Program Transformation for Non-interference Verification on Programs with Pointers

Mounir Assaf[1], Julien Signoles[1], Frédéric Tronel[2], and Éric Totel[2]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] Supelec, CIDre, Rennes France
firstname.lastname@supelec.fr

Abstract. Novel approaches for dynamic information flow monitoring are promising since they enable permissive (accepting a large subset of executions) yet sound (rejecting all unsecure executions) enforcement of non-interference. In this paper, we present a dynamic information flow monitor for a language supporting pointers. Our flow-sensitive monitor relies on prior static analysis in order to soundly enforce non-interference. We also propose a program transformation that preserves the behavior of initial programs and soundly inlines our security monitor. This program transformation enables both dynamic and static verification of non-interference.

1 Introduction

Information security is usually enforced through access control security policies. Those security policies, implemented at the OS level, can authorize or deny information flows at a coarse-grained subject/object level. Information flow control (IFC) mechanisms, occuring at the application level, offer more granularity to enforce precise flow policies.

The seminal work in IFC has been initiated by Denning and Denning [1]. They proposed a static analysis to verify that information is propagated inside programs securely with respect to a flow policy. For instance, a simple flow policy disallows leakage of secret variables into public ones, hence ensuring confidentiality. This notion has been generalized by Goguen and Meseguer [2] as non-interference. Non-interference, precisely its termination-insensitive formulation (TINI), has been widely adopted in IFC as a security policy [3,4,5]. Informally, it states that, when changing only secret inputs, terminating executions of a program must deliver the same public outputs.

Volpano et al. [3] formalize a Denning-style static analysis as a type system for a simple imperative language. Volpano's work provides the first soundness proof stating that a typable program is secure with respect to TINI. However, Volpano's type system lacks flow-sensitivity since security labels associated to variables are not allowed to change during analysis. For example, the program $public = secret; public = 0$ is secure because the final content of variable *public* is overridden. Still, this program is not typable by Volpano's type system because of flow-insensitivity.

Hunt and Sands [6] extend Volpano's type system with flow-sensitivity, hence permitting security labels to change in order to reflect the precise security level of their

L.J. Janczewski, H.B. Wolfe, and S. Shenoi (Eds.): SEC 2013, IFIP AICT 405, pp. 231–244, 2013.

contents. Introducing flow-sensitivity to security type systems contributes to more permissive security analyses. Hunt and Sands prove the soundness of their type system with respect to TINI, while typing a larger subset of secure programs in comparison with Volpano's type system.

Dynamic monitoring of information flows is also known to provide more permissiveness [7,5] (accepting a large subset of executions). Unlike static analyses which enforce TINI for all possible execution paths, dynamic monitoring ensures that a single execution path is secure. However, permissiveness through the combination of both dynamic monitoring and flow-sensitivity requires careful examination. Indeed, Russo and Sabelfeld [5] prove that flow-sensitivity in purely dynamic IFC introduces covert channels leaking information. The main idea behind this result is that a purely dynamic monitor ignores non-executed conditional branches, missing at the same time information flows they produce. Therefore, a flow-sensitive dynamic monitor must rely on static analyses for sound (rejecting all unsecure executions) IFC.

Contributions. In this paper, we investigate **permissive yet sound flow-sensitive** IFC for programs handling pointers. Our contributions are:

- We formalize a hybrid information flow monitor for an imperative language with pointers and aliasing, by relying on a semantics built upon the Clight [8] semantics. This semantics is especially used in the CompCert [9] provably correct compiler. We prove the soundness of our monitor with respect to TINI.
- We also propose a **sound program transformation** which inlines our information flow monitor. For languages that are compiled directly into native machine code as it is the case for the C language, inlining is necessary to ensure fine-grained information flow monitoring. To our knowledge, our program transformation is the first proven **sound inlining approach for dynamic monitors handling pointers**.
- Assuming the implementation of security labels and their join operator, TINI can be enforced by running the self-monitoring transformed program. This **dynamic approach** has the advantage of being permissive since it soundly monitors a single execution path, ignoring possible unsecure paths that are not executed. The program transformation T also enables the verification of TINI by static analysis for free. Such a **static approach** computes an over-approximation of the transformed program semantics, enforcing TINI for all execution paths.

Outline. Section 2 introduces information flow background. Section 3 formalizes our information flow monitor for a simple imperative language handling pointers and aliasing. Section 4 defines a program transformation inlining our information flow monitor. We discuss related work in Section 5 and future work in Section 6.

2 Background

Non-interference. Our attacker model assumes that attackers know the source code of analyzed programs. It also supposes that attackers can only modify public inputs and read public outputs. A program is non-interferent if two terminating executions which differ only on secret inputs deliver the same public outputs. This notion of non-interference [2] formalizes independence of public outputs from secret inputs.

Information flows. Explicit and implicit flows [1] are generally taken into account when enforcing TINI. Explicit flows are produced from any source variable y assigned to a destination variable x. Implicit flows are produced whenever an affectation occurs in conditional branches. For instance, the following program $if\ (secret)\ x = 1\ else\ skip$ generates an implicit flow from *secret* to x, whatever the executed branch is. Even if x is not assigned, an attacker could learn that *secret* is false if x is different from 1. As one generally enforces a sound approximation of TINI, we suppose that assignments inside conditionals always produce implicit flows from the guards to assigned variables.

Additional information flows arise in the presence of pointers. Consider for example, the program $if\ (secret)\ \{x = \&a\}\ else\ \{x = \&b\}\ print\ *x$. An attacker, knowing the initial values of a and b, may learn information about the value of variable *secret* whenever $*x$ is output : there is an information flow from *secret* to $*x$. There are actually two different kinds of information flows involved in this case. The first one is an implicit flow from *secret* to x because of assignments inside a conditional depending on *secret*. The second one, due to pointer aliasing and dereferencing, is from x to $*x$. Thus, by transitivity, there is an information flow from *secret* to x.

Similarly, the program $if\ (secret)\ \{x = \&a\}\ else\ \{x = \&b\}\ *x = 1$ exposes pointer-induced flows from *secret* to variables a and b. An attacker having access to either variables a or b after the assignment $*x = 1$, may learn information about variable *secret*. It is worth noting that even if a (resp. b) is not assigned by instruction $*x = 1$, an information flow from *secret* to a (resp. b) is still produced. In fact, this pointer-induced information flow involves all variables that could have been written by $*x = 1$ (here, both variables a and b).

As we are aiming at enforcing TINI, we ignore in this paper all covert channels due to diverging runs and timing channels. Hence, a program like $while\ (secret)\ skip$; could leak information about variable *secret*. Yet, this is acceptable since even in the presence of outputs, Askarov et al. [4] have proved that an attacker could not know the secret in polynomial time in the size of the secret.

3 Information Flow Monitoring Semantics

Language overview. Figure 1 presents the abstract syntax of our language. It is a simple imperative language handling basic types (κ) like integers and pointers ($ptr(\tau)$). It handles aliasing but no pointer arithmetics: binary operators do not take pointers as arguments. The semantics of this language is inspired by the Clight semantics [8]. Clight is formalized in the context of the CompCert verified compiler for C programs [9].

A simplified version of the Clight big-step operational semantics considers an environment E and a memory M. $E : Var \rightharpoonup Loc$ maps variables to statically allocated locations. $M : Loc \rightharpoonup \mathbb{V}$ maps locations to values of type τ. The evaluation of an instruction c in an environment E and a memory M, denoted by $E \vdash c, M \Rightarrow M'$, results in a new memory M'. Expressions can be evaluated as either left-values or right-values depending on the position in which they occur. Only expressions having the form id or $*a$ can occur in l-value positions such as the left-hand side of assignments, whereas any expression can occur in right-value position.

As illustrated by Figure 2, l-value evaluation of expression a_1 in environment E and memory M ($E \vdash a_1, M \Leftarrow l$) provides the location l where a_1 is stored, whereas r-value

$$
\begin{array}{rl}
\textbf{Types:} & \tau ::= \kappa \mid ptr(\tau) \\
\textbf{Expressions:} & a ::= n \mid id \mid uop\ a \mid a_1\ bop\ a_2 \\
 & \quad \mid *a \mid \&a \\
\textbf{Instructions:} & c ::= skip \mid a_1 = a_2 \mid c_1;\ c_2 \\
 & \quad \mid if\ (a)\ c_1\ else\ c_2 \mid while\ (a)\ c \\
\textbf{Declarations:} & dcl ::= (\tau\ id)^* \\
\textbf{Programs:} & P ::= dcl;\ c
\end{array}
$$

Fig. 1. Abstract syntax of our language

$$
(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M \Leftarrow l \\ E \vdash a_2, M \Rightarrow v \\ M' = M[l \mapsto v] \end{array}}{E \vdash a_1 = a_2, M \Rightarrow M'}
$$

Fig. 2. Assignment semantics in Clight

evaluation of a_2 ($E \vdash a_2, M \Rightarrow v$) provides the value v of expression a_2. The assignment rule then maps the value v to the location l in the new memory M'.

In order to extend Clight's three judgment rules with the information flow monitor semantics, we consider a lattice $\mathbb{S} = (SC, \sqsubseteq)$ where $public \in SC$ is the minimal element of $\mathbb{S}$. We note $\sqcup$ the associated join operator. We also consider a new kind of memory $\Gamma : Loc \rightharpoonup \mathbb{S}$, which maps locations to security labels. Informally, security memory Γ tracks the security level of locations content through tainting. For example, an assignment $x = y + z$ generates an information flow from y and z to x. Thus, Γ maps to $E(x)$ (*i.e.* the location associated to x) the security label $\Gamma(E(y)) \sqcup \Gamma(E(z))$.

Expressions. Both Clight's r-value and l-value evaluations of expressions are extended to support the propagation of security labels, as illustrated in Figure 3: the evaluation of expressions yields both a value $v \in \mathbb{V}$ and a security label $s \in S$. If the pair (l, s_l) is the result of l-value evaluation of expression a, then the security label s_l captures pointer-induced flows produced by possible dereferences occurring in a, whereas $s_r = \Gamma(l)$ captures explicit flows produced by reading the value $M(l)$ of a. Therefore, the r-value evaluation of a produces a value $v = M(l)$ and a security label $s = s_l \sqcup s_r$ taking into account both explicit and pointer-induced flows through the join operator (rule *RV*). Note that the semantics of Clight expressions can be obtained from Figure 3 by ignoring all the monitor related operations.

The security label associated to the r-value of a defines the label associated to the l-value of $*a$ (rule LV_{MEM}), hence taking into account the pointer-induced information flow from a to $*a$. R-values of constants are labeled as *public* because attackers are supposed to know the source code. Since the locations of variables are at known offsets from the base pointer, we associate *public* to the l-values of variables (rule LV_{ID}). The label of the l-value of a defines the label associated to the r-value of $\&a$ (rule RV_{REF}). The security label associated to the r-value of a is propagated to the r-value of $uop\ a$ (rule RV_{UOP}). Likewise, the security label associated to the r-value of $a_1\ bop\ a_2$ takes into account both a_1 and a_2 r-values security labels through the join operator.

Figure 4 illustrates an example of the r-value evaluation of $*x$. Supposing that x is stored at location l_x and points to a variable a stored at location l_a, the r-value evaluation of $*x$ takes into account both pointer-induced and explicit flows since both s_x (the security label of x) and s_a (the security label of a) affect the resulting security label s.

$$LV_{ID}\ \frac{E(id) = l}{E \vdash id, M, \Gamma \Leftarrow l, public} \qquad LV_{MEM}\ \frac{E \vdash a, M, \Gamma \Rightarrow ptr(l), s}{E \vdash *a, M, \Gamma \Leftarrow l, s}$$

$$RV_{CONST}\ E \vdash n, M, \Gamma \Rightarrow n, public \qquad RV\ \frac{E \vdash a, M, \Gamma \Leftarrow l, s_l \quad M(l) = v \quad s_r = \Gamma(l) \quad s = s_l \sqcup s_r}{E \vdash a, M, \Gamma \Rightarrow v, s}$$

$$RV_{REF}\ \frac{E \vdash a, M, \Gamma \Leftarrow l, s}{E \vdash \&a, M, \Gamma \Rightarrow ptr(l), s} \qquad RV_{UOP}\ \frac{E \vdash a, M, \Gamma \Rightarrow v, s \quad uop\ v = v'}{E \vdash uop\ a, M, \Gamma \Rightarrow v', s}$$

$$RV_{BOP}\ \frac{E \vdash a_1, M, \Gamma \Rightarrow v_1, s_1 \quad E \vdash a_2, M, \Gamma \Rightarrow v_2, s_2 \quad v_1\ bop\ v_2 = v \quad s_1 \sqcup s_2 = s}{E \vdash a_1\ bop\ a_2, M, \Gamma \Rightarrow v, s}$$

Fig. 3. Information flow monitor big-step semantics of expressions

$$RV\ \frac{LV_{MEM}\ \frac{LV_{ID}\ \frac{E(x) = l_x}{E \vdash x, M, \Gamma \Leftarrow l_x, public} \quad RV\ \frac{M(l_x) = ptr(l_a) \quad \Gamma(l_x) = s_x \quad s_x = public \sqcup s_x}{E \vdash x, M, \Gamma \Rightarrow ptr(l_a), s_x}}{E \vdash *x, M, \Gamma \Leftarrow l_a, s_x} \quad M(l_a) = v \quad \Gamma(l_a) = s_a \quad s = s_a \sqcup s_x}{E \vdash *x, M, \Gamma \Rightarrow v, s}$$

Fig. 4. An example of expression $*x$ evaluation

One consequence of rules LV_{ID} and RV_{REF} is that addresses of variables are labeled as *public*. Thus, they can be accessed by attackers and used to bypass security measures such as ASLR (Address Space Layout Randomization). In fact, this kind of information leaks is out of scope for our analysis since addresses of variables are not secret inputs of programs. Furthermore, mapping any security label s other than *public* to the l-value of variables *id* would taint all data accessed through dereferences of *id*, causing a label creep problem [10].

Instructions. The semantics of instructions is presented in Figure 5. It is a combination of dynamic monitoring and static analysis through the use of $S_P(c)$, the set of locations that may have been written by instruction c of program P. The statically computed set $S_P(c)$ is fed to the semantics whenever a call to the *update* operator occurs. We also introduce a new meta-variable $\underline{pc}$ to capture implicit flows. $\underline{pc}$ can be viewed as the security label of the program counter. Each time a program enters a conditional, $\underline{pc}$ is updated with the guard security label in order to reflect generated implicit flows. Therefore, evaluation of instructions occurs in a memory Γ, an execution context $\underline{pc}$ in addition to a memory M and an environment E. It produces new memories Γ' and M'.

For assignment $a_1 = a_2$ (rule *Assign*), the join of three security labels are mapped to the location of a_1. First, s_1 takes into account pointer-induced flows from the l-

$$(Assign)\ \frac{\begin{array}{c}E \vdash a_1,M,\Gamma \Leftarrow l_1,s_1 \quad E \vdash a_2,M,\Gamma \Rightarrow v_2,s_2 \\ s = s_1 \sqcup s_2 \sqcup \underline{pc} \quad s' = s_1 \sqcup \underline{pc} \quad M' = M[l_1 \mapsto v_2] \\ \Gamma'' = \Gamma[l_1 \mapsto s] \quad \Gamma' = update(a_1 = a_2, s', \Gamma'')\end{array}}{E \vdash a_1 = a_2, M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'}$$

$$(Comp)\ \frac{E \vdash c_1,M,\Gamma,\underline{pc} \Rightarrow M_1,\Gamma_1 \quad E \vdash c_2,M_1,\Gamma_1,\underline{pc} \Rightarrow M_2,\Gamma_2}{E \vdash c_1;c_2,M,\Gamma,\underline{pc} \Rightarrow M_2,\Gamma_2}$$

$$(If_{tt})\ \frac{\begin{array}{c}E \vdash a,M,\Gamma \Rightarrow v,s \quad istrue(v) \\ \underline{pc'} = s \sqcup \underline{pc} \quad E \vdash c_1,M,\Gamma,\underline{pc'} \Rightarrow M_1,\Gamma_1 \\ \Gamma_1' = update(c_2,\underline{pc'},\Gamma_1)\end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2,M,\Gamma,\underline{pc} \Rightarrow M_1,\Gamma_1'}$$

$$(W_{ff})\ \frac{\begin{array}{c}E \vdash a,M,\Gamma \Rightarrow v,s \quad isfalse(v) \\ \underline{pc'} = s \sqcup \underline{pc} \quad \Gamma' = update(c,\underline{pc'},\Gamma)\end{array}}{E \vdash while\ (a)\ c,M,\Gamma,\underline{pc} \Rightarrow M,\Gamma'}$$

$$(If_{ff})\ \frac{\begin{array}{c}E \vdash a,M,\Gamma \Rightarrow v,s \quad isfalse(v) \\ \underline{pc'} = s \sqcup \underline{pc} \quad E \vdash c_2,M,\Gamma,\underline{pc'} \Rightarrow M_2,\Gamma_2 \\ \Gamma_2' = update(c_1,\underline{pc'},\Gamma_2)\end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2,M,\Gamma,\underline{pc} \Rightarrow M_2,\Gamma_2'}$$

$$(W_{tt})\ \frac{\begin{array}{c}E \vdash a,M,\Gamma \Rightarrow v,s \quad istrue(v) \\ \underline{pc'} = s \sqcup \underline{pc} \\ E \vdash c,M,\Gamma,\underline{pc'} \Rightarrow M',\Gamma' \\ E \vdash while\ (a)\ c,M',\Gamma',\underline{pc} \Rightarrow M'',\Gamma''\end{array}}{E \vdash while\ (a)\ c,M,\Gamma,\underline{pc} \Rightarrow M'',\Gamma''}$$

$$(Skip)\ E \vdash skip,M,\Gamma,\underline{pc} \Rightarrow M,\Gamma \qquad update(c,s,\Gamma) \triangleq \begin{cases}\Gamma(l) & \forall l \notin S_P(c) \\ \Gamma(l) \sqcup s & \forall l \in S_P(c)\end{cases}$$

Fig. 5. Information flow monitor big-step semantics of instructions

value of a_1. Second, s_2 considers explicit flows from the r-value of a_2. Third, $\underline{pc}$ captures the implicit flows generated by conditionals. Additionally, assignments generate pointer-induced flows from the l-value of a_1 to the set of possibly written locations. Consequently, the *update* operator propagates the union of $\underline{pc}$ and s_1 to $S_p(a_1 = a_2)$. Assuming that x points to a variable a stored at location l_a, Figure 6 illustrates the evaluation of instruction $*x = 1$. The security label s_x (resp. pc) affects the security label of variable a in order to take into account pointer-induced flows (resp. implicit flows). Finally, the *update* operator propagates the security label s' to the set $S_P(*x = 1)$ to capture pointer-induced flows due to the assignment $*x = 1$.

$$(Assign)\ \frac{\begin{array}{c}E \vdash *x,M,\Gamma \Leftarrow l_a,s_x \quad E \vdash 1,M,\Gamma \Rightarrow 1,public \quad s = s_x \sqcup public \sqcup \underline{pc} \\ s' = s_x \sqcup \underline{pc} \quad M' = M[l_a \mapsto 1] \quad \Gamma'' = \Gamma[l_a \mapsto s] \quad \Gamma' = update(*x = 1, s', \Gamma'')\end{array}}{E \vdash *x = 1, M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'}$$

Fig. 6. An example of instruction $*x = 1$ evaluation

For conditionals (rules If_{tt} and If_{ff}), a new context of execution $\underline{pc'}$ takes into account implicit flows generated by the conditional guard a. When a is evaluated to *true* (rule If_{tt}, the other one is symmetrical), the resulting security memory takes into account the implicit flows induced by both the executed branch c_1 and the non-executed one c_2. Implicit flows in c_1 are computed by the evaluation of c_1 in $\underline{pc'}$, whereas the *update* operator handles the ones from c_2 by propagating $\underline{pc'}$ to the set $S_P(c_2)$. Rules W_{tt} and W_{ff} are similar to conditional rules. Finally, a sequence of instructions $c_1;c_2$ is executed in the same execution context (rule *Comp*).

Soundness. In order to formalize TINI, Definition 1 introduces an equivalence relation for memories: two memories M_1 and M_2 are s-equivalent if they are equal for the set of locations l whose label $\Gamma(l)$ is at most s.

Definition 1 (Equivalence relation $\sim^s_\Gamma$). *For all Γ, $s \in \mathbb{S}$, M_1, M_2, M_1 and M_2 are s-equivalent ($M_1 \sim^s_\Gamma M_2$) if and only if*

$$\forall l \in Loc, \Gamma(l) \sqsubseteq s \implies M_1(l) = M_2(l).$$

Non-interference, by Definition 2, ensures that an attacker knowing only inputs and outputs up to a security level s cannot gain any knowledge of inputs whose security levels are strictly higher than s.

Definition 2 (Termination-insensitive non-interference).
For all $c, E, \Gamma, M_1, M'_1, M_2, M'_2, s, \underline{pc} \in \mathbb{S}$, such that
$E \vdash c, M_1, \Gamma, \underline{pc} \Rightarrow M'_1, \Gamma'_1$ and $E \vdash c, M_2, \Gamma, \underline{pc} \Rightarrow M'_2, \Gamma'_2$,

$$M_1 \sim^s_\Gamma M_2 \implies \Gamma'_2 = \Gamma'_1 = \Gamma' \text{ and } M'_1 \sim^s_{\Gamma'} M'_2.$$

This definition of non-interference is termination-insensitive since it ignores behaviors of diverging runs, including information leaks due to the attacker ability to observe (non-)termination of programs. Definition 2 is equivalent to the definitions of TINI in the literature [3,4,6,11]. Moreover, our definition of non-interference is equivalent to what Askarov et al. [4] call batch job TINI, since attackers are not allowed to know intermediate results of computation through outputs.

Theorem 1 (Soundness). *The information flow-extended semantics is sound with respect to termination-insensitive non-interference as defined in 2.*

Theorem 1 proves that our monitor semantics is sound with respect to TINI. The proof, by induction on instructions evaluation $\Rightarrow$, relies on the fact that both l-value and r-value evaluations of expressions in s-equivalent memories yield the same result for expressions whose label is below s. This theorem also proves that attackers cannot learn information by observing the behavior of our monitor since it ensures that both output security memories are equal. Full details of our proofs can be found in the technical report [12].

4 Program Transformation

This section presents an inlining approach for our monitoring semantics as a program transformation. This approach has the benefits of enabling both static and dynamic analysis since both analyses can be considered depending on the required level of confidence. The former would focus on soundness by ensuring that all execution paths of the analysed program are secure. The latter would emphasize on permissiveness by enforcing non-interference for the execution path of a single run.

Informally, the program transformation maps a shadow variable —a security label— to each variable of $Var(P)$, the set of variables of the initial program P. Inlining our

monitor then consists of propagating those security labels with respect to the monitor semantics. For this reason, types of our language are extended with a type τ_s representing security labels. Expressions are extended with security labels denoted s and a join operator $\sqcup$ on security labels. The range of memories M is also extended to $\mathbb{V} \cup \mathbb{S}$.

In order to handle pointers, we introduce in Definitions 3 and 4 the depth $\mathscr{D}(id)$ of a variable id and a bijection $\Lambda(id,k)$, with $k \in [0, \mathscr{D}(id)]$. $\mathscr{D}(id)$ is the number of dereferences such that $*^{\mathscr{D}(id)}id$ yields a basic type κ, whereas Λ maps each initial variable id to $\mathscr{D}(id)$ different shadow variables. Basically, $*^k\Lambda(id,k)$ is the security label of $*^k id$.

Definition 3 (Depth $\mathscr{D}(x)$ of variable x).

Let τ_x be the type of variable $x \in Var(P)$. $\mathscr{D}(x) = \mathscr{D}(\tau) = \begin{cases} 0 & \text{if } \tau = \kappa \\ 1 + \mathscr{D}(\tau') & \text{if } \tau = ptr(\tau') \end{cases}$

Definition 4 (Bijection Λ).

$\Lambda : \{(x,k) : x \in Var(P)$ and $k \in [0, \mathscr{D}(x)]\} \to Var'$ such that $Var' \subset Var \setminus Var(P)$ is a bijection mapping to each initial variable x exactly $\mathscr{D}(x)$ shadow variables, denoted $\Lambda(x,k)$, such that $\Lambda(x,k)$ has a type $ptr^{(k)}(\tau_s)$.

We extend Λ to all l-value expressions ($\Lambda(*^r x,k) \triangleq *^r\Lambda(x,k+r)$) such that $\Lambda(*^k id,0)$ is equal to $*^k\Lambda(id,k)$. Hence $\Lambda(*^k id,0)$ also captures the security label of $*^k id$.

Our program transformation, denoted T, maintains a pointer-related invariant in order to correctly handle aliasing. Essentially, if x points to an integer variable a, shadow variable $\Lambda(x,1)$ also points to $\Lambda(a,0)$. This way, whenever we read (or write) the same integer through $*x$ or a, we also read (or write) the same security label through either $*\Lambda(x,1)$ or $\Lambda(a,0)$. Listings 1 and 2 illustrate an example of our program transformation. Instruction 3 in Listing 1 is transformed into instructions 3, 4, 5 and 6 in Listing 2. Instructions 3, 5 and 6 of the transformed program reproduce the semantics of *Assign* rules as defined in the monitoring semantics (Figure 5), whereas instruction 4 maintains the aliasing invariant. Thanks to instructions 4 and 9 of the transformed program, instruction 13 updates the correct security label during execution.

Listing 1. The initial program.

```
1 //S_P(c_3)=S_P(c_5)={E(x)}
2 if (secret)
3   x = &a;
4 else
5   x = &b;
6 //S_P(c_7)={E(a),E(b)}
7 *x = 1
```

Listing 2. The transformed program.

```
1  pc'=pc⊔Λ(secret,0);
2  if (secret) {
3    Λ(x,0) = public;
4    Λ(x,1) = &Λ(a,0);
5    Λ(x,0) = Λ(x,0)⊔pc';
6    x = &a;
7  } else {
8    Λ(x,0) = public;
9    Λ(x,1) = &Λ(b,0);
10   Λ(x,0) = Λ(x,0)⊔pc';
11   x = &b;
12 }
13 *Λ(x,1) = Λ(x,0)⊔public;
14 Λ(a,0) = Λ(x,0)⊔pc;
15 Λ(b,0) = Λ(x,0)⊔pc;
16 *x = 1
```

As in Definition 5, two expressions are aliased in memory M if their l-value evaluation yields the same location. Hence, the aliasing invariant, stated as Lemma 1, ensures that two l-value expressions are aliased iff their shadow variables are aliased.

Definition 5 (Aliasing equivalence relation $\sim^M_{lval}$).

For all $a_1, a_2 \in Exp$, for all E, M such that $E \vdash a_1, M \Leftarrow l_1$ and $E \vdash a_2, M \Leftarrow l_2$.

$$a_1 \sim^M_{lval} a_2 \iff l_1 = l_2$$

Lemma 1 (Aliasing invariant).
For all $E, c, M, M', \Gamma, \Gamma', \underline{pc}, pc$ *such that* $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.
Let the predicate $\Omega(M) \triangleq \forall x, y \in Var(P)$, *for all* $r \in [0, \mathscr{D}(y)]$,

$$x \sim^{M}_{lval} *^{r} y$$

$$\iff \forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M}_{lval} \Lambda(*^{r} y, k)$$

Then $\Omega(M) \implies \Omega(M')$.

Transformation T relies on Definition 6 of operators $\mathcal{L}_L$, $\mathcal{L}_R$ and $\mathcal{L}$ which express security labels of expressions in terms of shadow variables. They respectively capture the label of the l-value of a, the label of the r-value of a, and $\Gamma(l_a)$, where l_a is the location of a. They accurately reproduce the monitoring semantics for expressions as defined in Figure 3.

Definition 6 (Operators $\mathcal{L}_L$, $\mathcal{L}_R$ and $\mathcal{L}$).

$$\mathcal{L}_R(n) \triangleq public \qquad \mathcal{L}_R(uop\ a) \triangleq \mathcal{L}_R(a) \qquad \mathcal{L}_R(\&a) \triangleq \mathcal{L}_L(a)$$

$$\mathcal{L}_R(a_1\ bop\ a_2) \triangleq \mathcal{L}_R(a_1) \sqcup \mathcal{L}_R(a_2) \qquad \mathcal{L}_R(a) \triangleq \mathcal{L}_L(a) \sqcup \mathcal{L}(a) \qquad \mathcal{L}(a) \triangleq \Lambda(a,0)$$

$$\mathcal{L}_L(id) \triangleq public \qquad \mathcal{L}_L(*a) \triangleq \mathcal{L}_R(a)$$

The l-values of a variable id is associated with the security label $public$ (rule LV_{ID}), so does $\mathcal{L}_L(id)$. $\mathcal{L}_L(*a)$, the security label associated to the l-value $*a$, is defined as $\mathcal{L}_R(a)$, the security label associated to the r-value of a (rule LV_{MEM}). As for r-values (rule RV_{CONST}), the security label of constant integers $\mathcal{L}_R(n)$ is defined as $public$. The security label of r-values expressions $\mathcal{L}_R(a)$ is defined as the join of their l-value label $\mathcal{L}_L(a)$ and the label of their content $\mathcal{L}(a)$ (rule RV) in order to take into account both pointer-induced and explicit flows. $\mathcal{L}_R(\&a)$, the label of r-value expressions $\&a$ is defined as $\mathcal{L}_L(a)$, the label of the l-value a (rule RV_{REF}). $\mathcal{L}_R(uop\ a)$ and $\mathcal{L}_R(a_1\ bop\ a_2)$ are respectively defined according to rules RV_{UOP} and RV_{BOP}. Finally, the label $\mathcal{L}(a)$ associated to the content of a is defined as $\Lambda(a,0)$, which represents $\Gamma(l_a)$ in the monitoring semantics. Figure 7 illustrates the computation of the label associated to a r-value $*x$. Intuitively, for the transformation to be correct, we must ensure that the evaluation of $\Lambda(x,0)$ and $*\Lambda(x,1)$ in M respectively results in $s_x = \Gamma(l_x)$ and $s_a = \Gamma(l_a)$.

$$\begin{aligned}\mathcal{L}_R(*x) &= \mathcal{L}_L(*x) \sqcup \mathcal{L}(*x) = \mathcal{L}_R(x) \sqcup \Lambda(*x,0) = \mathcal{L}_L(x) \sqcup \mathcal{L}(x) \sqcup *\Lambda(x,1) \\ &= public \sqcup \Lambda(x,0) \sqcup *\Lambda(x,1)\end{aligned}$$

Fig. 7. An example of security label computation by both semantics and transformation T

We present the program transformation rules in Figure 8. For brevity, $c_k; \forall k \in [0,n]$ denotes the sequence of instructions $c_0; c_1; \ldots c_n$. Since the transformation T must maintain the execution context and must propagate it to all possibly written locations in non-executed branches, it creates for each conditional and loop a new shadow variable of

type τ_s, denoted pc'. Variable pc' captures the new execution context $\underline{pc'}$ defined in the semantics. The transformation then parameterizes the branches with the new shadow variable pc'. It also uses the inverse of environment E, denoted E^{-1}, in order to find the set of variables corresponding to the locations $l \in S_P(c)$. Then it propagates the execution context pc' to all the corresponding shadow variables. This way, the program transformation reproduces the semantics of the *update* operator for conditionals and loops. Note that E^{-1} is well defined since each location has only one corresponding declared variable. We are confident that even for further extensions including dynamically allocated locations, we should be able to find a corresponding shadow expression if there is an expression pointing to that location.

$$T[skip, pc] \mapsto skip \qquad T[c_1;\ c_2, pc] \mapsto T[c_1, pc];\ T[c_2, pc]$$

$$T[a_1 = a_2, pc] \mapsto \begin{cases} \Lambda(a_1, 0) = \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc; \\ \Lambda(a_1, k) = \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup pc; \forall l \in S_P(a_1 = a_2) \\ a_1 = a_2; \end{cases}$$

$$T[if\ (a)\ c_1\ else\ c_2, pc] \mapsto \begin{cases} pc' = \mathcal{L}_R(a) \sqcup pc; \\ if\ (a)\ \{ \\ \quad T[c_1, pc'] \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, pc']; \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

$$T[while\ (a)\ c, pc] \mapsto \begin{cases} while\ (a)\ \{ \\ \quad pc' = \mathcal{L}_R(a) \sqcup pc; \\ \quad T[c, pc']; \\ \} \\ pc' = \mathcal{L}_R(a) \sqcup pc; \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c) \end{cases}$$

Fig. 8. Program transformation semantics

For assignments $a_1 = a_2$, the program transformation propagates three security labels to the shadow expression of a_1 according the monitor semantics. Since assignments create new aliasing relations, transformation T also generates $\mathscr{D}(a_1)$ assignments to maintain the aliasing invariant stated in Lemma 1. Finally, T uses E^{-1} and Λ to find shadow variables corresponding to locations in $S_P(c)$ and taints them with the security label $\mathcal{L}_L(a_1) \sqcup pc$.

The transformed program $T(P)$ is behaviourally equivalent to the initial program P. Let $E|_{var(P)}$ (resp. $M|_{Loc(P)}$ and $\Gamma|_{Loc(P)}$) be the restriction of environment E (resp.

of memory M and Γ) to the set $Var(P)$ of initial variables (resp. to the set $Loc(P)$ of initial locations). More precisely, Theorem 2 states that for any terminating run, executions of P and $T(P)$ in equal input memories for initial locations $Loc(P)$ result in equal memories for those same locations. The proof by induction on instructions evaluation relies on the fact that program transformation T introduces only assignments handling shadow variables. Hence, those additional assignments do not modify neither values nor security labels associated to the set $Loc(P)$ of initial locations.

Theorem 2 (Initial semantics preservation). *For all* $c, E, M, \Gamma, \underline{pc}, pc$ *such that:* $E|_{Var(P)} \vdash c, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M_1, \Gamma_1$ *and* $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$.

$$\textit{Then, } M_2|_{Loc(P)} = M_1 \textit{ and } \Gamma_2|_{Loc(P)} = \Gamma_1.$$

Theorem 3 proves the soundness of the transformation T with respect to the monitor semantics presented in Figure 5. Informally, the theorem supposes that values of shadow variables (resp. execution context variable pc) are initialized according to the initial security memory Γ (resp. execution context $\underline{pc}$). Then after the execution of the transformed instructions, it states that the values of shadow variables capture the exact values of the output security memory.

Theorem 3 (Sound monitoring of information flows). *Let c, for all* $E, M, \Gamma, M', \Gamma'$ *such that* $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.

Let us define the predicate $\Upsilon(E, M, \Gamma) \triangleq$ *for all* $x \in Var(P)$, *for all* $k \in [0, \mathscr{D}(x)]$, $E \vdash *^k x, M \Leftarrow l_{xk}$ *and* $\Gamma(l_{xk}) = s_{xk} \implies E \vdash *^k \Lambda(x,k), M \Rightarrow s_{xk}$.

$$\textit{The following result holds: } \Upsilon(E, M, \Gamma) \textit{ and } E \vdash pc, M \Rightarrow \underline{pc} \implies \Upsilon(E, M', \Gamma').$$

As the program transformation is sound with respect to our information flow monitor semantics, it is also sound wrt. TINI. Therefore, we can soundly reason about information flows through security labels defined by this program transformation. To our knowledge, that is the first proof of soundness for inlining information flow monitors handling pointers with aliasing. The proof, by induction on instructions evaluation $\Rightarrow$, heavily relies on the aliasing invariant stated in Lemma 1.

TINI verification. Figure 9 shows that the program transformation T can be used to verify TINI through both dynamic and static analysis. Assuming the implementation of security labels and their join operator, running the self-monitoring program $T(P)$ enforces TINI dynamically —actually, this is a hybrid approach since the monitor relies on a prior static analysis S_P— for single execution paths. This dynamic approach has the advantage of being permissive since it ignores possible unsecure paths that are not executed. It also enables dynamic loading of security policies [13], taking into account eventual updates. The transformation T also enables the verification of TINI by static analysis: for instance, off-the-shelf abstract interpretation tools can compute an over-approximation of $T(P)$ semantics for all execution paths, without implementing new abstract domains. While still being more permissive than traditional type systems, such an approach freezes the enforced security policy. Yet, it enhances our confidence in the analyzed program. It also completely lifts the burden of runtime overhead.

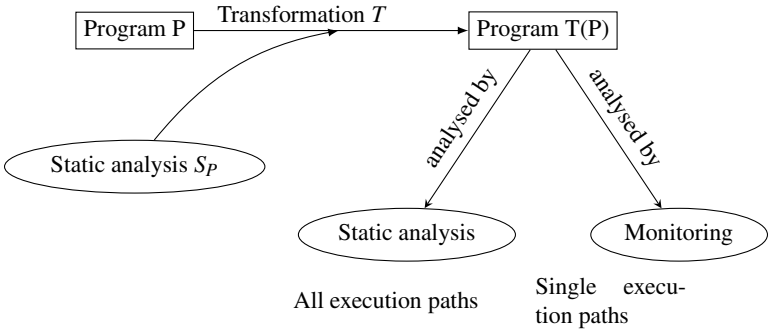

Fig. 9. Non-interference verification using the program transformation T

5 Related Work

Information flow monitors. Le Guernic et al. [7] formalize a sound flow-sensitive monitor for a simple imperative language with outputs. Le Guernic's monitor combines both static and dynamic analysis in order to enforce TINI. It is based on edit automata [14], which are monitors enforcing a security policy by modifying program actions, namely changing secret outputs to default values in Le Guernic's monitor. Extending our approach with outputs is straightforward. Le Guernic et al. suggest that their monitor can be implemented as a program transformation or a virtual machine (VM).

Russo and Sabelfeld [5] parameterize their hybrid monitor for a simple imperative language by different enforcement actions (default, failstop or suppress). They also prove the necessity to rely on static analysis to soundly monitor information flows while still being more permissive than Hunt-Sands-style [3] flow-sensitive type systems. Unlike monitors based on Russo and Sabelfeld's one, we use a big-step semantics. Hence, we neither need to maintain a stack of security labels for execution contexts, nor insert instructions to notify the monitor at the immediate postdominator of each conditional.

Moore and Chong [15] extend the VM-like monitor of Russo and Sabelfeld with dynamically allocated references, allowing different sound memory abstractions. In our semantics, we use the most precise instantiation of their memory abstraction where each concrete location correspond to one abstract location. While it is undecidable in the general case to determine which locations might be updated by an instruction, we argue that, for the sake of permissiveness, it is necessary to be as precise as possible at least for the set of finite statically allocated locations.

Austin and Flanagan [11] investigate a purely dynamic monitor for a λ-calculus language with references. Their monitor supports a limited flow-sensitivity since it implements a conservative no-sensitive upgrade policy; the monitor stops the execution when assigning a public variable in a secret context. Thus, their monitor is proven sound without having to rely on static analyses. Austin and Flanagan [16] also enhance their monitor by a permissive-upgrade approach; their monitor labels public data that is assigned in secret contexts as partially leaked, then soundly forbid branching on those data. Our monitor is fully flow-sensitive, hence more permissive.

Sound inlining. Chudnov and Naumann [17] design a sound monitor inlining approach based on Russo and Sabelfeld's monitor. As they aim at monitoring information flows for Javascript, they argue that VM monitors are impractical because of just-in-time compilation. Their language supports output instructions but no references. We also believe that inlining is necessary when the language is compiled rather than interpreted.

Magazinius et al. [18] investigate sound inlining of security monitors for an imperative language supporting dynamic code evaluation but no references. Their monitor is purely dynamic since it uses a no-sensitive upgrade policy as in Austin and Flanagan [11]. Our program transformation approach can also be applied for such a policy in order to soundly monitor information flows for richer languages, including pointers.

6 Conclusion and Future Work

We have formalized a sound flow-sensitive information flow monitor handling pointers and aliasing. We have also inlined our monitor through a program transformation proven sound with respect to our monitor semantics, hence with TINI. Our program transformation enables permissive yet sound enforcement of TINI by both dynamic and static analyses. Our monitor semantics ignores diverging runs since it is inspired by a simple version of the Clight big-step semantics stripped of coinduction [8]. As pointed by Le Guernic [7], this is not problematic when dealing with TINI because we ignore non-termination covert channels.

As we aim to support a large subset of the C language, we plan on extending both the semantics and the program transformation with richer C constructs. We are currently implementing our program transformation as a Frama-C plug-in, an open-source tool for modular analysis of C programs [19]. Frama-C enables the design of powerful analyses relying on the collaboration of off-the-shelf plug-ins. We are going to rely on Value Analysis [20], an abstract interpretation plug-in of Frama-C, in order to compute a correct approximation $S_P(c)$, of the set of locations that might be updated by an instruction c. Frama-C also supports ACSL [21], a formal specification language for C programs. This language can allow us to handle declassification annotations.

Acknowledgement. We would like to thank Sébastien Bardin for his valuable comments.

References

1. Denning, D., Denning, P.: Certification of Programs for Secure Information Flow. Communications of the ACM 20(7), 504–513 (1977)
2. Goguen, J., Meseguer, J.: Security Policies and Security Models. In: IEEE Symposium on Research in Security and Privacy (1982)
3. Volpano, D., Irvine, C., Smith, G.: A Sound Type System for Secure Flow Analysis. Journal in Computer Security 4(2-3), 167–187 (1996)
4. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-Insensitive Noninterference Leaks More Than Just a Bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008)

5. Russo, A., Sabelfeld, A.: Dynamic vs. Static Flow-Sensitive Security Analysis. In: Computer Security Foundations Symposium, pp. 186–199. IEEE (2010)
6. Hunt, S., Sands, D.: On Flow-Sensitive Security Types. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, vol. 41, pp. 79–90. ACM (2006)
7. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-Based Confidentiality Monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)
8. Blazy, S., Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language. Journal of Automated Reasoning 43(3), 263–288 (2009)
9. Leroy, X.: Formal Verification of a Realistic Compiler. Communications of the ACM 52(7), 107–115 (2009)
10. Sabelfeld, A., Myers, A.: Language-Based Information-Flow Security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
11. Austin, T., Flanagan, C.: Efficient Purely-Dynamic Information Flow Analysis. ACM Sigplan Notices 44(8), 20–31 (2009)
12. Assaf, M., Signoles, J., Tronel, F., Totel, E.: Program Transformation for Non-interference Verification on Programs with Pointers. Research report RR-8284, INRIA (April 2013), http://hal.inria.fr/hal-00814671
13. Chandra, D., Franz, M.: Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, pp. 463–475. IEEE (2007)
14. Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. International Journal of Information Security 4(1), 2–16 (2005)
15. Moore, S., Chong, S.: Static Analysis for Efficient Hybrid Information-Flow Control. In: 2011 IEEE 24th Computer Security Foundations Symposium (CSF), pp. 146–160. IEEE (2011)
16. Austin, T.H., Flanagan, C.: Permissive Dynamic Information Flow Analysis. In: PLAS 2010: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pp. 1–12. ACM (2010)
17. Chudnov, A., Naumann, D.: Information Flow Monitor Inlining. In: 2010 23rd IEEE Computer Security Foundations Symposium (CSF), pp. 200–214. IEEE (2010)
18. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly Inlining of Dynamic Security Monitors. Computers & Security (2011)
19. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Program Analysis Perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
20. Cuoq, P., Prevosto, V., Yakobowski, B.: Frama-C's Value Analysis Plug-in (September 2012), http://frama-c.com/download/frama-c-value-analysis.pdf
21. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (September 2012), http://frama-c.cea.fr/acsl.html