Hybrid Information Flow Control for Low-level Code

Eduardo Geraldo[1], José Fragoso Santos[2], and João Costa Seco[1]

[1] NOVA LINCS - NOVA University Lisbon, Portugal
[2] Instituto Superior Técnico & INESC-ID, Portugal

Abstract. Failure to ensure data confidentiality can have a significant financial and reputational impact on companies. To aggravate the issue, frequently used methods like testing are insufficient when proving data confidentiality in software systems. Existing information flow based approaches require heavy implementation and specification efforts or lack the expressiveness programmers desire. To tackle the issues, we propose a novel hybrid system for information flow control in low-level languages. By combining an information flow monitor with a type system that instruments programs with runtime security checks, we support value-dependent security types in a low-level setting. We formalise our type system and monitor using a TAL-like calculus and prove that they guarantee termination-insensitive non-interference. We present the first hybrid type system for information flow control. We also introduce the first hybrid mechanism for a low-level intermediate representation.

1 Introduction

The increasing access to internet-based services and the information they store leads to growing concerns over topics like data confidentiality. Users and regulatory entities expect companies to safeguard the data their systems store and process; failure to do so can have severe financial and reputational consequences such as fines and the loss of users.

Unfortunately, data protection is no easy task, with systems often exhibiting logic flaws or programming mistakes resulting in information leaks. To ensure systems' correctness, developers usually rely on testing, which requires complex test scenarios and careful I/O monitoring; hence test-based information leak detection is an intricate, error-prone process. Heavily tested systems like Github [9], Instagram [39], Facebook [32], and Twitter [1] registered incidents where they wrote unciphered user passwords to system logs.

Information flow control [11,26,34,38] (IFC) is the best-suited technique for information leak detection. The key idea behind it is to tag information and its receptacles (e.g., variables or I/O channels) with security labels arranged in a security lattice. Then, we track all data-processing operations and compute the security level of each datum, stopping information of a given security level from flowing to receptacles of a lower security level. IFC aims to ensure the non-interference property [34], stating that secret inputs should not cause visible

changes to public outputs; a sufficient condition to guarantee the absence of information leaks.

Mechanisms for IFC may enforce flow policies statically or dynamically. Static mechanisms [7,26,31], type systems for information flow control, detect leaks at compile-time with no impact on the runtime. However, they may reject correct programs, e.g. programs with leaks in unreachable code. Contrarily, dynamic mechanisms [4], based on reference monitors, are less prone to reject correct programs but introduce overhead at runtime; monitors have to oversee the execution of every instruction. Plus, they need a high testing coverage to be useful.

Hybrid approaches [38], combine a reference monitor for enacting flow policies at runtime and a type system for static IFC. The type system does most of the analysis, while the monitor only performs checks where necessary, i.e., code that is not possible to statically prove correct nor incorrect. This methodology avoids false positives while keeping the impact of the monitor to a minimum, meaning less overhead and fewer tests required to certify dynamically verified code.

In this paper, we present a formal system for hybrid value-dependent information flow control in Snitch IR, a small imperative low-level language inspired by TAL [29]. The language relates to low-level languages like JVM bytecode [25], the Common Intermediate Language [13], or the LLVM's intermediate language [23]. Value-dependent security labels [26] allow for more expressive security policies.

We introduce a static semantics that verifies and rewrites programs, embedding them with a flow monitor to overview, when necessary, data transfer operations. As an approach to hybrid IFC, it eases specification efforts as it supports incomplete specifications; defers decisions on unknown labels to runtime. We are working on a prototype tool, SNITCH, to detect leaks through the hybrid analysis of JVM bytecode (currently capable of fully dynamic IFC).

We start this paper with the related work in Section 2 and some key concepts in Section 3. Then, we present a sound dynamic semantics that preserves termination insensitive non-interference in Section 4. Next, in Section 5, we introduce type system for hybrid value-dependent information flow control. Lastly, we end with some final comments and future directions for this work in Section 6.

2 Related Work

There is a vast body of works on the application of information flow research to real-world programming languages, ranging from type systems for strongly typed languages such as OCaml [37], and Java [8], to dynamic analysis for scripting languages, such as JavaScript [21] and Python [19]. For a more thorough overview, we refer the reader to [34] and [22]. Here, we focus on IFC for low-level languages, hybrid analysis for IFC, and expressive information flow types.

Hybrid Information Flow Control Hybrid systems for IFC combine static analyses with different flavours of runtime monitoring. Most hybrid IFC analyses are based on gradual typing [36]. In fact, gradual information-flow type systems exist for lambda calculus [12,15] and a lightweight Java-like language [16]. These

type systems allow for polymorphic security labels, providing annotations for denoting statically unknown labels. The programmer must add runtime casts in code points where values of a pre-determined security type are expected. While the static type system guarantees adherence to the specified policy on the static side of a cast, the runtime analysis checks the policy on the dynamic side.

Amongst the existing hybrid IFC approaches, the most closely related to ours is [18], which introduces a hybrid system for IFC in a fragment of JavaScript. It combines a type system with a no-sensitive-upgrade monitor to instrument programs so that the monitor only performs the necessary checks. Our work, however, faces specific challenges related to the low-level nature of Snitch IR; most notably, the precise tracking of implicit flows in unstructured control flows.

Information Flow Control for Low Level Languages There is a number of works on static and dynamic information flow analyses for low-level languages. Barth et al [7]. were the first to design a type system for IFC in Java bytecode; later proved sound using the Coq proof assistant [6]. Their intermediate representation is similar to ours and make use of the concept of *control dependence regions*.

Aldous et al. [2] designed a static analysis for proving non-interference in a Dalvik-like language. They implemented the analysis and proved it sound. The analysis enriches the control flow graph of target programs with information computed by an abstract interpreter. The authors show that resulting graphs, called *execution point graphs*, can improve the precision of the analysis. They further showed [3] how to derive a sound IF monitor from their original analysis. We believe that we could use execution point graphs to improve the precision of the static component of our system; this is, however, left as future work.

Recently, Balliu et al. [5] showed how to leverage SMT solvers to prove the non-interference of ARMv7 binaries. They demonstrate the applicability of their approach by using it to verify a sophisticated kernel system call handler, combining handwritten assembly code with complex compiler-generated code.

Expressive Security Types Multiple techniques to enhance the expressiveness of security policies have been proposed. For instance, the more flexible decentralised label model by Meyers and Liskov [30] replaces the security lattice with an ownership-based model. An entity can own an information receptacle, and only the entity or those authorized by the entity via read and write sets can read or write to the receptacle. The owner can delegate the control over the read and write sets to other entities through the "acts for" relation, another feature of the decentralised label model. This model is used with dynamic labels in JIF [31].

Also, value dependent approaches to hybrid systems include the runtime use of static analysis, invoked by the reference monitor [20], thus achieving a permissive IFC verification. This approach suits a purely dynamic setting as Javascript, but not low-level languages. We take the traditional approach instead, of having a static analysis establishing the border between guarded and unguarded code.

Purely static verification of flexible policies relies on dependent information security levels in type based information flow control systems [17,27,26]. The encoding of dependent security types is present in liquid information flow

control [33], which also includes the ability to repair detected errors. Value-dependent IFC builds on top of the traditional approach by enhancing lattices with security labels parameterised with runtime values. These labels allow for dynamic security lattices capable of expressing many real-world situations.

We take inspiration in these approaches and propose a hybrid approach, applied to low-level languages, thus making the verification more expressive and applicable in real situations, requiring less annotation work from the developer.

3 Overview

To fend off bug-induced confidentiality breaches that pester information systems, we present a solution based on value-dependent hybrid IFC to detect and prevent information leaks. A system following such an approach exhibits the advantages of static and dynamic information flow control mechanisms while minimising their disadvantages and supports richer finer-grained information flow policies.

Our approach, as other hybrid approaches, relies on a type system and a reference monitor. The type system enforces flow policies and injects the monitor into target programs. Having the type system perform program rewriting bypasses a standalone instrumentation phase and allows for a seamless integration of the monitor based on the static analysis of each instruction.

We base our approach on a small low-level imperative language, depicted in Figure 1 and inspired by work on typed assembly languages [29] and type-based program rewriting [35]. Our language easily relates to existing low-level languages such as the Java bytecode [25], LLVM's intermediate representation [23], and the Common Language Infrastructure (.NET) instructions [13]. Low-level representations have many advantages. For instance, many high-level languages compile to a single low-level one, e.g., Java, Scala, Groovy, Kotlin, and Clojure, all compile to JVM's bytecode; targeting a low-level language allows for tools to support multiple high-level ones. Furthermore, support for low-level languages brings support for compiled programs as long as it is possible to write specifications for them.

Reference monitors require program instrumentation [14] or an execution environment capable of monitoring executions. The latter results in deep cumbersome changes to third-party virtual machines; hard to automate and maintain. Thus, we instrument target programs, depending only on the higher stability of low-level languages.

Our hybrid approach starts with the static verification of the code, whose purpose is two-fold. As previously mentioned, the static analysis performs the static information flow verification, thus rejecting provably wrong programs, and instruments code that cannot be proved wrong. The combination results from the fact that the instrumentation is dependent on the results of the static analysis; statically correct code segments do not require runtime verifications.

Considering the hybrid nature of our approach, we foresee two types of security levels: concrete levels used at runtime and sets of symbolic levels employed

Operands:

Registers	$r ::= r_1 \mid r_2 \mid \ldots \mid r_n$
Literals	$c ::= c_1 \mid c_2 \mid \ldots \mid c_n$
Operands	$v ::= r \mid c$
Block Labels	$\ell ::= \ell_1 \mid \ell_2 \mid \ldots \mid \ell_n$

Annotations:

Security Label	$k ::= k_1 \mid k_2 \mid \ldots \mid k_n$
Label Interval	$\widehat{k} ::= [k_l, k_u],\ k_l \sqsubseteq k_u$
Security Annotation	$a ::= \widehat{k} \mid k$

Instructions:

Unguarded Instructions	$i_u ::= r := v^a \mid r_d := r_s \oplus v^a$
Guarded Instructions	$i_g ::= r :\stackrel{\circ}{=} v^k \mid r_d :\stackrel{\circ}{=} r_s \oplus v^k$
Instructions	$i ::= i_u \mid i_g \mid \texttt{pop}\,\ell$
Final Instructions	$i_f ::= \texttt{jump}\,\ell \mid \texttt{if}\ r\ ,\ell_t\ ,\ell_f$
Sequences	$I ::= i; I \mid i_f$

Fig. 1: SNITCH IR syntax.

in the static analysis. The latter, we abstract using security intervals. Considering SC the set of all security labels and $\rightarrow$ the partial order relation, we define comparisons and the least upper bound between labels as follows: equality ($=$) and its negation ($\neq$) have the usual semantics. Comparison between labels depends exclusively on the flow relation ($\sqsubseteq = \rightarrow$), $\forall a, b.\ a \sqsubseteq b \Leftrightarrow a \rightarrow b$. $\#$ reflects label divergence, $\forall a \in SC.\ \forall b \in SC.\ a \# b \Leftrightarrow a \not\sqsubseteq b \wedge b \not\sqsubseteq a$. $\sqcup$ yields the least upper bound of both arguments. The security classes (SC) together with the partial ordering ($\rightarrow$) and the least upper bound ($\sqcup$) forms the security lattice. We define a security interval A as $[a_L, a_U]$, where $a_L \sqsubseteq a_U$; a more compact notation for defining sets, $[a_L, a_U] = \{k \mid \forall k \in SC.\ a_L \sqsubseteq k \sqsubseteq a_U\}$. We define set divergence, the least upper bound between sets, and set comparison as follows: $A \# B = \forall k_a \in A.\ \forall k_b \in B.\ k_a \# k_b$ $A \sqcup B = \{k \mid \forall k_a \in A.\ \forall k_b \in B.\ k = k_a \sqcup k_b\}$, and

$$A \preccurlyeq B = \begin{cases} \mathit{true} & \forall k_a \in A.\ \forall k_b \in B.\ k_a \sqsubseteq k_b \\ \mathit{false} & \forall k_a \in A.\ \forall k_b \in B.\ k_b \sqsubseteq k_a \vee A \# B \\ \downarrow & \mathit{otherwise} \end{cases}$$

SNITCH IR foresees guarded and unguarded instructions, each having a different runtime behaviour. Guarded instructions require extra checks that unguarded instructions do not. The need for both types of instructions stems from the hybrid nature of our approach. In a fully dynamic setting, we would consider only guarded instructions. However, taking into account the results of the static analysis, it is possible to replace statically correct guarded instructions with their unguarded counterpart. The instruction set of SNITCH IR also includes a pop instruction, a control instruction required for manipulating scope-related data structures as we will show when presenting the runtime semantics.

Value-Dependent Security Labels Although we use plain security labels in the syntax depicted in Figure 1, our work can also be combined with value-dependent security labels as our semantics and soundness proofs are independent of the underlying security labels. In order to extend our system with support for

⟦D-SafeAssignment⟧

$$\frac{}{(\Delta, \gamma, (\ell, pc) :: \sigma, \ell_c : r := v^k; I) \longrightarrow (\Delta, \gamma[r : v^{pc \sqcup k}], (\ell, pc) :: \sigma, \ell_c : I)}$$

⟦D-UnsafeAssignment⟧

$$\frac{pc \sqsubseteq k_d}{(\Delta, \gamma[r : v_d^{k_d}], (\ell, pc) :: \sigma, \ell_c : r :\hat{=} v^k; I) \longrightarrow (\Delta, \gamma[r : v^{pc \sqcup k}], (\ell, pc) :: \sigma, \ell_c : I)}$$

Fig. 2: Dynamic semantics for assignments.

value-dependent labels, we would first need to change the definition of security labels to account for value-dependencies:

$$\begin{array}{ll} \text{Security Label} & k ::= S[\![x]\!] \mid \bot \mid \top \\ \text{Security Classes} & S ::= S_1 \mid S_2 \mid \ldots \mid S_n \end{array}$$

Where S denotes a security class ($S \in SC$). At the static level value-dependent security classes are parametrized with function parameters, while at the dynamic level they are parametrized with their corresponding runtime values. For instance, one could use the label `Student`(id) to denote the security level of information that can only be read by the student with the specified identifier; at runtime, the id would be replaced with the corresponding value for the student identifier.

It is the role of the runtime semantics to bind the static parameters of value-dependent security classes to their corresponding values. More precisely, when interpreting a function call, the function's parameters may be associated with security labels that depend on their corresponding values or on the values of the other parameters. In such cases, the security monitor binds the values of the parameters to the security labels that dependend on them.

4 Dynamic Semantics

We now define the semantics of the information flow monitor for Snitch IR. The monitor handles guarded and unguarded instructions. The former entail runtime checks to avoid information leakage, while the latter only require label propagation. Our monitor ensures non-interference on the guarded fragment but has some runtime overhead, and requires significant testing to achieve high reliability. Thus, we present a type system, in Section 5, to perform program rewriting, adding guards only where necessary; the reference monitor does not verify at runtime statically correct operations.

We define the operational semantics of the monitor by means of a transition system, with the reduction rules of Figure 2 and Figure 3 and using runtime configurations of the form $(\Delta, \gamma, \sigma, \ell_c : I)$. In a configuration, the code heap Δ maps code labels to instruction sequences, the store for register labels γ maps register names to pairs of values and their respective security levels. Finally, the

⟦BranchT-Push⟧

$$\frac{\ell_{pd} = \mathit{postDom}(\ell_c) \quad \ell \neq \ell_{pd} \quad \sigma' = (\ell_{pd}, k \sqcup pc) :: (\ell, pc) :: \sigma}{(\Delta, \gamma[r : \mathit{true}^k], (\ell, pc) :: \sigma, \ell_c : \texttt{if } r \texttt{ , } \ell_t \texttt{ , } \ell_f) \longrightarrow (\Delta, \gamma[r : \mathit{true}^k], \sigma', \ell_t : \texttt{pop}\, \ell_t; \Delta(\ell_t))}$$

⟦BranchF-Push⟧

$$\frac{\ell_{pd} = \mathit{postDom}(\ell_c) \quad \ell \neq \ell_{pd} \quad \sigma' = (\ell_{pd}, k \sqcup pc) :: (\ell, pc) :: \sigma}{(\Delta, \gamma[r : \mathit{false}^k], (\ell, pc) :: \sigma, \ell_c : \texttt{if } r \texttt{ , } \ell_t \texttt{ , } \ell_f) \longrightarrow (\Delta, \gamma[r : \mathit{false}^k], \sigma', \ell_f : \texttt{pop}\, \ell_f; \Delta(\ell_f))}$$

⟦BranchT-NoPush⟧

$$\frac{\ell = \mathit{postDom}(\ell_c) \quad \sigma' = (\ell, k \sqcup pc) :: \sigma}{(\Delta, \gamma[r : \mathit{true}^k], (\ell, pc) :: \sigma, \ell_c : \texttt{if } r \texttt{ , } \ell_t \texttt{ , } \ell_f) \longrightarrow (\Delta, \gamma[r : \mathit{true}^k], \sigma', \ell_t : \texttt{pop}\, \ell_t; \Delta(\ell_t))}$$

⟦BranchF-NoPush⟧

$$\frac{\ell = \mathit{postDom}(\ell_c) \quad \sigma' = (\ell, k \sqcup pc) :: \sigma}{(\Delta, \gamma[r : \mathit{false}^k], (\ell, pc) :: \sigma, \ell_c : \texttt{if } r \texttt{ , } \ell_t \texttt{ , } \ell_f) \longrightarrow (\Delta, \gamma[r : \mathit{false}^k], \sigma', \ell_f : \texttt{pop}\, \ell_f; \Delta(\ell_f))}$$

⟦D-Jump⟧

$$\frac{}{(\Delta, \gamma, \sigma, \ell_c : \texttt{jump}\, \ell) \longrightarrow (\Delta, \gamma, \sigma, \ell : \texttt{pop}\, \ell; \Delta(\ell))}$$

⟦D-Pop⟧

$$\frac{}{(\Delta, \gamma, (\ell, -) :: \sigma, \ell_c : \texttt{pop}\, \ell; I) \longrightarrow (\Delta, \gamma, \sigma, \ell_c : I)}$$

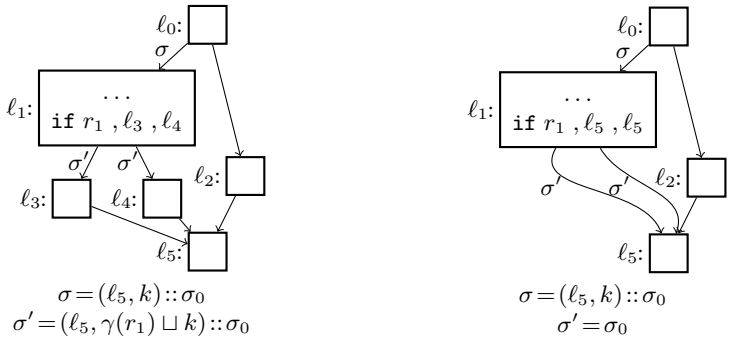
⟦D-NoPop⟧

$$\frac{\ell \neq \ell'}{(\Delta, \gamma, (\ell, -) :: \sigma, \ell_c : \texttt{pop}\, \ell'; I) \longrightarrow (\Delta, \gamma, (\ell, -) :: \sigma, \ell_c : I)}$$

Fig. 3: Dynamic semantics for control instructions.

stack σ tracks the current security level of the computation (pc), maintaining the nesting information needed to implement program scopes, relevant for IFC. We define a stack σ as a list of pairs of the form (ℓ, k), such that ℓ is the first post-dominant (function *postDom*, as used in Figure 3) of the branching instruction adding the entry, and k the security level of the new scope. To denote the security level of the current scope we use pc. Finally, $\ell_c : I$ denotes the current basic block: ℓ_c represents the label of the block, and I the instructions comprising the block.

Assignments The semantics for assignments, depicted in Figure 2, emphasizes the hybrid nature of the information flow monitor, distinguishing guarded ($\hat{=}$) from unguarded ($=$) instructions. The axiom ⟦D-SafeAssignment⟧ defines the behaviour of unguarded assignments. It updates register r with the value v and security level k. Rule ⟦D-UnsafeAssignment⟧, for guarded assignments, follows ⟦D-SafeAssignment⟧ but includes a runtime check; the context's security level (pc) must be smaller than or equal to the security level of the assigned register ($pc \sqsubseteq k_d$). This check is necessary to avoid implicit informations leaks, that is, leaks arising from the control structure of programs. It must not be possible to write to registers visible at a level lower than the context's level, nor should it be



(a) No-push no-pop (b) No-push pop

(c) Push no-pop (d) Push pop

Fig. 4: Conditional jump influence on σ based on the control flow graph.

possible to write a value with a label lower than the context's label; visible registers remain visible and secret registers remain secret. SNITCH IR foresees binary operations ($r_d := r_s \oplus v^a$ and $r_d :\hat{=} r_s \oplus v^k$) not semantically covered here. However, they follow ⟦D-SAFEASSIGNMENT⟧ and ⟦D-UNSAFEASSIGNMENT⟧, and the target register's label depends on the labels of all operands (r_s and v), plus the context's label (pc).

Control flow dependencies SNITCH IR does not address the structure of a program, namely the scoping of conditions' security levels. So, we resort to an auxiliary data structure, a stack, to track such scopes. The security level of a new scope depends on the levels of outer scopes and the security level of the condition starting the scope. To overcome the lack of a well defined structure, we rely on post-dominance analyses [24] of the control flow graph (CFG) to determine the code contained in each scope.

In graph theory, a node a **post-dominates** a node b if every path from b to the exit node must go through a. Considering the graph in Figure 4c, both

ℓ_5 and ℓ_6 are **post-dominators** of ℓ_1, but ℓ_3 and ℓ_4 are not, due to the flow divergence in ℓ_1. Moreover, we say that ℓ_5 is the **first post-dominator** of ℓ_1, since it is the post-dominator closest to ℓ_1, i.e. the node where the flows that diverges at ℓ_1 merge.

Instructions starting a new scope (conditional jumps) push onto the stack a new security level and the label where the scope ends, the first node post-dominating the current instruction. When performing unconditional jumps (and degenerate conditional jumps), we need to check if the scope has ended and recover the previous context security level.

The stack invariant defines that the topmost entry always contains the label where the current context ends, and security levels do not decrease (see [[BranchT-Push]] and [[BranchF-Push]]). However, if a scope terminates in the same place as its parent, a new entry is not necessary. Instead, we update the topmost entry's security level to consider the new scope's security level and maintain the structure of σ. This is explicit in rules [[BranchT-NoPush]] and [[BranchF-NoPush]]. At runtime, we add `pop` in all rules for conditional branches. The runtime instruction `pop` is never present in the original source code. The reduction of unconditional jumps ([[D-Jump]]) leaves σ unchanged but adds a `pop` to the next block; `pop` will always be the first instruction to execute after a jump. Rules [[D-Pop]] and [[D-NoPop]] define the behaviour of `pop` as it may or may not change σ. We must check if `pop`'s label matches the label of σ's topmost entry. If it does, the scope ends and we pop σ ([[D-Pop]]). Otherwise, σ remains unchanged ([[D-NoPop]]).

The stack represents scope nesting in a program and identifies merging points where context security levels can decrease, i.e. scopes end. Figure 4 depicts four possible outcomes of the successive application of reduction rules pushing and popping entries in σ. Consider the conditional jump in block ℓ_1.

In Figure 4a, [[BranchT-NoPush]] and [[D-NoPop]] apply in sequence. The branch in ℓ_1 merges in the same place as the enclosing scope (ℓ_5), and the branches follow distinct paths ($\ell_3 \neq \ell_4$). The label at the top of σ (ℓ_5) stays the same, the security level accounts for the the condition's security level ($\gamma(r_1)$).

In Figure 4b, rules [[BranchT-NoPush]] and [[D-Pop]] apply in sequence. The branching ℓ_1 merges in the same place as the enclosing scope (ℓ_5), and both branches lead directly to it. Therefore, the scope closes, and the stack pops.

In Figure 4c, [[BranchT-Push]] and [[D-NoPop]] apply in sequence. The branching in ℓ_1 merges in a location (ℓ_5) other than the enclosing scope's end (ℓ_6), and the branches follow distinct paths ($\ell_3 \neq \ell_4$). We push a new pair to σ, with the scope's ending location and the appropriate security level.

Finally, in Figure 4d, [[BranchT-Push]] and [[D-Pop]] apply in sequence. The branching in ℓ_1 merges immediately after the jump in a location (ℓ_5) other than the end of the enclosing scope (ℓ_6); the scope is empty, σ remains unchanged. The degenerate cases of Figure 4b and Figure 4d can result from a translation of structured programs and from compilers' optimizations.

4.1 Monitor Non-interference

Our monitor enforces non-interference for programs containing only guarded instructions ([[D-UnsafeAssignment]]). In this section, we sketch the non-interference proof, and later, we show that it also holds for well-formed rewritten programs with unguarded instructions. The proof consists in showing that two configurations of the same program are indistinguishable at an observation level g, if their initial states (σ and γ) are indistinguishable at g.

It is first convenient to define store (γ) and stack (σ) indistinguishability.

Definition 1 (Store Projection). *We define the projection of σ with relation to a security level g, $\gamma\lfloor_g$, as follows:*

$$\gamma\lfloor_g \triangleq \{r_1 : v_1^{k_1}, ..., r_n : v_n^{k_n}\},\ \forall r_i \in dom(\gamma).\ \gamma(r_i) = v_i^{k_i}\ \wedge\ k_i \sqsubseteq g$$

Definition 2 (Store Indistinguishability). *We define the indistiguishability relation on stores γ, γ', written $\gamma \sim_g \gamma'$, as follows:*

$$\gamma \sim_g \gamma' \triangleq\ \gamma\lfloor_g =\ \gamma'\lfloor_g$$

Definition 3 (Stack Projection). *We define the projection of a stack σ at observation level g, written $\sigma\lfloor_g$, as follows:*

$$\begin{array}{rl} nil\lfloor_g \triangleq & nil \\ ((\ell, a) :: \sigma)\lfloor_g \triangleq & (\ell, a) :: \sigma\lfloor_g \ \textit{If}\ a \sqsubseteq g \\ ((\ell, a) :: \sigma)\lfloor_g \triangleq & \sigma\lfloor_g \qquad\quad \textit{If}\ a \not\sqsubseteq g \end{array}$$

Definition 4 (Stack Indistinguishability). *We define the indistinguishability relation between two stacks, σ and σ', as follows:*

$$\sigma \sim_g \sigma' \ \triangleq\ \sigma\lfloor_g =\ \sigma'\lfloor_g$$

As an aid for the non-interference proof, resorting to Ω and Ω' as two configurations, we make a distinction between distinguish between computations observable at distinct security levels.

Definition 5 (High Transition). *A transition is high, written $\Omega \rightharpoonup_g \Omega'$, if the context security level (pc) is not lower than the observation level (g):*

$$\frac{(\Delta, \gamma, (\ell, pc) :: \sigma, \ell_c : I) \longrightarrow (\Delta, \gamma', \sigma', \ell'_c : I') \qquad pc \not\sqsubseteq g}{(\Delta, \gamma, (\ell, pc) :: \sigma, \ell_c : I) \rightharpoonup_g (\Delta, \gamma', \sigma', \ell'_c : I')}$$

Definition 6 (Low Transition). *A transiistion low, written $\Omega \rightharpoondown_g \Omega'$, if the context security level (pc) is lower than the observation level (g):*

$$\frac{(\Delta, \gamma, (\ell, pc) :: \sigma, \ell_c : I) \longrightarrow (\Delta, \gamma', \sigma', \ell'_c : I') \qquad pc \sqsubseteq g}{(\Delta, \gamma, (\ell, pc) :: \sigma, \ell_c : I) \rightharpoondown_g (\Delta, \gamma', \sigma', \ell'_c : I')}$$

Definition 7 (Mixed Transitions). *We define a mixed sequence of transitions, written* $\Omega \xrightarrow{(L,H)}_g \Omega'$*, as a combination of low (L) and high (H) transitions:*

$$\frac{\begin{array}{c}(\Delta,\gamma,(\ell,pc)::\sigma,\ell:I) \rightharpoonup^{l}_g (\Delta,\gamma_l,\sigma_l,\ell_l:I_l) \\ (\Delta,\gamma_l,\sigma_l,\ell_l:I_l) \rightharpoonup^{h}_g (\Delta,\gamma_h,\sigma_h,\ell_h:I_h) \\ (\Delta,\gamma_h,\sigma_h,\ell_h:I_h) \xrightarrow{(L-l,H-h)}_g (\Delta,\gamma',\sigma',\ell':I')\end{array}}{(\Delta,\gamma,(\ell,pc)::\sigma,\ell:I) \xrightarrow{(L,H)}_g (\Delta,\gamma',\sigma',\ell':I')}$$

If we reach a configuration Ω through a mixed sequence of L low transitions and H high transitions, then we there is Ω' such that we reach Ω' in l low transitions and h high transitions, and from Ω' we reach Ω through $L-l$ low transitions and $H-h$ high transitions. Take note that this also applies to programs starting in high transitions ($l=0$).

The non-interference proof follows from verifying that both high and low transition sequences, when executed separately, preserve σ and γ indistinguishability.

We first prove the confinement of high transitions, expressed in the lemma below. Instructions executed at a security level not lower than g do not visibly change γ and σ. The proof follows from case analysis of the reduction relation.

Lemma 1 (Confinement).
If $(\Delta,\gamma,\sigma,\ell_c:i;I) \rightharpoonup_g (\Delta,\gamma',\sigma',\ell'_c:I')$ *then* $\gamma \sim_g \gamma' \wedge \sigma \sim_g \sigma'$.

Store indistinguishability ($\gamma \sim_g \gamma'$) comes from two key aspects of rule ⟦D-UnsafeAssignment⟧: (i) guarded assignments do not write to registers whose security level is lower than the context security level; public information remains public; (ii) when writing to a register, the monitor computes the resulting security level using the computation's security label; secret information remains secret. Finally, control instructions do not modify γ, making them irrelevant in this proof.

We prove stack indistinguishability focusing on the rules for control instructions. ⟦D-Pop⟧ removes the topmost entry from σ. In a high transition, where the topmost entry is secret, its removal does not change the visible part of stack σ. Rule ⟦BranchT-Push⟧ (and ⟦BranchF-Push⟧) adds a new (ℓ,k) pair to σ with a higher security level ($PC(\sigma) \sqsubseteq k$). Thus, it does not introduce visible changes in the stack. Rule ⟦BranchT-NoPush⟧ updates the topmost entry of σ to a higher security level, and no changes to the stack are visible.

We now prove one-step non-interference for low transitions.

Lemma 2 (Low One-Step Non-interference).
If $(\Delta,\gamma,\sigma,\ell_c:i;I) \rightharpoonup_g (\Delta,\gamma_f,\sigma_f,\ell_{cf}:I_f)$ *and*
$(\Delta,\gamma',\sigma',\ell_c:i;I) \rightharpoonup_g (\Delta,\gamma'_f,\sigma'_f,\ell_{cf}{}':I'_f)$ *with* $\gamma \sim_g \gamma' \wedge \sigma \sim_g \sigma'$
then $\gamma_f \sim_g \gamma'_f$, $\sigma_f \sim_g \sigma'_f$, $\sigma_f = (\ell,pc)::\sigma$, *and* $pc \sqsubseteq g \implies I_f = I'_f$.

This proof follows by inspecting the reduction rules; the same instruction on both configurations will produce identical effects on γ and σ and evolve equally.

The only exception where $I_f \neq I'_f$ holds is when entering a sequence of high transitions. For $I_f \neq I'_f$ to hold the branch condition must evaluate to different values, which is only possible if the register in the condition is secret.

Considering programs as sequences of low and high transitions, we can prove non-interference by induction on the number of transitions.

Theorem 1 (Non-interference).

If $(\Delta, \gamma, \sigma, \ell_c : I) \xrightarrow{(x,y)}_g (\Delta, \gamma_f, \sigma_f, \ell_{cf} : I_f)$ *and*
$(\Delta, \gamma', \sigma', \ell_c : I) \xrightarrow{(x,z)}_g (\Delta, \gamma'_f, \sigma'_f, \ell'_{cf} : I'_f)$ *with* $\gamma \sim_g \gamma'$ *and* $\sigma \sim_g \sigma'$
then $\gamma_f \sim_g \gamma'_f$ *and* $\sigma_f \sim_g \sigma'_f$.

Two indistinguishable executions at the same point, by the transitive closure of Lemma 2, will remain indistinguishable until they start high transitions:

$$(\Delta, \gamma_a, \sigma_a, \ell_{ca} : I_a) \xrightarrow{i}_g (\Delta, \gamma_b, \sigma_b, \ell_{cb} : I_b)$$
$$(\Delta, \gamma_x, \sigma_x, \ell_{cx} : I_x) \xrightarrow{i}_g (\Delta, \gamma_y, \sigma_y, \ell_{cy} : I_y)$$

where $\gamma_a \sim_g \gamma_x$, $\sigma_a \sim_g \sigma_x$, $\ell_{ca} = \ell_{cx}$, $I_a = I_x$, $\gamma_b \sim_g \gamma_y$, $\sigma_b \sim_g \sigma_y$, $\ell_{cb} \neq \ell_{cy}$, $I_b \neq I_y$.

For high transitions, we apply the transitive closure of Lemma 1, concluding

$$(\Delta, \gamma_b, \sigma_b, \ell_{cb} : I_b) \xrightarrow{j}_g (\Delta, \gamma_c, \sigma_c, \ell_{cc} : I_c)$$
$$(\Delta, \gamma_y, \sigma_y, \ell_{cy} : I_y) \xrightarrow{k}_g (\Delta, \gamma_z, \sigma_z, \ell_{cz} : I_z)$$

This ensures that $\gamma_b \sim_g \gamma_c$, $\sigma_b \sim_g \sigma_c$, $\gamma_y \sim_g \gamma_z$, and $\sigma_y \sim_g \sigma_z$. By transitivity we have that $\gamma_c \sim_g \gamma_z$, $\sigma_c \sim_g \sigma_z$. For the hypothesis to apply, $I_c = I_z$ needs to hold. This condition is given by the properties of the control flow graph; if two executions diverge at the same point of the CFG then they will converge at the same point. With all conditions met, the proof follows by induction.

5 Static Semantics

We now present the procedure for rewriting unguarded Snitch IR programs to their hybrid counterpart. Our relation approximates IFC, rejecting only programs proven incorrect, and producing an equivalent program with less (or equal number of) guarded instructions. We assume that source programs do not contain guarded assignments nor `pop` instructions. Only the type system introduces guarded assignments, and only the monitor adds `pop` instructions at runtime.

We define a rewriting system recursively in the structure of each basic block, iterating each via the rules in Figure 5 and Figure 6. The rewriting relation

$$\Delta, \widehat{\Gamma}, \widehat{\Sigma} \vdash \overline{\ell \mapsto I} \Downarrow \overline{\ell \mapsto I'}$$

takes three environments: the code repository Δ, storing each basic block's instructions; the environment $\widehat{\Gamma}$, tracking the security levels of all registers on entry for each basic block; and the map of stacks $\widehat{\Sigma}$, capturing the control flow structure and corresponding nesting of security levels in all blocks. Rule [[Forall]]

⟦Forall⟧
$$\frac{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\Gamma}(\ell_i) \vdash \ell_i : I_i \Downarrow I_i' \quad \forall (\ell_i, I_i) \in \overline{\ell \mapsto I}}{\Delta, \widehat{\Gamma}, \widehat{\Sigma} \vdash \overline{\ell \mapsto I} \Downarrow \overline{\ell \mapsto I'}}$$

⟦S-SafeAssignment⟧
$$\frac{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma}[r : PC(\widehat{\Sigma}(\ell_c)) \sqcup k] \vdash \ell_c : I \Downarrow I' \quad PC(\widehat{\Sigma}(\ell_c)) \preccurlyeq \widehat{\gamma}(r)}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma} \vdash \ell_c : r := v^k; I \Downarrow r := v^k; I'}$$

⟦S-UnsafeAssignment⟧
$$\frac{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma}[r : PC(\widehat{\Sigma}(\ell_c)) \sqcup k] \vdash \ell_c : I \Downarrow I' \quad PC(\widehat{\Sigma}(\ell_c)) \preccurlyeq \widehat{\gamma}(r) = \downarrow}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma} \vdash \ell_c : r := v^k; I \Downarrow r :\hat{=} v^k; I'}$$

Fig. 5: Static Semantics for assignments

defines the rewriting of all blocks in a program ($\overline{\ell \mapsto I}$) to produce the final program ($\overline{\ell \mapsto I'}$). For each block, the rewriting procedure relies on the relation

$$\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma} \vdash \ell_c : I \Downarrow I'$$

where the fourth environment, $\widehat{\gamma}$, initialized out of environment $\widehat{\Gamma}$, maps registers of the present basic-block to security levels (intervals). This judgment relates a valid unguarded set of instructions I, part of basic block ℓ_c, to a set of valid guarded and unguarded instructions I'. All unguarded instructions are guaranteed to preserve data confidentiality (non-interference). We present the soundness results of the checking/rewriting system in Section 5.1.

We define the semantics in a syntax-directed way, with non-terminating instructions depicted in Figure 5, and block terminating instructions depicted in Figure 6. Rule ⟦S-SafeAssignment⟧ shows that the instruction is not modified since the safety conditions statically hold, i.e. the current security level $PC(\widehat{\Sigma}(\ell_c))$ is lower than the register's security level $\widehat{\gamma}(r)$. The changed register's security level now accounts for the level of the context, the level of the assigned value, in the rewriting of the subsequent instructions ($\widehat{\gamma}[r : \widehat{\Sigma}(\ell_c) \sqcup k]$).

If we know statically that the context's security level is lower than the security level of the register, rule ⟦S-SafeAssignment⟧, the assignment is secure and is not trapped at runtime. If the comparison is undefined, i.e. security intervals intersect ($PC(\widehat{\Sigma}(\ell_c)) \preccurlyeq \widehat{\gamma}(r) = \downarrow$), rule ⟦S-UnsafeAssignment⟧, we rewrite it as a guarded assignment, so that the monitor prevents any leaks. Finally, if the context's level is higher or unrelated, no rule applies, the program gets rejected.

The rules included in Figure 6 define the static semantics for jump and conditional jump instructions, comparing and validating the nesting structure of stacks and the compatibility between registers in the departing and landing blocks. Notice that these rules only check the structure of the control flow graph, and they do not introduce or rewrite the existing code as the previous set of rules.

The security level stack assigned to each block ($\widehat{\Sigma}(\ell_c)$) stores the nesting hierarchy ruling the block, containing information about where each scope ends.

⟦S-Branch-NoPush-NoPop⟧

$$\frac{\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell_i) \quad \widehat{\Sigma}(\ell_c) = (\ell, \widehat{pc}) :: \widehat{\sigma} \quad (\ell, \widehat{pc} \sqcup \widehat{\gamma}(r)) :: \widehat{\sigma} \preccurlyeq \widehat{\Sigma}(\ell_i) \quad \mathit{postDom}(\ell_c) = \ell \quad \ell_i \neq \ell \quad \ell_i \in \{\ell_t, \ell_f\}}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma}[r : \widehat{k}] \vdash \ell_c : \texttt{if}\ r\ , \ell_t\ , \ell_f \Downarrow \texttt{if}\ r\ , \ell_t\ , \ell_f}$$

⟦S-Branch-NoPush-Pop⟧

$$\frac{\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell_i) \quad \widehat{\Sigma}(\ell_c) = (\ell, \widehat{pc}) :: \widehat{\sigma} \quad \widehat{\sigma} \preccurlyeq \widehat{\Sigma}(\ell_i) \quad \mathit{postDom}(\ell_c) = \ell = \ell_i \quad \ell_i \in \{\ell_t, \ell_f\}}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma}[r : \widehat{k}] \vdash \ell_c : \texttt{if}\ r\ , \ell_t\ , \ell_f \Downarrow \texttt{if}\ r\ , \ell_t\ , \ell_f}$$

⟦S-Branch-Push-NoPop⟧

$$\frac{\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell) \quad \widehat{\Sigma}(\ell_c) = (\ell, \widehat{pc}) :: \widehat{\sigma} \quad (\ell', \widehat{pc} \sqcup \widehat{\gamma}(r)) :: \widehat{\Sigma}(\ell_c) \preccurlyeq \widehat{\Sigma}(\ell_i) \quad \mathit{postDom}(\ell_c) = \ell' \quad \ell' \neq \ell \quad \ell' \neq \ell_i \quad \ell_i \in \{\ell_t, \ell_f\}}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma}[r : \widehat{k}] \vdash \ell_c : \texttt{if}\ r\ , \ell_t\ , \ell_f \Downarrow \texttt{if}\ r\ , \ell_t\ , \ell_f}$$

⟦S-Branch-Push-Pop⟧

$$\frac{\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell) \quad \widehat{\Sigma}(\ell_c) = (\ell, \widehat{pc}) :: \widehat{\sigma} \quad \widehat{\Sigma}(\ell_c) \preccurlyeq \widehat{\Sigma}(\ell) \quad \mathit{postDom}(\ell_c) = \ell_i \quad \ell_i \neq \ell \quad \ell_i \in \{\ell_t, \ell_f\}}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma}[r : \widehat{k}] \vdash \ell_c : \texttt{if}\ r\ , \ell_t\ , \ell_f \Downarrow \texttt{if}\ r\ , \ell_t\ , \ell_f}$$

⟦S-Jump-NoPop⟧

$$\frac{\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell) \quad \widehat{\Sigma}(\ell_c) = (\ell', \widehat{pc}) :: \widehat{\sigma} \quad \ell \neq \ell' \quad \widehat{\Sigma}(\ell_c) \preccurlyeq \widehat{\Sigma}(\ell)}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma} \vdash \ell_c : \texttt{jump}\ \ell \Downarrow \texttt{jump}\ \ell}$$

⟦S-Jump-Pop⟧

$$\frac{\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell) \quad \widehat{\Sigma}(\ell_c) = (\ell, \widehat{k}) :: \widehat{\sigma} \quad \widehat{\sigma} \preccurlyeq \widehat{\Sigma}(\ell)}{\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma} \vdash \ell_c : \texttt{jump}\ \ell \Downarrow \texttt{jump}\ \ell}$$

Fig. 6: Static Semantics for control instructions

The stacks associated with basic blocks work in a similar fashion to the stack used in the monitor. They store pairs with a security level and the corresponding target label in an increasing sequence, depicting how deep the block is in the control flow graph. The security level stored at the top of the stack is the current scope's security level, and the label specifies the scope's closing point in the CFG.

Conditional jump instructions capture branching in the control flow graph and there are four rules to consider, each matching one of the cases depicted in Figure 4. In Figure 4(a), rule ⟦S-Branch-NoPush-NoPop⟧ captures the case where a new scope ends at the same node as its parent, and the next node is not the post-dominant node. In this case, the rule checks that the security level in both target blocks matches the current stack updated with the combination of the current level with the condition's security level ($(\ell, \widehat{pc} \sqcup \widehat{\gamma}(r)) :: \widehat{\sigma} \preccurlyeq \widehat{\Sigma}(\ell_i)$). We also check that the registers in the next blocks match ($\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell)$). In Figure 4(b),

rule ⟦S-Branch-NoPush-Pop⟧ captures the degenerated case where we start a new scope terminating in the same node as the parent, and we reach said node in one step; both branches jump to the same block. In this case, we need to check that the registers in the following blocks match and that the stack is one element shorter to close the current (and parent) scope(s) ($\widehat{\sigma} \preccurlyeq \widehat{\Sigma}(\ell_i)$ where $\widehat{\sigma}$ is the stack after the pop). In Figure 4(c), rule ⟦S-Branch-Push-NoPop⟧ reflects the case where we start a scope whose post-dominant differs from the parent scope's, and we jump to a node other than the post-dominant node of the current block. Here, we check that the registers match and that the target blocks expect a stack with a new entry, $((\ell', \widehat{pc} \sqcup \widehat{\gamma}(r)) :: \widehat{\Sigma}(\ell_c) \preccurlyeq \widehat{\Sigma}(\ell_i))$. Rule ⟦S-Branch-Push-Pop⟧ captures the degenerated case where we start and terminate a new scope whose post-dominant is different from the parent's scope. We check the compatibility between the source and target registers ($\widehat{\gamma} \preccurlyeq \widehat{\Gamma}(\ell)$) and stack ($\widehat{\Sigma}(\ell_c) \preccurlyeq \widehat{\Sigma}(\ell)$).

Unconditional jump instructions, together with the degenerated cases from conditional jumps above (⟦S-Branch-NoPush-Pop⟧ and ⟦S-Branch-Push-Pop⟧), represent path convergence in the CFG. We cover two outcomes of the jump instruction. Rule ⟦S-Jump-NoPop⟧ checks if the registers match and if the stack is compatible with the target block's stack. Rule ⟦S-Jump-Pop⟧ checks if the registers match and if the stack, except for the topmost entry, is compatible with the target block's stack. We pop the topmost entry from $\widehat{\sigma}$ since we are jumping to the current scope's (and, potentially, parent scopes) end.

In summary, the static semantics checks that all basic blocks match a given specification for $\widehat{\sigma}$ and $\widehat{\gamma}$ and all jumps abide by the same nesting discipline, matching the security levels specified in the $\widehat{\sigma}$. Algorithmically, the checking and rewriting procedure takes as input Δ and the program ($\overline{\ell \mapsto I}$). It is necessary to synthesize the environments $\widehat{\Gamma}$ and $\widehat{\Sigma}$, and the output program $\overline{\ell \mapsto I'}$ from constraints collected when constructing the proof of rewriting, in the style of Hindley Milner type systems [28,10]. We next prove the soundness of the static checking procedure with relation to the dynamic semantics presented in Section 4, showing that no untrapped errors occur in the unguarded part of the program and that the reference monitor captures all remaining errors.

5.1 Soundness

The soundness result of our approach guarantees that only guarded instructions raise errors at runtime and are, therefore, rightfully trapped. Thus, we prove that all omitted verifications are unnecessary, as unguarded instructions will not cause any illegal flows.

The semantics in Section 4 only verifies guarded instructions, as unguarded instructions just require label propagation. Consider the extended semantics including the rules present in Figure 5, Figure 6, and Figure 7. The latter conatins new rules for unguarded assignments and error trapping for all instructions. We introduce two kinds of error, one that the monitor traps (↯) and other that the monitor does not trap (☠). We want to prove that for well typed programs the latter never occurs. To relate the static and the dynamic semantics, we define an interpretation for static stores and stacks:

⟦D-SafeAssignment-Bad⟧

$$\frac{\gamma(r) = u^{k'} \quad PC(\sigma) \not\sqsubseteq k'}{(\Delta, \gamma, \sigma, \ell_c : r := v^k; I) \longrightarrow \skull}$$

⟦D-UnsafeAssignment-Trapped⟧

$$\frac{\gamma(r) = u^{k'} \quad PC(\sigma) \not\sqsubseteq k'}{(\Delta, \gamma, \sigma, \ell_c : r :\doteq v^k; I) \longrightarrow \lightning}$$

Fig. 7: Error aware dynamic rules

Definition 8 (Static Store Interpretation). *We define the interpretation of* $\widehat{\gamma}$*, written,* $[\![\widehat{\gamma}]\!]$*, as follows:*

$$[\![\widehat{\gamma}]\!] \triangleq \{\gamma \mid \forall r \in dom(\gamma).\ r \in dom(\widehat{\gamma}) \wedge \gamma(r) = v^k \wedge k \in \widehat{\gamma}(r)\}$$

Definition 9 (Static Stack Interpretation). *We define the interpretation of* $\widehat{\sigma}$*, written,* $[\![\widehat{\sigma}]\!]$*, as follows:*

$$\begin{aligned}[\![\widehat{\sigma}]\!] \triangleq \{\sigma \mid\ & (\widehat{\sigma} = nil \wedge \sigma = nil) \vee \\ & (\sigma = (\ell, k) :: \sigma' \wedge \widehat{\sigma} = (\ell, \widehat{k}) :: \widehat{\sigma}' \wedge k \in \widehat{k} \wedge \sigma' \in [\![\widehat{\sigma}']\!])\}\end{aligned}$$

The soundness lemma for our tyoe system with relation to the operational semantics is as follows.

Theorem 2 (Static Soundness).
If $\Delta, \widehat{\Gamma}, \widehat{\Sigma}, \widehat{\gamma} \vdash \ell_c : I \Downarrow I'$ *and* $(\Delta, \gamma, \sigma, \ell_c : I') \xrightarrow{n} \Omega'$ *with* $\gamma \in [\![\widehat{\gamma}]\!]$ *and* $\sigma \in [\![\widehat{\Sigma}(\ell_c)]\!]$ *then* $\Omega' \neq \skull$.

The structure of the proof resembles that of a proof of progress and follows by induction on the number of reduction steps. It follows that all well-formed rewriting judgments match the premises of the initial semantics (including the error $\lightning$), and untrapped errors $\skull$ never occur. For the proof, we consider as base case, the scenario where we reach the final configuration trough zero transitions, i.e., we start in the final configuration. For the induction step, we prove that the monitor only transits to states other than $\skull$, otherwise we have that all other cases do not apply by contradiction.

Considering an assignment, according to rule ⟦D-SafeAssignment⟧, the system will not reduce to $\skull$. Moreover, due to ⟦S-SafeAssignment⟧, Definition 8 , and Definition 9 , the conditions for the induction hypothesis hold. When analysing a transition through ⟦D-SafeAssignment-Bad⟧, we reach a contradiction. According to this rule, we have that $PC(\sigma) \not\sqsubseteq k$ but by ⟦S-SafeAssignment⟧, Definition 8 , and Definition 9 , we have $PC(\sigma) \sqsubseteq k$. Rule ⟦D-UnsafeAssignment⟧ follows rule ⟦D-SafeAssignment⟧. When considering rule ⟦D-UnsafeAssignment-Trapped⟧, the system evolves to $\lightning$ which, by definition, is different from $\skull$.

Binary operations (not covered in the semantics) are similar regular assignments, and the proof for binary operations follows the proofs for assignments. The remaining instructions never transit to ☠, hence, it is only necessary to prove that they preserve the condition required for the induction hypothesis.

6 Conclusions

We presented Snitch IR, a low-level language with a hybrid IFC mechanism with dependent security levels. The core of our approach is a rewriting procedure that checks the validity of unstructured programs and injecting guard into programs. Guarded instructions will check dynamically situations that do not fail in the permissive check of the type system. We prove that our monitor preserves termination insensitive non-interference in Snitch IR, and we prove that the hybrid monitor is sound in that it satisfies the non-interference property while minimizing the runtime checks needed.

We identify as future research directions the formal support for an interprocedural analysis, and the extension of the language with function calls. Also, we envision the further integration of function parameters in the security lattice in the formal presentation.

Our current implementation supports fully dynamic analysis of JVM bytecode, and we aim to support the hybrid IFC analysis, a work in progress. Furthermore, we wish to extend the prototype to support other mainstream low-level languages. The particular aspects of dynamically allocated memory (records, arrays, and objects) are also interesting as future work.

Acknowledgements

This work is supported by FCT/MCTES SFRH/BD/149043/2019, FCT/MCTES Grant NOVA LINCS - UIDB/04516/2020 and GOLEM Lisboa-01-0247-Feder-045917, INESC-ID multi-annual funding (UIDB/50021/2020) and INFOCOS (PTDC/CCI-COM/32378/2017).

References

1. Agrawal, P.: Keeping your account secure, `https://blog.twitter.com/official/en_us/topics/company/2018/keeping-your-account-secure.html`, accessed on 15.10.2021
2. Aldous, P., Might, M.: Static analysis of non-interference in expressive low-level languages. In: Static Analysis. pp. 1–17 (2015)
3. Aldous, P., Might, M.: A posteriori taint-tracking for demonstrating non-interference in expressive low-level languages. In: IEEE Security and Privacy Workshops. pp. 179–184 (2016)
4. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (2009)

5. Balliu, M., Dam, M., Guanciale, R.: Automating information flow analysis of low level code. In: ACM SIGSAC Conference on Computer and Communications Security. pp. 1080–1091 (2014)
6. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference java bytecode verifier. In: Programming Languages and Systems, 16th European Symposium on Programming. pp. 125–140 (2007)
7. Barthe, G., Rezk, T.: Non-interference for a jvm-like language. p. 103–112. TLDI '05 (2005)
8. Barthe, G., Rezk, T., Naumann, D.: Deriving an information flow checker and certifying compiler for java. In: IEEE Symposium on Security and Privacy. p. 230–242 (2006)
9. Cimpanu, C.: Github accidentally recorded some plaintext passwords in its internal logs (May 2018), `https://www.bleepingcomputer.com/news/security/github-accidentally-recorded-some-plaintext-passwords-in-its-internal-logs/`, accessed on 15.10.2021
10. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 207–212 (1982)
11. Denning, D.E.: A lattice model of secure information flow. Commun. ACM p. 236–243 (May 1976)
12. Disney, T., Flanagan, C.: Gradual information flow typing. In: STOP'11 (2011)
13. ECMA International: Standard ECMA-335 - Common Language Infrastructure (CLI) (Dec 2010)
14. Erlingsson, U., Schneider, F.B.: Sasi enforcement of security policies: A retrospective. In: Workshop on New Security Paradigms. p. 87–95 (1999)
15. Fennell, L., Thiemann, P.: Gradual security typing with references. In: IEEE Computer Security Foundations Symposium. pp. 224–239 (2013)
16. Fennell, L., Thiemann, P.: LJGS: Gradual Security Types for Object-Oriented Languages. In: European Conference on Object-Oriented Programming. pp. 9:1–9:26 (2016)
17. Ferreira, P.J.A.D.: Msc dissertation. information flow analysis using data-dependent logical propositions. (2012), faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
18. Fragoso Santos, J.I.F.T., Jensen, T., Rezk, T., Schmitt, A.: Hybrid typing of secure information flow in a javascript-like language. In: Trustworthy Global Computing. pp. 63–78 (2016)
19. Ghosal, S., Shyamasundar, R.K.: Pifthon: A compile-time information flow analyzer for an imperative language. CoRR (2021)
20. Hedin, D., Bello, L., Sabelfeld, A.: Value-sensitive hybrid information flow control for a javascript-like language. In: IEEE Computer Security Foundations Symposium. p. 351–365 (2015)
21. Hedin, D., Sabelfeld, A.: Information-flow security for a core of javascript. In: 2012 IEEE 25th Computer Security Foundations Symposium. pp. 3–18 (2012)
22. Hedin, D., Sabelfeld, A.: A perspective on information-flow control. In: Software Safety and Security - Tools for Analysis and Verification, pp. 319–347 (2012)
23. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International Symposium on Code Generation and Optimization (Mar 2004)
24. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. pp. 121–141 (1979)

25. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification, Java SE 8 Edition (2014)
26. Lourenço, L., Caires, L.: Dependent information flow types. SIGPLAN Not. p. 317–328 (Jan 2015)
27. Lourenço, L., Caires, L.: Information flow analysis for valued-indexed data security compartments. In: Trustworthy Global Computing. pp. 180–198 (2014)
28. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences pp. 348–375 (1978)
29. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system f to typed assembly language. ACM Trans. Program. Lang. Syst. p. 527–568 (May 1999)
30. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. p. 410–442 (Oct 2000)
31. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif 3.0: Java information flow (2006), accessed on 15.10.2021
32. O'Flaherty, K.: Facebook exposed up to 600 million passwords – here's what to do (Mar 2019), https://www.forbes.com/sites/kateoflahertyuk/2019/03/21/facebook-has-exposed-up-to-600-million-passwords-heres-what-to-do/#6f301fe4bc90, accessed on 15.10.2021
33. Polikarpova, N., Stefan, D., Yang, J., Itzhaky, S., Hance, T., Solar-Lezama, A.: Liquid information flow control. Proc. ACM Program. Lang. (Aug 2020)
34. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
35. Schneider, F., Morrisett, G., Harper, R.: A Language-Based Approach to Security (2001)
36. Siek, J., Taha, W.: Gradual typing for objects. In: European Conference on Object-Oriented Programming (2007)
37. Simonet, V., Rocquencourt, I.: Flow caml in a nutshell. First APPSEM-II Workshop (Apr 2003)
38. Toro, M., Garcia, R., Tanter, E.: Type-driven gradual security with references. ACM Trans. Program. Lang. Syst. pp. 1–55 (Dec 2018)
39. Winder, D.: Facebook quietly confirms millions of unencrypted instagram passwords exposed – change yours now (Apr 2019), https://www.forbes.com/sites/daveywinder/2019/04/19/facebook-quietly-confirms-millions-of-unencrypted-instagram-passwords-exposed-change-yours-now/#22e5d5844537, accessed on 15.10.2021