \mapsto [H, H]\mathsf{false}^?)\ /\\
\qquad \{\mathsf{skip}\}; c_2\\
\longrightarrow [H, H]\ H \triangleright [L, L]\ L,\ (\delta_t, \delta_z, y \mapsto [H, H]\mathsf{false}^?)\ /\\
\qquad \mathsf{skip}; c_2\\
\longrightarrow [L, L]\ L,\ (\delta_t, \delta_z, y \mapsto [H, H]\mathsf{false}^?)\ /\ c_2\\
\longrightarrow \mathsf{abort}
\end{array}$$

In the last step, *rfL* fails, because the operation $\mathit{refine}([H, H], [L, L])$ produces an invalid label-interval. Now let's see the execution starting from $x \mapsto \mathsf{false}$.

$$\begin{array}{l}
\ [L, L]\ L, (\delta_f, \delta_z, \delta_y)\ /\ c_1; c_2\\
\longrightarrow [L, L]\ L, (\delta_f, \delta_z, y \mapsto [H, H]\mathsf{true}^?)\ /\\
\qquad \mathsf{if}\ x\ \mathsf{then}\ y := [L, H]\mathsf{false}^?\ \mathsf{else\ skip}; c_2\\
\longrightarrow [H, H]\ H \triangleright [L, L]\ L, (\delta_f, \delta_z, y \mapsto [H, H]\mathsf{true}^?)\ /\\
\qquad \{\mathsf{skip}\}; c_2\\
\longrightarrow [H, H]\ H \triangleright [L, L]\ L, (\delta_f, \delta_z, y \mapsto [H, H]\mathsf{true}^?)\ /\\
\qquad \mathsf{skip}; c_2\\
\longrightarrow [L, L]\ L, (\delta_f, \delta_z, y \mapsto [H, H]\mathsf{true}^?)\ /\ c_2\\
\longrightarrow \mathsf{abort}
\end{array}$$

Notice that the label-intervals of y are changed the same way as when we start the execution from δ_t. Ultimately, the program aborts for the same reason.

V. NONINTERFERENCE

To prove noninterference, we extend $\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$ with pairs of values, expressions, and commands to simulate two executions which differ on secret values. This allows us to reduce our noninterference proof to a preservation proof [18].

A. Paired Execution

Syntax: The augmented syntax with pairs is shown below.

$$\begin{array}{llll}
\mathit{Values} & v & ::= & (\iota\ u)^g \mid \langle \iota_1\ u_1 \mid \iota_2\ u_2 \rangle^g\\
\mathit{Cmd.} & c & ::= & \cdots \mid \langle \kappa_1, \iota_1, c_1 \mid \kappa_2, \iota_2, c_2 \rangle_g
\end{array}$$

The store δ is extended to contain pairs of values. We also extend commands to be paired but do not allow pairs to be nested; an invariant maintained by our operational semantics. We only use pairs for values and commands whose values and effects are not observable by the adversary (are "high"). Pairs of commands are part of the runtime statement, generated as

Expression Semantics: $\boxed{\delta /_i e \Downarrow v}$

$$\frac{}{\delta /_i (\iota\, u)^g \Downarrow (\iota\, u)^g}\ \textsc{P-Const} \qquad \frac{}{\delta /_i x \Downarrow \mathsf{rd}_i\ \delta(x)}\ \textsc{P-Var}$$

$$\frac{\delta /_i e \Downarrow v \qquad v' = (E, g) \triangleright v}{\delta /_i E^g\ e \Downarrow v'}\ \textsc{P-Cast}$$

Command Semantics: $\boxed{\kappa, \delta /_i c \xrightarrow{\alpha} \kappa', \delta' /_i c'}$

$$\frac{\begin{array}{c}\kappa_i \triangleright \iota_{pc} \curlyvee \iota_i\ g_{pc} \curlyvee_c g, \delta /_i c_i \xrightarrow{\alpha} \\ \kappa'_i \triangleright \iota_{pc} \curlyvee \iota_i\ g_{pc} \curlyvee_c g, \delta' /_i c'_i \\ c_j = c'_j \qquad \kappa_j = \kappa'_j \qquad \{i, j\} = \{1, 2\}\end{array}}{\begin{array}{c}\iota_{pc}\ g_{pc}, \delta\ /\ \langle \kappa_1, \iota_1, c_1 \mid \kappa_2, \iota_2, c_2 \rangle_g \xrightarrow{\alpha} \\ \iota_{pc}\ g_{pc}, \delta'\ /\ \langle \kappa'_1, \iota_1, c'_1 \mid \kappa'_2, \iota_2, c'_2 \rangle_g\end{array}}\ \textsc{P-C-Pair}$$

$$\frac{\begin{array}{c}c_j = c_1 \text{ if } u_1 = \mathsf{true} \qquad c_j = c_2 \text{ if } u_1 = \mathsf{false} \\ c_k = c_1 \text{ if } u_2 = \mathsf{true} \qquad c_k = c_2 \text{ if } u_2 = \mathsf{false}\end{array}}{\begin{array}{c}\iota_{pc}\ g_{pc}, \delta\ /\ \mathsf{if}\ \langle \iota_1\ u_1 \mid \iota_2\ u_2 \rangle^g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \longrightarrow \\ \iota_{pc}\ g_{pc}, \delta\ /\ \langle \emptyset, \iota_1, c_j \mid \emptyset, \iota_2, c_k \rangle_g\end{array}}\ \textsc{P-Lift-If}$$

$$\frac{}{\begin{array}{c}\iota_{pc}\ g_{pc}, \delta\ /\ \langle \emptyset, \iota_1, \mathsf{skip} \mid \emptyset, \iota_2, \mathsf{skip} \rangle_g \longrightarrow \\ \iota_{pc}\ g_{pc}, \delta\ /\ \mathsf{skip}\end{array}}\ \textsc{P-Skip-Pair}$$

$$\frac{\delta /_i e \Downarrow v \qquad v' = \textit{refineLB}(\iota_{pc}, v)}{\begin{array}{c}\iota_{pc}\ g_{pc}, \delta /_i x := e \longrightarrow \\ \iota_{pc}\ g_{pc}, \delta[x \mapsto \mathsf{upd}_i\ \delta(x)\ v'] /_i \mathsf{skip}\end{array}}\ \textsc{P-Assign}$$

$$\frac{\begin{array}{c}\delta /_i e \Downarrow v \\ v' = \textit{refineLB}(\iota_{pc}, v) \qquad v'' = \mathsf{updL}\ [\ell, \ell]\ v'\end{array}}{\iota_{pc}\ g_{pc}, \delta /_i \mathsf{output}(\ell, e) \xrightarrow{(i, \ell, v'')} \iota_{pc}\ g_{pc}, \delta /_i \mathsf{skip}}\ \textsc{P-Out}$$

Fig. 9. Selected rules of paired executions

a result of evaluating a branching statement. Each command represents an independent execution, capable of changing its own pc stack. As a result, we include local pc stacks in the pair with each command. The rationale behind additional pc stacks in command pairs is explained with the semantics.

Label-interval operations on pairs: The interval of a paired value is a pair of intervals, defined below.

$$\textit{intvl}\,(\langle \iota_1\ u_1 \mid \iota_2\ u_2 \rangle^g) = \langle \iota_1 \mid \iota_2 \rangle$$

The intersection of an interval and a paired value is defined as follows. Other extensions to label-interval operations can be found in the full version of the paper [19].

$$\iota \bowtie \langle \iota_1\ u_1 \mid \iota_2\ u_2 \rangle^g = \langle \iota \bowtie \iota_1\ u_1 \mid \iota \bowtie \iota_2\ u_2 \rangle^g$$

Memory read and update operations: As we allow the intervals of values to be refined, the store read (rd) and update (upd) operations for paired values need to make sure that the correct paired value is read or updated. These functions are shown in Fig. 17 in the Appendix.

Operational semantics for pairs: The operational semantics are augmented with an index, i. The judgments now are of the form $\delta /_i e \Downarrow e'$ and $\kappa, \delta /_i c \longrightarrow \kappa', \delta' /_i c'$. The index i indicates which branch of a pair is executing (when $i \in \{1, 2\}$) or if it is a top-level execution (when i is omitted). Most of the rules can be directly obtained by adding the i to the monitor semantics shown in Fig. 6 and 7. Rules that deal with pairs, including read and write to the store need to be modified. We explain important rule changes (shown in Fig. 9).

Rule P-Var uses the function $\mathsf{rd}_i\ v$ to retrieve the value indexed by i within v. To evaluate a cast over a pair of values, we push the cast inside the pair (P-Cast).

Each command in the pair (P-C-Pair) can make progress independently and the premise of the rule is indexed by the corresponding i. Here κ_i is the pc stack specific to c_i. Consider a command $c = \langle c_1 \mid c_2 \rangle$, where both c_1 and c_2 have nested if statements. The execution of c will create different κ_1 and κ_2 when executing c_1 and c_2. Next, ι_i is the pc label-interval demonstrating that c is supposed to execute in a "high" context (unobservable by the adversary). The bottom pc in the stack is joined with ι_i. We will come back to this point when explaining the typing rules.

Rule P-Lift-If lifts the pair that appears as branch conditions to generate a paired command. The resulting commands on each side of the pair are determined by the value in the corresponding side of the branch condition. The branching context ι_i is the runtime interval of the branching condition. The initial local pc stack is empty.

Note that the individual branches do not contain pairs of commands. The only rule that generates paired command is P-Lift-If. To see how the semantics prevent nesting command pairs and how paired execution represents low runs with different secrets, consider the program in Listing 6. Assume that a and b are variables containing paired values such that $a = \langle u_{a1} \mid u_{a2} \rangle^H$ and $b = \langle u_{b1} \mid u_{b2} \rangle^H$, meaning both a and b contain secrets and u_{a1} and u_{b1} are values for the first execution and u_{a2} and u_{b2} are for the second. We ignore the intervals in this example for simplicity of exposition. On line 1, we use the P-Lift-If rule since we branch on a pair of values to create paired commands. In the first execution, if $u_{a1} = \mathsf{true}$, we take the then branch. When evaluating b inside the branch we take the first part of the pair using the expression evaluation rules and rd_i operation (Fig. 17) for $i = 1$, i.e., $\mathsf{rd}_1\ b = \lfloor b \rfloor_1 = u_{b1}$. Thus, the branch on line 2 becomes: $\mathsf{if}\ u_{b1}\ \mathsf{then} \ldots$ while the remaining parts remain the same. Similarly in the second execution, based on the value of u_{a2}, either the then branch or the else branch is chosen. If the then branch is chosen, the branch on line 2 becomes $\mathsf{if}\ u_{b2}\ \mathsf{then} \ldots$ as we are in the second execution of the branch ($i = 2$) on line 1 and $\mathsf{rd}_2\ b = \lfloor b \rfloor_2 = u_{b2}$. Generating two different runs of the program is sufficient for reasoning about noninterference, which is what the projection semantics do.

Local pc refinements in pairs are forgotten when both sides of the pair finish executing in P-Skip-Pair. This is similar to the P-Pop rule where pc for the branch or loop is forgotten.

Rule P-Assign deals with the complexity of pairs updating

```
1 if $a$ then
2     if $b$ then $y := [\bot, \top]1^?$
3     else skip
4 else $y := [\bot, \top]2^?$
```

Listing 6. Example program to explain branching on pairs

the store in one branch with the helper function $\mathsf{upd}_i\ v_o\ v_n$ (defined in Fig. 17). The refinement of labels during store updates is the same as the monitor semantics. When the update comes from a specific branch of execution ($i \in \{1, 2\}$), the value for the other branch should be preserved. If the value in the store is already a pair, only the i^{th} sub-expression is updated. Reconsider the example in Listing 6. The assignment on line 2 happens in either of the two branches, or both the branches depending on the values of u_{a1} to u_{b2}. If it happens in only the first projection, the first part of the value-pair in y is updated. Suppose that $y = \langle [H, \top]0 \,|\, [H, \top]42 \rangle^?$, initially, and $u_{a1} = u_{b1} = \mathsf{true}$. Then, the value of y after the assignment on line 2 becomes $y = \langle [H, \top]1 \,|\, [H, \top]42 \rangle^?$. If $u_{a2} = \mathsf{false}$, then the else branch is taken, and at the end of the assignment on line 4 the value of y is updated to $y = \langle [H, \top]1 \,|\, [H, \top]2 \rangle^?$. The first part of the pair is already updated through the then branch as we evaluate the two runs one after the other when branching on a pair of values.

If the store value is not a pair, the value becomes a pair where the i^{th} sub-expression is the updated value, and the other sub-expression is the old value. Considering the same example as above, if initially $y = ([H, \top]42)^?$, then at the end of then branch with $u_{a1} = \mathsf{true}$, the updated value of y is $y = \langle [H, \top]1 \,|\, [H, \top]42 \rangle^?$. When updates happen at the top-level, the entire value in the store should be updated. The first rule applies when either the old or the new value is a pair and the second rule applies when none of them are pairs. Note that this is the reason why the intervals in a pair may differ.

The output rule is mostly the same. The event being output now includes the index to aid the statement and proof of noninterference. The P-IF-REFINE rule (for M-IF-REFINE) uses an augmented version of *rfL*, which only refines label-intervals for the i^{th} branch.

B. Semantic Soundness and Completeness

To connect the semantics of the extended language with pairs to the monitor semantics, we prove soundness and completeness theorems. These theorems depend on projections of the store, expression- and command-configurations. Similar to the value projection seen before, the goal of these projections is to obtain one execution from a paired execution.

The projection of a paired value, a paired interval, a normal value and interval are straightforward and defined in Fig. 17. The projection of stores (δ) and traces ($\mathbb{T}$) is inductively defined as shown in Fig. 10. The projection function only keeps the output events produced by the execution of concern and ignores output performed by the other execution. The projection function for expression configurations is $\lfloor \delta \ / \ e \rfloor_i = \lfloor \delta \rfloor_i \ / \ e$ and for command configurations is defined in Fig. 10.

Store projection:

$$\lfloor \cdot \rfloor_i = \ \cdot \qquad\qquad \lfloor \delta, x \mapsto v \rfloor_i = \lfloor \delta \rfloor_i, x \mapsto \lfloor v \rfloor_i$$

Trace projection:

$$\begin{array}{rcll}
\lfloor \cdot \rfloor_i & = & \cdot & \\
\lfloor \mathbb{T}, (\ell, v) \rfloor_i & = & \lfloor \mathbb{T} \rfloor_i, (\ell, \lfloor v \rfloor_i) & \\
\lfloor \mathbb{T}, (j, \ell, v) \rfloor_i & = & \lfloor \mathbb{T} \rfloor_i, (\ell, v) & \textit{if } i = j \\
\lfloor \mathbb{T}, (j, \ell, v) \rfloor_i & = & \lfloor \mathbb{T} \rfloor_i & \textit{if } i \neq j
\end{array}$$

Command-configuration projection:

$$\frac{}{\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ \mathsf{skip} \rfloor_i = \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ \mathsf{skip}}$$

$$\frac{\lfloor \kappa, \delta \ / \ c_1 \rfloor_i = \kappa', \delta' \ / \ c_1'}{\lfloor \kappa, \delta \ / \ c_1; c_2 \rfloor_i = \kappa', \delta' \ / \ c_1'; c_2}$$

$$\frac{}{\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ x := e \rfloor_i = \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ x := e}$$

$$\frac{\lfloor \kappa, \delta \ / \ c \rfloor_i = \kappa', \delta' \ / \ c'}{\lfloor \kappa \rhd \iota_{pc}\ g_{pc}, \delta \ / \ \{c\} \rfloor_i = \kappa' \rhd \iota_{pc}\ g_{pc}, \delta' \ / \ \{c'\}}$$

$$\frac{}{\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ \mathsf{output}(\ell, e) \rfloor_i = \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ \mathsf{output}(\ell, e)}$$

$$\frac{\forall \{i, j\} \in \{1, 2\}, \ c_i' = \begin{cases} \mathsf{skip} & \text{if } c_i = \mathsf{skip} \text{ and } c_j \neq \mathsf{skip} \\ \{c_i\} & \text{else} \end{cases}}{\begin{array}{c}\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ \langle \kappa_1, \iota_1, c_1 \,|\, \kappa_2, \iota_2, c_2 \rangle_g \rfloor_i = \\ \kappa_i \rhd (\iota_{pc} \curlyvee \iota_i)\ (g_{pc} \curlyvee_c g) \rhd \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ c_i' \end{array}}$$

$$\frac{}{\begin{array}{c}\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ \langle \emptyset, \iota_1, \mathsf{skip} \,|\, \emptyset, \iota_2, \mathsf{skip} \rangle_g \rfloor_i = \\ \kappa_i \rhd (\iota_{pc} \curlyvee \iota_i)\ (g_{pc} \curlyvee_c g) \rhd \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ \{\mathsf{skip}\} \end{array}}$$

$$\frac{}{\begin{array}{c}\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ \mathsf{if}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rfloor_i = \\ \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ \mathsf{if}\ \lfloor v \rfloor_i\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \end{array}}$$

$$\frac{}{\begin{array}{c}\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ \mathsf{if}^X\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rfloor_i = \\ \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ \mathsf{if}^X\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \end{array}}$$

$$\frac{}{\lfloor \iota_{pc}\ g_{pc}, \delta \ / \ \mathsf{while}^X\ e\ \mathsf{do}\ c \rfloor_i = \iota_{pc}\ g_{pc}, \lfloor \delta \rfloor_i \ / \ \mathsf{while}^X\ e\ \mathsf{do}\ c}$$

Fig. 10. Projections

The interesting case is the projection of a command pair. We reassemble the *pc* stack and wrap c_i with curly braces to reflect the fact that these pairs only appear in an if branch.

The Soundness theorem ensures that if a configuration can transition to another configuration, then its projection can transition to the projection of the resulting configuration, generating the same trace modulo projection. The Completeness theorem ensures that if both projections of a configuration terminate, then the configuration terminates in an equivalent state. We write, $\vdash \kappa, \delta \ /_i \ c$ wf, to indicate that the configuration is

$$\boxed{\Gamma; \kappa \vdash_r c}$$

$$\frac{\Gamma; \kappa \vdash_r c}{\Gamma; \kappa \rhd \iota\ g_{pc} \vdash_r \{c\}}\ \text{R-POP} \qquad \frac{\Gamma; \iota\ g_{pc} \vdash c}{\Gamma; \iota\ g_{pc} \vdash_r c}\ \text{R-END}$$

$$\frac{\Gamma; \kappa \rhd \iota_{pc}\ g_{pc} \vdash_r c_1 \qquad \Gamma; \iota_{pc}\ g_{pc} \vdash c_2 \qquad \kappa \neq \emptyset}{\Gamma; \kappa \rhd \iota_{pc}\ g_{pc} \vdash_r c_1; c_2}\ \text{R-C-SEQ}$$

$$\frac{\begin{array}{c}\forall i \in 1,2,\ \Gamma; \kappa_i \rhd (\iota_{pc} \curlyvee \iota_i)\ (g_{pc} \curlyvee_c g) \vdash_r c_i \\ \iota_i \vdash g \in H(\ell_A)\end{array}}{\Gamma; \iota_{pc}\ g_{pc} \vdash_r \langle \kappa_1, \iota_1, c_1 \mid \kappa_2, \iota_2, c_2 \rangle_g}\ \text{R-C-PAIR}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{bool}^g \\ \iota_g = \gamma(g) \qquad \forall i \in \{1,2\},\ \Gamma; \iota_{pc} \curlyvee \iota_g\ g_{pc} \curlyvee_c g \vdash c_i\end{array}}{\Gamma; \iota_{pc}\ g_{pc} \vdash_r \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}\ \text{R-C-IF}$$

Fig. 11. Typing rules for commands with pairs

well-formed (defined in Appendix A). Theorems 1 and 2 are the formal soundness and completeness theorem statements. The proofs can be found in the full version of the paper [19].

Theorem 1 (Soundness)**.** *If* $\kappa, \delta\ /\ c \overset{\mathbb{T}}{\longrightarrow}{}^* \kappa', \delta'\ /\ c'$ *where* $\vdash \kappa, \delta\ /\ c\ \mathsf{wf}$*, then* $\forall i \in \{1,2\}$, $\lfloor \kappa, \delta\ /\ c \rfloor_i \overset{\lfloor \mathbb{T} \rfloor_i}{\longrightarrow}{}^* \lfloor \kappa', \delta'\ /\ c' \rfloor_i$

Theorem 2 (Completeness)**.** *If* $\forall i \in \{1,2\}$, $\lfloor \kappa, \delta\ /\ c \rfloor_i \overset{\mathbb{T}_i}{\longrightarrow}{}^* \kappa_i, \delta_i\ /\ \mathsf{skip}$ *and* $\vdash \kappa, \delta\ /\ c\ \mathsf{wf}$*, then* $\exists \kappa', \delta'$ *s.t.* $\kappa, \delta\ /\ c \overset{\mathbb{T}}{\longrightarrow}{}^* \kappa', \delta'\ /\ \mathsf{skip}$*,* $\lfloor \kappa', \delta'\ /\ \mathsf{skip} \rfloor_i = \kappa_i, \delta_i\ /\ \mathsf{skip}$ *and* $\mathbb{T}_i = \lfloor \mathbb{T} \rfloor_i$*.*

C. Preservation

Before we explain the typing rules for the extended configuration, we define another label relation. A gradual label is said to be "high" w.r.t an attacker, if the lower label in the interval is not lower than or equal to the level of the attacker.

$$\frac{\iota = [\ell_l, \ell_r] \qquad \ell_l \not\preccurlyeq \ell_A \qquad \iota \sqsubseteq \gamma(g)}{\iota \vdash g \in H(\ell_A)}$$

All the pair typing rules are parameterized over attacker's label ℓ_A, which we omit from the rules for simplicity. The typing rule for value-pairs is shown below. The second premise checks that the interval is representative of the gradual type U. The last premise checks if U's security label is high, meaning this pair of values is non-observable to the adversary.

$$\frac{\begin{array}{c}\forall i \in \{1,2\},\ \Gamma \vdash \iota_i\ u_i : \tau^g \\ \iota_i \sqsubseteq \gamma(g) \qquad \iota_i \vdash (g) \in\ H(\ell_A)\end{array}}{\Gamma \vdash \langle \iota_1\ u_1 \mid \iota_2\ u_2 \rangle : \tau^g}\ \text{R-V-PAIR}$$

The judgement for typing commands with pairs is of the form $\Gamma; \kappa \vdash_r c$. Fig. 11 summarizes these typing rules.

Rule R-POP types the inner command with only the top part of the *pc* stack. When the *pc* stack contains only one element, R-END directly uses command typing. For pairs, R-C-PAIR first checks that each c_i is well-typing, using the *pc* context assembled from the local *pc* context. The second premise makes sure that these commands are typed (executed) in a high context. Here ι_i is the witness for g, which demonstrates that c_i are high commands. The sequencing statement types the second command using only the last *pc* on the stack because the execution order is from left to right. We can only encounter branches in the first part of a sequencing statement and not the second part before beginning the execution of the second command in the sequence. The typing rule for if statements without a write set is straightforward.

Store typing:

$$\frac{}{\vdash \cdot : \cdot}\ \text{T-S-EMP} \qquad \frac{\vdash \delta : \Gamma \qquad \Gamma \vdash v : U}{\vdash \delta, x \mapsto v : \Gamma, x : U}\ \text{T-S-IND}$$

Configuration typing:

$$\frac{\vdash \delta : \Gamma \qquad \Gamma; \kappa \vdash_r c}{\vdash \kappa, \delta, c}\ \text{T-CONF}$$

Trace typing:

$$\frac{\vdash v : U \qquad \vdash \mathit{intvl}\,(v) \preccurlyeq [\ell, \ell]}{\vdash (\ell, v)}\ \text{T-A-OUT}$$

$$\frac{\vdash v : U \qquad \ell \not\preccurlyeq \ell_A \qquad \vdash \mathit{intvl}\,(v) \preccurlyeq [\ell, \ell]}{\vdash (i, \ell, v)}\ \text{T-A-OUTI}$$

$$\frac{}{\vdash \cdot}\ \text{T-T-EMP} \qquad \frac{\vdash \alpha \qquad \vdash \mathbb{T}}{\vdash \alpha, \mathbb{T}}\ \text{T-T-IND}$$

Fig. 12. Store, trace and configuration typing

We define store, trace and configuration typing in Fig. 12. The store δ types in the typing environment Γ if all variables in δ are mapped to their respective type and gradual label in Γ. We define top-level configuration typing as $\vdash \kappa, \delta, c$. To type traces and actions, the output value needs to be well-typed, and the label-interval of the value has to be lower than or equal to the channel label.

Using these definitions, we prove that our paired execution semantics preserve the configuration typing and generate a well-typed trace (Theorem 3). We write, $\vdash \kappa, \delta\ /_i\ c\ \mathsf{sf}$ for $i \in \{\cdot, 1, 2\}$, to indicate that the configuration is safe. We say a configuration is safe if all of the following hold:

1) if $i \in \{1,2\}$, then $\kappa \in H(\ell_A)$, $\forall x \in \mathit{WtSet}(c)$, $\mathit{intvl}\,(\delta(x)) \in H(\ell_A)$
2) if $c = \mathsf{if}\ \langle \_ \mid \_ \rangle\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$, then $\forall x \in \mathit{WtSet}(c)$, $\mathit{intvl}\,(\delta(x)) \in H(\ell_A)$
3) if $c = \langle \kappa_1, \iota_1, c_1 \mid \kappa_2, \iota_2, c_2 \rangle_g$, then $\forall i \in \{1,2\}$, $\iota_i \vdash g \in H(\ell_A)$, and $\forall x \in \mathit{WtSet}(c)$, $\mathit{intvl}\,(\delta(x)) \in H(\ell_A)$

Theorem 3 (Preservation)**.** *If* $\kappa, \delta\ /\ c \overset{\mathbb{T}}{\longrightarrow}{}^* \kappa', \delta'\ /\ c'$ *with* $\vdash \kappa, \delta, c$ *and* $\vdash \kappa, \delta\ /\ c\ \mathsf{sf}$*, then* $\vdash \kappa', \delta', c'$ *and* $\vdash \mathbb{T}$

Store equivalence:

$$\frac{g \preccurlyeq_c \ell_A \qquad \iota \sqsubseteq \gamma(g) \qquad \vdash (\iota\ u)^g : U}{\vdash (\iota\ u)^g \approx_{\ell_A} (\iota\ u)^g : U}\ \textsc{EqV-L}$$

$$\frac{\forall i \in \{1,2\}, \iota_i \vdash g \in H(\ell_A) \qquad \vdash (\iota_i\ u_i)^g : U}{\vdash (\iota_1\ u_1)^g \approx_{\ell_A} (\iota_2\ u_2)^g : U}\ \textsc{EqV-H}$$

$$\frac{}{\vdash \cdot \approx_{\ell_A} \cdot : \cdot}\ \textsc{EqS-Emp}$$

$$\frac{\vdash \delta_1 \approx_{\ell_A} \delta_2 : \Gamma \qquad \vdash v_1 \approx_{\ell_A} v_2 : U}{\vdash \delta_1, x \mapsto v_1 \approx_{\ell_A} \delta_2, x \mapsto v_2 : \Gamma, x : U}\ \textsc{EqS-Ind}$$

Trace equivalence:

$$\frac{}{\vdash [] \approx_{\ell_A} []}\ \textsc{EqT-E}$$

$$\frac{\vdash \mathbb{T}_1 \approx_{\ell_A} \mathbb{T}_2 \qquad \ell_1 = \ell_2 \preccurlyeq \ell_A \qquad v_1 = v_2}{\vdash (\ell_1, v_1) :: \mathbb{T}_1 \approx_{\ell_A} (\ell_2, v_2) :: \mathbb{T}_2}\ \textsc{EqT-L}$$

$$\frac{\vdash \mathbb{T}_1 \approx_{\ell_A} \mathbb{T}_2 \qquad \ell_1 \not\preccurlyeq \ell_A}{\vdash (\ell_1, v_1) :: \mathbb{T}_1 \approx_{\ell_A} \mathbb{T}_2}\ \textsc{EqT-Hl}$$

$$\frac{\vdash \mathbb{T}_1 \approx_{\ell_A} \mathbb{T}_2 \qquad \ell_2 \not\preccurlyeq \ell_A}{\vdash \mathbb{T}_1 \approx_{\ell_A} (\ell_2, v_2) :: \mathbb{T}_2}\ \textsc{EqT-Hr}$$

Fig. 13. Equivalence definitions

D. Noninterference

We show that the gradual type system presented above satisfies termination-insensitive noninterference. We start by defining equivalence for values and stores (Fig. 13). Two values are said to be equivalent to an adversary at level ℓ_A if either they are both visible to the adversary and are the same, or neither are observable by the adversary. We also define equivalence of traces w.r.t an adversary at level ℓ_A in Fig. 13. The following noninterference theorem (Theorem 4) states that given a program and two stores equivalent for an adversary at level ℓ_A, if the program terminates in both the runs, then the ℓ_A-observable actions on both runs are the same.

Theorem 4 (Noninterference). *Given an adversary label ℓ_A, a program c, and two stores δ_1, δ_2, s.t., $\vdash \delta_1 \approx_{\ell_A} \delta_2 : \Gamma$, and $\Gamma; [\bot, \bot]\ \bot \vdash c$, and $\forall i \in \{1,2\}$, $[\bot, \bot]\ \bot, \delta_i\ /\ c \xrightarrow{\mathbb{T}_i}{}^* \kappa_i, \delta_i', \mathsf{skip}$, then $\vdash \mathbb{T}_1 \approx_{\ell_A} \mathbb{T}_2$.*

We know that only when $\ell \not\preccurlyeq \ell_A$, the individual runs can produce (i, ℓ, v) because the two runs diverge only when branching on pairs. Similarly, $(\ell, \langle v_1 \,|\, v_2 \rangle)$ can only be produced if $\ell \not\preccurlyeq \ell_A$, because pairs can only be typed if each individual interval in the pair is high and rule P-Out makes sure ℓ is lower than or equal to the pair's interval. We prove a simple lemma that establishes that given a well-typed trace of actions $\mathbb{T}$, $\vdash \lfloor \mathbb{T} \rfloor_1 \approx_{\ell_A} \lfloor \mathbb{T} \rfloor_2$. By combining

Labels and intervals:

$$\frac{}{\ell \sqsubseteq ?} \qquad \frac{}{\ell \sqsubseteq \ell} \qquad \frac{\ell_1' \preccurlyeq \ell_1 \qquad \ell_2 \preccurlyeq \ell_2'}{[\ell_1, \ell_2] \sqsubseteq [\ell_1', \ell_2']}$$

Expressions:

$$\frac{\iota_1 \sqsubseteq \iota_2 \qquad g_1 \sqsubseteq g_2}{(\iota_1\ u)^{g_1} \sqsubseteq (\iota_2\ u)^{g_2}} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{e_1\ \mathsf{bop}\ e_2 \sqsubseteq e_1'\ \mathsf{bop}\ e_2'}$$

$$\frac{E \sqsubseteq E' \qquad g_1 \sqsubseteq g_2 \qquad e_1 \sqsubseteq e_2}{E_{g_1} e_1 \sqsubseteq E'_{g_2} e_2}$$

Commands:

$$\frac{}{\mathsf{skip} \sqsubseteq \mathsf{skip}} \qquad \frac{c_1 \sqsubseteq c_1' \qquad c_2 \sqsubseteq c_2'}{c_1; c_2 \sqsubseteq c_1'; c_2'}$$

$$\frac{e_1 \sqsubseteq e_2}{x := e_1 \sqsubseteq x := e_2} \qquad \frac{e_1 \sqsubseteq e_2}{\mathsf{output}(\ell, e_1) \sqsubseteq \mathsf{output}(\ell, e_2)}$$

$$\frac{e_1 \sqsubseteq e_2 \qquad c_1 \sqsubseteq c_1' \qquad c_2 \sqsubseteq c_2'}{\mathsf{if}^X\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \sqsubseteq \mathsf{if}^X\ e_2\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'}$$

$$\frac{e_1 \sqsubseteq e_2 \qquad c_1 \sqsubseteq c_2}{\mathsf{while}^X\ e_1\ \mathsf{do}\ c_1 \sqsubseteq \mathsf{while}^X\ e_2\ \mathsf{do}\ c_2}$$

Store, Types and Typing-context:

$$\frac{g \sqsubseteq g'}{\tau^g \sqsubseteq \tau^{g'}} \qquad \frac{\forall x \in \Gamma.\ \Gamma(x) \sqsubseteq \Gamma'(x)}{\Gamma \sqsubseteq \Gamma'}$$

PC-stack and Configurations:

$$\frac{}{\emptyset \sqsubseteq \emptyset} \qquad \frac{\iota \sqsubseteq \iota' \qquad g \sqsubseteq g'}{\iota\ g \sqsubseteq \iota'\ g'} \qquad \frac{\kappa_1 \sqsubseteq \kappa_1' \qquad \kappa_2 \sqsubseteq \kappa_2'}{\kappa_1 \rhd \kappa_2 \sqsubseteq \kappa_1' \rhd \kappa_2'}$$

$$\frac{\forall x \in \delta.\ \delta(x) \sqsubseteq \delta'(x)}{\delta \sqsubseteq \delta'} \qquad \frac{\kappa \sqsubseteq \kappa' \qquad \delta \sqsubseteq \delta' \qquad c \sqsubseteq c'}{\kappa, \delta\ /\ c \sqsubseteq \kappa', \delta'\ /\ c'}$$

Fig. 14. Precision relations

the Preservation, Soundness, and Completeness Theorems, it follows that our gradual type system satisfies termination-insensitive noninterference.

VI. Gradual Guarantees

The gradual guarantees state that if a program with more precise labels type-checks and is accepted by the runtime semantics of the gradual type system, then the same program with less precise labels is also accepted by the gradual type system. To establish these guarantees, we define a precision relation between labels, expressions, and commands. The precision relations are shown in Fig. 14.

Our type system with gradual labels satisfies the static gradual guarantee. The dynamic gradual guarantee is also ensured by our calculus, i.e., if a command takes a step under a store and pc stack, then a less precise command can also take a step under a less precise store and pc stack.

Theorem 5 (Static Guarantee). *If* $\Gamma_1; g_1 \vdash c_1$, $\Gamma_1 \sqsubseteq \Gamma_2$, $g_1 \sqsubseteq g_2$, *and* $c_1 \sqsubseteq c_2$, *then* $\Gamma_2; g_2 \vdash c_2$.

Theorem 6 (Dynamic Guarantee). *If* $\kappa_1, \delta_1 \ / \ c_1 \xrightarrow{\alpha_1} \kappa_1', \delta_1' \ / \ c_1'$ *and* $\kappa_1, \delta_1 \ / \ c_1 \sqsubseteq \kappa_2, \delta_2 \ / \ c_2$, *then* $\kappa_2, \delta_2 \ / \ c_2 \xrightarrow{\alpha_2} \kappa_2', \delta_2' \ / \ c_2'$ *such that* $\kappa_1', \delta_1' \ / \ c_1' \sqsubseteq \kappa_2', \delta_2' \ / \ c_2'$ *and* $\alpha_1 = \alpha_2$.

VII. Discussion

Monitor comparison: Our monitor is a hybrid monitor. The differences between our monitor and traditional hybrid monitors (c.f., [13], [17], [21]–[23]) are that (1), we update the memory before executing the branch, while other hybrid monitors update the labels at the merge points; (2), our monitor will not upgrade variables with fixed static labels (the monitor will abort) and only refine label intervals for dynamically labeled variables. Fig. 15 highlights the differences between our monitor and other information flow monitors using the example program in Listing 4 and a two point lattice ($L \preccurlyeq H$). We show two cases for our monitor, differing in y's security label (?, or L). Note that hybrid monitors including ours have the same behavior regardless of x's value. When y has a dynamic label, our monitor is more precise than NSU, as precise as permissive-upgrade [12], and less precise than a traditional hybrid monitor [13]. Ours can be as precise as a traditional hybrid monitor if z's initial value is $([L, H]\mathsf{true})^?$. This would allow us to refine z's label interval, and only abort at the output. When y has a static label, or an effectively static label (i.e., $([L, L]\mathsf{true})^L$), our monitor is the least precise and will abort at the first branch. This rigidity is due to our decision to not update variables' label-intervals, except by refinement.

Implicit leaks manifested in noninterference proofs: Let's revisit the example at the end of Section IV-C to see how the implicit leak manifests in the paired execution and why it leads to our current design; where we do not use the write sets in the if statements and simply refine the label-intervals. Because x is H, it's initialized with a paired value.

$$\begin{array}{rcl} \delta & = & x \mapsto \langle [H, H]\mathsf{true}^H \mid [H, H]\mathsf{false}^H \rangle, \\ & & y \mapsto [L, H]\mathsf{true}^?, \ z \mapsto [L, L]\mathsf{true}^L \\ c_1 & = & \mathsf{if}\ x\ \mathsf{then}\ y := [L, H]\mathsf{false}^?\ \mathsf{else\ skip} \\ c_2 & = & \mathsf{if}\ y\ \mathsf{then}\ z := [L, L]\mathsf{false}^L\ \mathsf{else\ skip} \end{array}$$

$$\begin{array}{ll} [L, L]\ L,\ \delta & / \ c_1; c_2 \longrightarrow \\ \quad\cdots & / \ \mathsf{if}\ \langle [H, H]\mathsf{true}^H \mid [H, H]\mathsf{false}^H \rangle\ \mathsf{then} \\ & \qquad y := [L, H]\mathsf{false}^?\ \mathsf{else\ skip}; c_2 \longrightarrow \\ \quad\cdots & / \ \langle \cdots, y := [L, H]\mathsf{false}^?; c_2 \mid \cdots, \mathsf{skip}; c_2 \rangle \longrightarrow \end{array}$$

The variable y is updated only in the left branch. To prove soundness and completeness of the paired semantics, the two executions should be independent. Therefore, we try to update y in the store as $\langle [H, H]\mathsf{false} \mid [L, H]\mathsf{true} \rangle^?$. However, this pair is not well-formed because pairs are only well-typed if both intervals are in H. Clearly, the right branch of y does not satisfy this requirement. Therefore, we cannot prove preservation for the assignment case. For preservation to succeed, we would need to refine the right branch to be $[H, H]\mathsf{true}$ when assigning to y in the left execution. But then the two executions are no longer independent, which breaks soundness (i.e., the projected execution is not guaranteed to make progress or stay in the same state).

With these constraints in place, we need to refine y before the branch, which ultimately leads to our final design.

VIII. Related Work

Static Information Flow Type Systems: Quite a few type-systems have been proposed to statically enforce noninterference by annotating variables with labels. Volpano et al. [1] present the first type-system with information flow labels that satisfies a variant of noninterference, also known as termination-insensitive noninterference. If all variables are annotated with concrete security labels, our type system behaves the same as a flow-insensitive information flow type system. Being a gradual type system, we can additionally accept programs with no security labels and enforce termination-insensitive noninterference at runtime. Our formalization borrows the proof-technique from FlowML, presented by Pottier and Simonet [18], for enforcing noninterference using pairs.

Static type systems that resemble gradual typing: JFlow [24] (and later Jif) includes polymorphic labels, for which programmers can specify the upper bound of a polymorphic label. Polymorphic labels are essentially (bounded) universally quantified labels and these labels are instantiated by concrete labels at runtime. There is no label refinement associated with polymorphic labels. Jif also allows run-time labels (also called dynamic labels). These are runtime representation of label objects that users can generate and perform tests on. This is not to be confused with the dynamic label (?) in gradual typing. Runtime labels do not mean unknown security labels, nor are they refined at runtime. Finally, label inference is a widely used compile-time algorithm to reduce programmers' annotation burdens. A flow-insensitive type system with the most powerful inference algorithm is less permission than our system. Ill-typed programs, rejected by a type system, can be accepted by our type system. Their safety is ensured by our runtime monitors.

Purely dynamic monitors: Dynamic approaches use a runtime monitor to track the flow of information through the program. The labels are mostly flow-sensitive in nature. Austin and Flanagan [11] present a purely dynamic information flow monitoring approach that disallows assignments to public values in secret contexts. Our monitor semantics follows a similar approach to prevent information leaks at runtime. Subsequent work presents approaches to make the analysis more permissive and amenable to dynamic languages [12], [25]–[27]. A recent paper shows the equivalence between coarse-grained and fined-grained dynamic monitors [28]. Detailed comparisons between our monitor and dynamic monitors can be found in Fig. 15.

$y = \mathsf{true}^L$, $z = \mathsf{true}^L$	$x = \mathsf{false}^H$	$x = \mathsf{true}^H$			$\mathbb{WHILE}^{\mathsf{G}}_{\mathsf{Evd}}$	
Program	NSU/Permissive	NSU	Permissive	Hybrid	$y = [L, H]\mathsf{true}^?$	$y = [L, L]\mathsf{true}^L$
$\mathsf{if}\ x$	branch not taken	branch taken	branch taken		$y \uparrow [H, H]$	try $y \uparrow [H, H]$, abort
$\mathsf{then}\ y := \mathsf{false}^L$		$pc = H$, abort	$y \uparrow P$	$y \uparrow H$		
$\mathsf{if}\ y$	branch taken		abort		try $z \uparrow [H, H]$, abort	
$\mathsf{then}\ z := \mathsf{false}^L$				$z \uparrow H$		
$\mathsf{output}(L, z)$	$\mathsf{output}(L, \mathsf{false})$			abort		

Fig. 15. Comparison of monitor behavior. $y \uparrow \ell$ denotes monitor's attempt to update y's label (interval).

Hybrid monitors: To leverage the benefits of static and dynamic approaches for precision and permissiveness, researchers have also proposed hybrid approaches to enforce noninterference [13], [17], [21]–[23], [29]–[31]. We demonstrate that the hybrid monitoring approach is suitable for generating runtime behavior of gradual types that rely on refining label intervals. Gradual typing has the added benefit of allowing programmers to reject ill-typed programs. As shown in Fig. 15, our monitor aborts earlier than a typical hybrid monitor because of the lack of support for label updating. We support termination-insensitive noninterference, while others support progress-sensitive noninterference [22], [31].

Gradual information flow type systems: More closely related to our work are works on gradual security types. Disney and Flanagan [3] study gradual security types for a pure lambda calculus, and Fennell and Thiemann [4] present a gradual type system for a calculus with ML-style references. However, these works are based on adding explicit programmer-provided checks and casts to the code. Fennell and Thiemann [5] extend their prior work to object-oriented programs in a flow-sensitive setting for a Java-like language. They use a hybrid approach to perform effect analysis that upgrades the labels of variables similar to the write set used in our analysis. At runtime, these systems cast the dynamic label to a fixed label, rather than a set of possible labels, and the monitors updates labels of memory locations, which we do not do. On the other hand, our approach has fixed gradual labels and refines only the label-intervals associated with the value to satisfy dynamic gradual guarantee. More recently, Toro et al. [6] presented a type-driven gradual type system for a higher-order language with references based on abstract gradual typing [9]. Their formalization satisfies the static gradual guarantee, but sacrifices the dynamic gradual guarantee for noninterference. They briefly discuss the idea of using hybrid approaches and faceted evaluation for regaining the dynamic gradual guarantee. The language presented in this paper is simpler than their language but has mutable global variables and hence, a similar issue with proving noninterference while satisfying the dynamic gradual guarantee.

GLIO [7] presents another interpretation of gradual information flow types that enjoys both noninterference and gradual guarantees. *GLIO* is the most expressive among the above-mentioned projects; it includes higher-order functions, general references, coarse-grained information flow control, and first-class labels. *GLIO*'s monitor decides the concrete label for a dynamically labeled reference at allocation time. While avoiding problems stemmed from refining label intervals, the concrete label results in a less permissive approach.

Our work explores yet another design space of gradual information flow types and highlights the necessity of a hybrid approach for a system that refines label intervals to ensure both noninterference and the gradual guarantees. Extending our static type system to include higher-order functions and references would require a precise static analysis to determine the write set accurately, as pointed out by prior work [17]. This is common for hybrid approaches. For instance, LJGS [5] uses a sophisticated points-to analysis. Moore and Chong have identified sufficient conditions for safely incorporating memory abstractions and static analyses into a hybrid information-flow monitor [17]. An interesting future direction is to investigate such conditions and abstractions for a higher-order language. Another possible direction for handling languages with first-class functions and references can be using the ideas proposed by Nielson et al. [32] and Foster et al. [33], who use regions and side-effect analysis to determine aliases.

IX. Conclusion

We presented a gradual information flow type system for a simple imperative language that enforces termination-insensitive noninterference and ensures the gradual guarantee at the same time. We demonstrated that our hybrid monitor can stop implicit flows by refining the labels for references in the write-sets of both branches, regardless of which branch is taken. The non-conventional proof technique of noninterference that we used helps us identify the conditions for ensuring the gradual guarantees.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Science Foundation via grant CNS1704542 and the CyLab Presidential Fellowship at Carnegie Mellon University.

References

[1] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996.

[2] J. G. Siek and W. Taha, "Gradual typing for functional languages," in *In Scheme And Functional Programming Workshop*, 2006, pp. 81–92.

[3] T. Disney and C. Flanagan, "Gradual information flow typing," in *Proceedings of the 2nd International Workshop on Scripts to Programs Evolution*, 2011.

[4] L. Fennell and P. Thiemann, "Gradual security typing with references," in *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, 2013, pp. 224–239.

[5] ——, "LJGS: gradual security types for object-oriented languages," in *30th European Conference on Object-Oriented Programming*, 2016, pp. 9:1–9:26.

[6] M. Toro, R. Garcia, and E. Tanter, "Type-driven gradual security with references," *ACM Trans. Program. Lang. Syst.*, vol. 40, no. 4, pp. 16:1–16:55, Dec. 2018.

[7] A. A. de Amorim, M. Fredrikson, and L. Jia, "Reconciling noninterference and gradual typing," in *Proceedings of the 35th ACM/IEEE Symposium on Logic in Computer Science*, 2020, pp. 116–129.

[8] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland, "Refined Criteria for Gradual Typing," in *1st Summit on Advances in Programming Languages*, ser. Leibniz International Proceedings in Informatics, vol. 32, 2015, pp. 274–293.

[9] R. Garcia, A. M. Clark, and E. Tanter, "Abstracting gradual typing," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 429–442.

[10] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[11] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009, pp. 113–124.

[12] ——, "Permissive dynamic information flow analysis," in *Proc. 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2010, pp. 3:1–3:12.

[13] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, 2010, pp. 186–199.

[14] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 79–90.

[15] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Proceedings 10th Computer Security Foundations Workshop*, June 1997, pp. 156–168.

[16] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Proceedings of the 12th International Conference on Static Analysis*, 2005, pp. 352–367.

[17] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, 2011, pp. 146–160.

[18] F. Pottier and V. Simonet, "Information flow inference for ML," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 319–330.

[19] A. Bichhawat, M. McCall, and L. Jia, "First-order gradual information flow types and gradual guarantees," https://arxiv.org/abs/2003.12819, 2021.

[20] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[21] D. Hedin, L. Bello, and A. Sabelfeld, "Value-sensitive hybrid information control for a JavaScript-like language," in *Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium*, 2015, pp. 351–365.

[22] A. Bedford, S. Chong, J. Desharnais, E. Kozyri, and N. Tawbi, "A progress-sensitive flow-sensitive inlined information-flow control monitor," *Computers & Security*, vol. 71, pp. 114 – 131, 2017.

[23] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in Haskell," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 289–301.

[24] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, pp. 228–241.

[25] D. Hedin and A. Sabelfeld, "Information-flow security for a core of JavaScript," in *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, 2012, pp. 3–18.

[26] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Generalizing permissive-upgrade in dynamic information flow analysis," in *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, 2014, pp. 15:15–15:24.

[27] J. F. Santos and T. Rezk, "An information flow monitor-inlining compiler for securing a core of JavaScript," in *ICT Systems Security and Privacy Protection*, 2014, pp. 278–292.

[28] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, "From fine- to coarse-grained dynamic information flow control and back," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[29] D. Chandra and M. Franz, "Fine-grained information flow analysis and enforcement in a Java virtual machine," in *23rd Annual Computer Security Applications Conference*, 2007, pp. 463–475.

[30] S. Just, A. Cleary, B. Shirley, and C. Hammer, "Information flow analysis for JavaScript," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, 2011, pp. 9–18.

[31] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *2015 IEEE 28th Computer Security Foundations Symposium*, 2015, pp. 137–151.

[32] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Heidelberg: Springer-Verlag, 1999.

[33] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 1–12.

APPENDIX

A. Well-formedness

1) $\vdash v$ wf, if
   a) $v = \langle v_1 \mid v_2 \rangle$, then $\forall i \in \{1, 2\}.v_i = (\iota_i\ u_i)$
   b) $v = (\iota\ u)^g$
2) $\vdash \delta$ wf, if $\forall x \in \delta, \vdash \delta(x)$ wf
3) $\vdash \delta\ /_i\ e$ wf for $i \in \{\cdot, 1, 2\}$ if $\vdash \delta$ wf
4) $\vdash c$ wf, when the following hold:
   a) if $c = \langle \kappa_1, \iota_1, c_1 \mid \kappa_2, \iota_2, c_2 \rangle_g$, then $\vdash c_1$ wf, $\vdash c_2$ wf, and c_1 and c_2 do not contain pairs
   b) if $c =$ if e then c_1 else c_2, then $\vdash c_1$ wf, $\vdash c_2$ wf, and c_1 and c_2 do not contain pairs or braces

$$\boxed{\kappa, \delta \;/\; c \xrightarrow{\alpha} \kappa', \delta' \;/\; c'}$$

$$\frac{\kappa, \delta \;/\; c_1 \longrightarrow \mathsf{abort}}{\kappa, \delta \;/\; c_1; c_2 \longrightarrow \mathsf{abort}}\ \textsc{M-Seq-Err}$$

$$\frac{\begin{array}{c}\delta \;/\; e \Downarrow v \\ \mathit{rfL}(\delta, X, \iota_{pc} \curlyvee \mathit{intvl}\,(v)) = \mathtt{undef}\end{array}}{\iota_{pc}\ g_{pc}, \delta \;/\; \mathsf{if}^X\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \longrightarrow \mathsf{abort}}\ \textsc{M-If-Refine-Err}$$

$$\frac{\delta \;/\; e \Downarrow v \qquad \mathit{refineLB}(\iota_{pc}, v) = \mathtt{undef}}{\iota_{pc}\ g_{pc}, \delta \;/\; x := e \longrightarrow \mathsf{abort}}\ \textsc{M-Assign-Err}$$

$$\frac{\begin{array}{c}\delta \;/\; e \Downarrow v \qquad v' = \mathit{refineLB}(\iota_{pc}, v) \\ \mathsf{updL}(\mathit{intvl}\,(\delta(x)), v') = \mathtt{undef}\end{array}}{\iota_{pc}\ g_{pc}, \delta \;/\; x := e \longrightarrow \mathsf{abort}}\ \textsc{M-Assign-Err2}$$

$$\frac{\delta \;/\; e \Downarrow v \qquad \mathit{refineLB}(\iota_{pc}, v) = \mathtt{undef}}{\iota_{pc}\ g_{pc}, \delta \;/\; \mathsf{output}(\ell, e) \longrightarrow \mathsf{abort}}\ \textsc{M-Out-Err}$$

$$\frac{\begin{array}{c}\delta \;/\; e \Downarrow v \qquad \mathit{refineLB}(\iota_{pc}, v) = v' \\ \mathsf{updL}([\ell, \ell], v') = \mathtt{undef}\end{array}}{\iota_{pc}\ g_{pc}, \delta \;/\; \mathsf{output}(\ell, e) \longrightarrow \mathsf{abort}}\ \textsc{M-Out-Err2}$$

Fig. 16. Monitor abort cases for commands

Read operations:

$$\mathsf{rd}\ v = v \qquad \mathsf{rd}_1\ v = \lfloor v \rfloor_1 \qquad \mathsf{rd}_2\ v = \lfloor v \rfloor_2$$

Write operations:

$$\frac{\begin{array}{c} v_n = \langle \iota_1\ u_1 \,|\, \iota_2\ u_2 \rangle^g \\ \forall i \in 1,2, \mathit{restrictLB}(\iota_o, \iota_i) = \iota'_i \end{array}}{\mathsf{updL}\ \iota_o\ v_n = \langle \iota'_1\ u_1 \,|\, \iota'_2\ u_2 \rangle^g}$$

$$\frac{\mathit{restrictLB}(\iota_o, \iota_n) = \iota}{\mathsf{updL}\ \iota_o\ (\iota_n\ u_n)^g = (\iota\ u_n)^g}$$

$$\frac{\begin{array}{c} v_n = \langle \iota_1\ u_1 \,|\, \iota_2\ u_2 \rangle^g \\ \exists j \in [1,2], \mathit{restrictLB}(\iota_o, \iota_j) = \mathtt{undef} \end{array}}{\mathsf{updL}\ \iota_o\ v_n = \mathtt{undef}}$$

$$\frac{\mathit{restrictLB}(\iota_o, \iota_n) = \mathtt{undef}}{\mathsf{updL}\ \iota_o\ (\iota_n\ u_n)^g = \mathtt{undef}}$$

c) if $c = \ \mathsf{while}^X\ e\ \mathsf{do}\ c$, then $\vdash c$ wf and c does not contain pairs or braces
d) if $c = \ c_1; c_2$ then $\vdash c_1$ wf, $\vdash c_2$ wf and c_2 does not contain pairs or braces
e) if $c = \{c_1\}$, then $\vdash c_1$ wf

5) $\vdash \kappa, \delta\ /_i\ c$ wf for $i \in \{\cdot, 1, 2\}$) if all of the following hold
   a) $\vdash c$ wf and $\vdash \delta$ wf
   b) if $i \in \{1,2\}$, then c does not contain pairs

B. Additional definitions

We define the following constraints on configurations to facilitate proofs related to paired values and commands. We start by defining $\iota \in H(\ell_A)$, $\Pi \in H(\ell_A)$ and $\kappa \in H(\ell_A)$ for any observer at level ℓ_A.

$$\frac{\begin{array}{c} \iota = [\ell_l, \ell_r] \\ \ell_l \not\preccurlyeq \ell_A \end{array}}{\iota \in H(\ell_A)}\ \iota\text{-H} \qquad \frac{\begin{array}{c} \Pi = \langle \iota_1 \,|\, \iota_2 \rangle \\ \iota_i \in H(\ell_A)\ ,\ i \in \{1,2\} \end{array}}{\Pi \in H(\ell_A)}\ \Pi\text{-H}$$

$$\frac{\begin{array}{c} \kappa = \iota\ g \rhd \kappa' \\ \iota \in H(\ell_A) \qquad (\kappa' \in H(\ell_A) \vee \kappa' = \emptyset) \end{array}}{\kappa \in H(\ell_A)}\ \kappa\text{-H}$$

We say a configuration is safe (written $\vdash \kappa, \delta\ /_i\ c$ sf for $i \in \{\cdot, 1, 2\}$) if all of the following hold

1) if $i \in \{1,2\}$, then $\kappa \in H(\ell_A)$, $\forall x \in \mathit{WtSet}(c)$, $\mathit{intvl}\,(\delta(x)) \in H(\ell_A)$
2) if $c = \mathsf{if}\ \langle \_ \,|\, \_ \rangle$ $\mathsf{then}\ c_1\ \mathsf{else}\ c_2$, then $\forall x \in \mathit{WtSet}(c)$, $\mathit{intvl}\,(\delta(x)) \in H(\ell_A)$
3) if $c = \langle \kappa_1, \iota_1, c_1 \,|\, \kappa_2, \iota_2, c_2 \rangle_g$, then $\forall i \in \{1,2\}$, $\iota_i \vdash g \in H(\ell_A)$, and $\forall x \in \mathit{WtSet}(c)$, $\mathit{intvl}\,(\delta(x)) \in H(\ell_A)$

We first define when a gradual label of an initial store location is not observable by the attacker. Formally: $g \in H(\ell_A)$ iff $g = \ell$ and $\ell \not\preccurlyeq \ell_A$.

Simple write/output operations:

$$\frac{\begin{array}{c} \mathit{intvl}\,(v_o) = \langle \iota_1 \,|\, \iota_2 \rangle \text{ or } v_n = \langle \iota'_1\ u_1 \,|\, \iota'_2\ u_2 \rangle^g \\ \forall i \in 1,2,\ \ \mathit{restrictLB}(\lfloor \mathit{intvl}\,(v_o) \rfloor_i, \lfloor \mathit{intvl}\,(v_n) \rfloor_i) = \iota''_i \end{array}}{\mathsf{upd}\ v_o\ v_n = \langle \iota''_1\ u_1 \,|\, \iota''_2\ u_2 \rangle^g}$$

$$\frac{\begin{array}{c} \mathit{intvl}\,(v_o) = \langle \iota_1 \,|\, \iota_2 \rangle \text{ or } v_n = \langle \iota'_1\ u_1 \,|\, \iota'_2\ u_2 \rangle^g \\ \exists i \in 1,2, \mathit{restrictLB}(\lfloor \mathit{intvl}\,(v_o) \rfloor_i, \lfloor \mathit{intvl}\,(v_n) \rfloor_i) = \mathtt{undef} \end{array}}{\mathsf{upd}\ v_o\ v_n = \mathtt{undef}}$$

$$\frac{\mathit{restrictLB}(\iota_o, \iota_n) = \iota}{\mathsf{upd}\ (\iota_o\ u_o)^g\ (\iota_n\ u_n)^g = (\iota\ u_n)^g}$$

$$\frac{\mathit{restrictLB}(\iota_o, \iota_n) = \mathtt{undef}}{\mathsf{upd}\ (\iota_o\ u_o)^g\ (\iota_n\ u_n)^g = \mathtt{undef}}$$

$$\frac{\begin{array}{c} \lfloor v_n \rfloor_1 = (\iota_1\ u_n)^{g'} \qquad \mathit{restrictLB}(\lfloor \mathit{intvl}\,(v_o) \rfloor_1, \iota_1) = \iota'_1 \\ \lfloor v_o \rfloor_2 = (\iota_2\ u_2)^g \end{array}}{\mathsf{upd}_1\ v_o\ (\iota_n\ u_n)^g = \langle \iota'_1\ u_n \,|\, \iota_2\ u_2 \rangle^g}$$

$$\frac{\begin{array}{c} \lfloor v_n \rfloor_2 = (\iota_2\ u_n)^{g'} \qquad \mathit{restrictLB}(\lfloor \mathit{intvl}\,(v_o) \rfloor_2, \iota_2) = \iota'_2 \\ \lfloor v_o \rfloor_1 = (\iota_1\ u_1)^g \end{array}}{\mathsf{upd}_2\ v_o\ (\iota_n\ u_n)^g = \langle \iota_1\ u_1 \,|\, \iota'_2\ u_n \rangle^g}$$

$$\frac{\lfloor v_n \rfloor_i = (\iota_i\ u_n)^{g'} \qquad \mathit{restrictLB}(\lfloor \mathit{intvl}\,(v_o) \rfloor_i, \iota_i) = \mathtt{undef}}{\mathsf{upd}_i\ v_o\ (\iota_n\ u_n)^g = \mathtt{undef}}$$

Fig. 17. Operations with pairs

$$\boxed{\kappa, \delta\ /_i\ c \xrightarrow{\alpha} \kappa', \delta'\ /_i\ c'}$$

$$\frac{\begin{array}{c} \kappa_i \rhd \iota_{pc} \curlyvee \iota_i\ g_{pc} \curlyvee_c g, \delta\ /_i\ c_i \longrightarrow \mathsf{abort} \\ \{i, j\} = \{1, 2\} \end{array}}{\iota_{pc}\ g_{pc}, \delta\ /\ \langle \kappa_1, \iota_1, c_1 \,|\, \kappa_2, \iota_2, c_2 \rangle_g \longrightarrow \mathsf{abort}}\ \textsc{P-C-Pair-Err}$$

Fig. 18. Paired executions abort

We define merging of two stores ($\delta_1 \bowtie \delta_2$) as below:

$$\frac{}{\Gamma \vdash \cdot \bowtie \cdot = \cdot}\ \textsc{MgS-Emp}$$

$$\frac{\begin{array}{c} \Gamma \vdash \delta_1 \bowtie \delta_2 = \delta \\ \mathit{lab}(\Gamma(x)) \in H(\ell_A) \qquad v_i = (\iota_i\ u_i)^g (i \in \{1,2\} \end{array}}{\Gamma \vdash \delta_1, x \mapsto v_1 \bowtie \delta_2, x \mapsto v_2 = \delta, x \mapsto \langle \iota_1\ u_1 \,|\, \iota_2\ u_2 \rangle^g}\ \textsc{MgS-H}$$

$$\frac{\begin{array}{c} \Gamma \vdash \delta_1 \bowtie \delta_2 = \delta \\ \mathit{lab}(\Gamma(x)) \notin H(\ell_A) \qquad v_1 = v_2 = v \end{array}}{\Gamma \vdash \delta_1, x \mapsto v_1 \bowtie \delta_2, x \mapsto v_2 = \delta, x \mapsto v}\ \textsc{MgS-L}$$