Hybrid Monitoring of Attacker Knowledge

Frédéric Besson, Nataliia Bielova and Thomas Jensen
Inria, France

***Abstract*—Enforcement of noninterference requires proving that an attacker's knowledge about the initial state remains the same after observing a program's public output. We propose a hybrid monitoring mechanism which dynamically evaluates the knowledge that is contained in program variables. To get a precise estimate of the knowledge, the monitor statically analyses non-executed branches. We show that our knowledge-based monitor can be combined with existing dynamic monitors for non-interference. A distinguishing feature of such a combination is that the combined monitor is provably more permissive than each mechanism taken separately. We demonstrate this by proposing a knowledge-enhanced version of a no-sensitive-upgrade (NSU) monitor. The monitor and its static analysis have been formalized and proved correct within the Coq proof assistant.**

I. Introduction

Information-flow control provides a promise of a strong information security property [24]. Today most research has focused on *monitors for noninterference* [4], [5], [18], [25], that block executions where secret inputs flow into public outputs. Such flow can happen due to explicit or implicit information flow. An explicit flow occurs when *secret* information is stored in a *public* variable visible to an attacker. An implicit flow happens when assignments to public variables are made under *secret control* (*i.e.*, following a test on a secret variable), like in Program 1.[1]

```
1 l = 0; if (h) then l = 1; output l
```
Program 1

There is an implicit flow from h to l because by observing the value of l the attacker can deduce the secret value h.

Dynamic monitors control one execution of the program and propagate a security label to each program variable. If the monitor suspects a possible flow (explicit or implicit) from secret inputs to a variable labeled as public, it blocks the execution. Such *purely dynamic information flow control* was first proposed by Fenton [11] and has recently regained interest [5], [25] for (at least) two reasons. First, some languages, such as JavaScript, are so dynamic that a precise static analysis is practically impossible. Therefore, an attractive alternative is to resort to dynamic monitoring, either by extending an interpreter for the language or by inlining a monitor in the program. Second, even if a program may have some insecure executions, there may be other executions that are perfectly secure. While a static analysis would reject such programs, dynamic monitors can identify and allow some secure executions of insecure programs.

However, dynamic monitors also have several limitations, due to the fact that they analyse only one execution of a program. As a result, they make the worst-case assumption about what happens later on in the current execution, and what could happen in other executions. Dynamic information flow control first proposed by Zdancewic [25] and later used by Austin and Flanagan [4] is based on the *no-sensitive-upgrade* (NSU) principle: it halts an execution when a public variable gets assigned under secret control. This principle severely limits the *permissiveness* of dynamic monitors in certain cases.

- They may block executions too early: if later a variable is updated, then there is no information leakage.

```
1 if (h) then l = 1;
2 l = 0;
3 output l
```
Program 2

Program 2 is secure but its execution is blocked by NSU when $h = true$.

- If the variable is assigned the same value on both branches, there is no leakage. Program 3 is secure but NSU blocks all its executions.

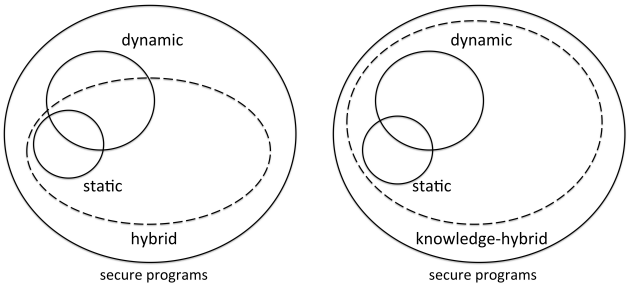
```
1 x = 1;
2 if (h) then l = 1 else l = x;
3 output l
```
Program 3

The first problem of blocking execution too early was addressed by the *permissive-upgrade* (PU) principle [5], by introducing a special *"partially leaked"* label. The second problem requires knowledge about other executions and has motivated a strand of research in hybrid information flow monitors that combine static and dynamic analysis.

Hybrid monitors (e.g. [7], [17], [18], [23]) analyse the source code of each non-executed branch under secret control to detect possible implicit flows. A dynamic monitor can be enhanced with a variety of static analyses. Le Guernic *et al.* [17]–[19] proposed the first combination of dynamic information flow monitors with static dependency analyses. Besson *et al.* [7] extended this work to a more sophisticated constant propagation and dependency analysis. Recently, Hedin *et al.* have shown how to improve their dynamic information flow analysis for JavaScript [14] with a points-to analysis [12].

This research has been partially supported by the French ANR projects AJACS ANR-14-CE28-0008 and ANR-10-LABX-07-01 Laboratoire d'excellence Comin Labs.

[1] In all examples, variables with names starting with "h" are secret, and all the other variables are public.



(a) from Hedin *et al.* [12]. (b) our result

Fig. 1: Relative permissiveness revisited.

Permissiveness. All the monitors discussed above provably enforce noninterference, however some of them may block more program executions than others. Intuitively, *permissiveness* defines how many program executions are accepted by the monitor even if the program may be insecure. Hedin *et al.* [12] have proven that purely dynamic and hybrid monitors are incomparable in their permissiveness. For example, a dynamic NSU monitor allows execution of Program 1 when $h = \mathit{false}$, while the hybrid monitor stops it. On the other hand, the dynamic NSU monitor will stop execution of Program 2 when $h = \mathit{true}$ while the hybrid monitor will allow it. Figure 1a graphically shows the secure programs, and programs, for which the static, dynamic and hybrid analysis identify as secure all its executions (from Hedin *et al.* [12]).

In this paper, we propose a knowledge-based hybrid monitor that is able to reach a level of permissiveness that was deemed impossible for standard hybrid monitors. Figure 1b graphically shows that all the executions of all the secure programs that are accepted by dynamic monitors, are also accepted by an enhanced version of our knowledge-based hybrid monitor.

Modelling attacker knowledge. Basic information flow control detects whether or not a program execution may leak information, but will not provide a more precise description of *what information is being leaked*, or equivalently, *what knowledge an attacker gains from an observation*. However, switching to such knowledge-based analysis can provide a finer control over information flow. Previously [7], the authors have proposed a hybrid information flow control that combines dynamic monitoring and static analysis. This technique computes the leakage of a *concrete program execution* by labeling every program variable with a logical formula over the secret inputs. This formula is a logical description of the knowledge that an attacker can deduce about the initial state of the program when observing the value of a variable.

Program 4

```
1 x = 0; y = 0;
2 if (h1) then y = 1;
3 if (h2) then x = 1 else x = y;
4 output x
```

Consider Program 4 and its execution when $h1 = \mathit{false}$ and $h2 = \mathit{true}$. When Program 4 outputs 1, the attacker learns that either $h1$ or $h2$ was true:

$$\text{Attacker knowledge: } h1 \vee h2$$

Given the same initial memory where $h1 = \mathit{false}$ and $h2 = \mathit{true}$, the hybrid monitor of Besson *et al.* [7] associates the knowledge to the variable after analysing each test: the first test on $h1$ fails, thus the value of y remains unchanged and the knowledge of y is $\neg h1$; upon the second test, the monitor concludes that the value of x is 1 in the true branch and 0 in the false branch. As the values are not the same, the knowledge of the output $x = 1$ might depend on $h2$ and on the knowledge of y. Therefore, the knowledge computed by the monitor is:

$$\text{Approximated knowledge: } \neg h1 \wedge h2$$

As is readily seen, the approximated knowledge is much less precise than the real attacker knowledge. The reason for this gap between estimated and actual knowledge is in the choice of the model of knowledge domain. Intuitively, the knowledge in [7] is limited to a set of environments that can contribute to the *current value of* x.

A more expressive knowledge domain. In this paper, we propose a more general representation of attacker knowledge: the knowledge associated with a variable x is the set of environments that lead to a particular value of x *for several possible values of* x. In other words, the knowledge in x groups the initial environments into equivalence classes, such that two environments are equivalent if they lead to the same value of x. This models much more of the input-output relation of the program.

This new knowledge domain leads to several advantages over the existing previous work [7]. The first advantage is that it empowers the monitor to reason about several executions, and hence *to prove noninterference in more cases* than in previous work [7]. For a concrete example, see Example 1 in Section IV-B. With the previous knowledge representation in [7], we would infer that z depends on x and y; while with the new representation we prove that x and y do not interfere with z.

The second advantage of this more expressive knowledge domain is that it *enables a composition with other monitors*, and we demonstrate such composition with the no-sensitive upgrade (NSU) monitor. The new knowledge domain allows the hybrid monitor to reason about the other executions that would be blocked by the other dynamic monitor for non-executed paths. This leads to a composition of monitors that is strictly more permissive that each monitor separately. We summarise this result in Figure 1b, showing that the knowledge-based hybrid monitor accepts more secure executions than any purely dynamic monitor it is built upon[2].

[2]Notice that the permissiveness result with respect to static analysis is achieved by transitivity of permissiveness: a knowledge-based hybrid monitor is provably more permissive than a standard hybrid monitor, and a standard hybrid monitor is more permissive than a static analysis of Hunt and Sands [15] (see Thm. 3 of [23]).

The third advantage is that the proposed knowledge domain allows us to design a *more precise static analysis*. Our knowledge domain is used for both the dynamic analysis of the executed branch and the static analysis of non-executed branched. This gives a pleasant uniformity to the theory and provides a more general framework, compared to the *ad-hoc* static analyses for the non-executed branches in [7].

Contributions.

- We propose a hybrid monitor that computes the knowledge of the attacker. Our monitor combines a dynamic analysis with a static analysis of the non-executed branches. The knowledge domain allows the monitor to compute an attacker's knowledge more precisely than in previous works [7].
- The knowledge-based hybrid monitor is proved to be correct (it safely over-approximates the attacker's knowledge) and sound (it can be used to enforce non-interference). The proof has been formalized in the Coq proof assistant [1].
- The proposed monitor can be combined with existing dynamic or hybrid monitors for non-interference. A distinguishing feature of such a combination is that the combined monitor is provably more permissive than the monitor it builds upon.
- We have proposed an effective and symbolic representation of knowledge and implemented the computation of knowledge as part of our Coq formalization. The results reported in this paper are all computed with this implementation.

II. Attacker knowledge and non-interference

A. Attacker model

We consider a classical attacker model, following the definition of *gadget attacker* [6]. An attacker provides the program source code and this program runs in an environment that contains secret information, producing some outputs, observable to the attacker.

B. Attacker knowledge

Given an observation at the end of an execution, an *attacker knowledge* is the set of all possible input environments that can lead to that observation. This naturally induces an equivalence relation on input environments. Landauer and Redmond [16] propose a lattice of equivalence classes of environments for representing the knowledge of an attacker. Askarov and Sabelfeld [3] and Askarov and Chong [2] give a characterisation of non-interference in terms of attacker knowledge. The remainder of this section summarizes notions and results from these papers.

We assume a security policy in the form of a lattice of two security levels $(\{\mathsf{L},\mathsf{H}\},\sqsubseteq)$, where $\mathsf{L}\sqsubseteq\mathsf{H}$ and we use $\sqcup$ as the least upper bound. A labelling function Γ assigns security levels to all program variables. We write ρ_L for the L-projection of the environment ρ onto those variables x whose level is lower than L, *i.e.*, for which $\Gamma(x)\sqsubseteq\mathsf{L}$, and in the future notations we drop an implicit labelling function Γ. We write $[\rho]_\mathsf{L}$ for the set of environments that agree with ρ on low variables: $[\rho]_\mathsf{L}=\{\rho' \mid \rho_\mathsf{L}=\rho'_\mathsf{L}\}$.

The program semantics is given by a relation $(P,\rho)\downarrow v$, where P is a program that produces output v at the end of the execution. This output is visible to the attacker. The knowledge is defined as the set of low-equivalent environments that can produce the same output v.

Definition 1 (Attacker knowledge)**.** Given a program P, an initial environment ρ, and a final observation v, the *attacker knowledge* is the set of environments that agree with ρ on low variables and leads to the observation of v:

$$\mathcal{K}^{\downarrow}(P,v,\rho)=\{\rho' \mid \rho_\mathsf{L}=\rho'_\mathsf{L}\wedge(P,\rho')\downarrow v\}.$$

Notice that a smaller knowledge set represents fewer possible inputs that produce the same program output, thus a smaller set corresponds to a bigger amount of information. Therefore, a smaller knowledge set is a safe approximation of the actual attacker knowledge.

C. Termination-Insensitive Noninterference

In the following, we describe the relationship between knowledge and the standard notion of noninterference. We shall focus on *termination-insensitive noninterference (TINI)*, and hence restrict attention to a termination-insensitive version of knowledge that only considers environments in which the program terminates and where the program output is visible to the attacker. The knowledge obtained just from observing termination, given an initial observation ρ_L is called *initial attacker knowledge*.

Definition 2 (Initial attacker knowledge)**.** Given program P and an environment ρ, the *initial attacker knowledge* is:

$$\mathcal{I}^{\downarrow}(P,\rho)=\{\rho' \mid \rho_\mathsf{L}=\rho'_\mathsf{L}\wedge\exists v.(P,\rho')\downarrow v]\}.$$

Later on, we omit the superscript $\downarrow$ when it can be inferred from the context.

The security condition states that the attacker's knowledge should not grow with the new observation produced by the program execution.

Definition 3 (Knowledge-based security for input environment ρ)**.** Program P is *secure* for an initial input environment ρ if whenever $(P,\rho)\downarrow v$ then

$$\mathcal{K}(P,v,\rho)=\mathcal{I}(P,\rho).$$

Notice that if the program P does not terminate in an environment ρ, then P is considered secure for ρ. However, the program still might be insecure for any other low-equal environment, in which the program terminates.

The standard notion of termination-insensitive noninterference (TINI) is stated by comparing pairs of low-equivalent initial environments.

Definition 4 (TINI)**.** A program P is *termination-insensitively noninterferent (TINI)* if whenever $\rho^1_\mathsf{L}=\rho^2_\mathsf{L}$, and $(P,\rho^1)\downarrow v_1$ and $(P,\rho^2)\downarrow v_2$, then $v_1=v_2$.

Askarov and Sabelfeld [3, Prop. 2] have shown that there is an equivalence between the knowledge-based security and TINI for a lattice with two elements.

Lemma 1. *A program P satisfies TINI if and only if P is secure for all initial environments ρ.*

III. Preliminary definitions

A. Language

We use a simple untyped imperative language extended with a specific output command $\mathsf{output}\ x$ which evaluates the variable x and outputs its value. This output is visible to the attacker at security level L. All the commands of the language are standard, except perhaps for the $\mathsf{assume}(e)$ operator, which evaluates e and continues or halts an execution depending on whether its value is true or false. The syntax of this language is as follows:

$$\mathbb{P} \ni P ::= c; \mathsf{output}\ x \qquad \mathbb{E} \ni e ::= n \mid x \mid e_1 \oplus e_2 \mid \neg e$$
$$\mathbb{C} \ni c ::= \mathsf{skip} \mid x := e \mid c_1; c_2 \mid \mathsf{assume}(\mathrm{e}) \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c$$

The set of expressions contains the usual numeric and Boolean expressions. Every expression can be interpreted as a boolean value, and hence conditional commands take an arbitrary expression as condition. The exact interpretation of expressions as booleans is, however, not essential to the results in this paper and will be left unspecified. We use $\oplus$ to denote an arbitrary binary operator.

An environment $\rho \in Env = Var \to \mathbb{V}$ maps variables to values. The big-step program semantics is presented in Figure 2. The semantics of commands is denoted by a binary relation $(c, \rho) \downarrow \rho'$ meaning that command c when executed in environment ρ will evaluate to ρ' and the semantics of programs is denoted by $(P, \rho) \downarrow v$ meaning that program P when executed in environment ρ will produce an output v.

$$\textsc{skip}\ \frac{}{(\mathsf{skip}, \rho) \downarrow \rho} \qquad \textsc{assign}\ \frac{}{(x := e, \rho) \downarrow \rho[x \mapsto [\![e]\!]_\rho]}$$

$$\textsc{seq}\ \frac{(c_1, \rho) \downarrow \rho' \quad (c_2, \rho') \downarrow \rho''}{(c_1; c_2, \rho) \downarrow \rho''} \qquad \textsc{assume}\ \frac{C[\![e]\!]_\rho = tt}{(\mathsf{assume}(e), \rho) \downarrow \rho}$$

$$\textsc{if}\ \frac{C[\![e]\!]_\rho = \alpha \qquad (c_\alpha, \rho) \downarrow \rho'}{(\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}, \rho) \downarrow \rho'}$$

$$\textsc{while}\ \frac{(\mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}, \rho) \downarrow \rho'}{(\mathsf{while}\ e\ \mathsf{do}\ c, \rho) \downarrow \rho'}$$

$$\textsc{output}\ \frac{(c, \rho) \downarrow \rho' \qquad [\![x]\!]_{\rho'} = v}{(c; \mathsf{output}\ x, \rho) \downarrow v}$$

where $[\![x]\!]_\rho = \rho(x) \quad [\![n]\!]_\rho = n \quad [\![e_1 \oplus e_2]\!]_\rho = [\![e_1]\!]_\rho \oplus [\![e_2]\!]_\rho$

Fig. 2: Language semantics

Let $\mathbb{B}$ denote the set of boolean values. To accommodate the fact that any expression can be used as a condition, we assume a function $C : \mathbb{V} \to \mathbb{B}$ that specifies how each value is interpreted as a boolean. C satisfies the following two constraints which ensure that $\neg e$ represents the negation of the expression e and $e = e'$ models the fact that e and e' evaluates to the same value:

$$C([\![\neg e]\!]_\rho) = \neg C([\![e]\!]_\rho) \quad C([\![e = e']\!]_\rho) = ([\![e]\!]_\rho = [\![e']\!]_\rho)$$

B. Notations

Given sets A and V, we write $V^\sharp$ for $V \cup \{\bot, \top\}$. In domains of the form $V^\sharp$ we write $\lfloor v \rfloor$ to assert that v is an element that is neither $\bot$ nor $\top$. For a function $f \in A \to V^\sharp$, we write $f(a) = \lfloor v \rfloor$ for $f(a) = v \wedge v \notin \{\bot, \top\}$ and $f^{\text{-}1}(v) = \{a \mid f(a) = \lfloor v \rfloor\}$ for the pre-image of v.

Given a domain $V^\sharp$, we get a flat lattice $(V^\sharp, \preccurlyeq, \bot, \top)$ such that $\bot \preccurlyeq x$, $x \preccurlyeq \top$ and $\lfloor x \rfloor \preccurlyeq \lfloor x \rfloor$. Given x, y, we write $x \curlyvee y$ (resp. $x \curlywedge y$) the the least upper bound (resp. greatest lower bound) of x and y. The ordering, least upper bound and greatest lower bound are lifted to functions in the standard pointwise fashion: $f \preccurlyeq g$ iff $\forall x, f(x) \preccurlyeq g(x)$; $(f \curlyvee g)(x) = f(x) \curlyvee g(x)$; $(f \curlywedge g)(x) = f(x) \curlywedge g(x)$.

Given a unary operator $o : V \to W$, we define an operator $o^\sharp : V^\sharp \to W^\sharp$ as follows

$$o^\sharp(x) = \text{if } x \in \{\bot, \top\} \text{ then } x \text{ else } o(x).$$

Similarly, for a binary operator $\oplus : V \to V \to V$ we define $\oplus^\sharp : V^\sharp \to V^\sharp \to V^\sharp$ as

$$x \oplus^\sharp y = \begin{cases} v_1 \oplus v_2 & \text{if } x = \lfloor v_1 \rfloor \wedge y = \lfloor v_2 \rfloor \\ \bot & \text{if } x = \bot \vee y = \bot \\ \top & \text{otherwise.} \end{cases}$$

We define a binary operator $assume : \mathbb{B}^\sharp \to V^\sharp \to V^\sharp$ which returns its second argument if the first argument is either $true$ or $\top$, and undefined otherwise.

$$assume(b, v) = \text{if } (b = true \vee b = \top) \text{ then } v \text{ else } \bot.$$

Finally, we define a conditional operator $if : \mathbb{B}^\sharp \to V^\sharp \to V^\sharp \to V^\sharp$ built from $assume$, where $\neg^\sharp$ is a standard negation operator extended to the domain $\mathbb{B}^\sharp$:

$$if(b, v, v') = assume(b, v) \curlyvee assume(\neg^\sharp b, v').$$

Given $c : A \to \mathbb{V}^\sharp$ and $f, g \in A \to \mathbb{V}^\sharp$, we write $Assume(c, f)$ and $IF(c, f, g)$ for the assume and conditional operators lifted to functions:

$$\begin{aligned} Assume(c, f)(a) &= assume(C^\sharp(c(a)), f(a)) \\ IF(c, f, g)(a) &= if(C^\sharp(c(a)), f(a), g(a)). \end{aligned}$$

where $C^\sharp$ is the function C lifted to the domain $\mathbb{V}^\sharp \to \mathbb{B}^\sharp$.

IV. A hybrid knowledge-based monitor

Our hybrid information flow analysis computes the knowledge of a program's input-output behaviour that the attacker obtains by observing the result of a given execution of the program. More precisely, we shall define the domain $\mathbf{K}$ of knowledge to be the set of functions K that maps environments ρ to values v, with the intention that $K(\rho) = v$ if the program when started in

initial environment ρ will either produce an output v or not terminate. If the hybrid monitor from initial state ρ calculates final value v and knowledge K, then $K^{-1}(v)$ is a safe approximation of the set of all the environments that produce v. This set will allow us to safely approximate the attacker's knowledge (see Theorem 1).

Definition 5 (Knowledge)**.** The domain $\mathbf{K}$ of knowledge is defined by:

$$\mathbf{K} = \mathit{Env} \to \mathbb{V}^\sharp.$$

Notice that the monitor may compute knowledge $K \in \mathbf{K}$ that will map an environment to $\bot$ or $\top$. This is due to the presence of the static analysis. If $K(\rho) = \bot$ then the static analysis has established that computation started in ρ will not terminate. On the other hand, $K(\rho) = \top$ means that approximations in the static analysis made it impossible for the monitor to determine what value will result from an execution starting in ρ. This means that ρ cannot be added to the knowledge set $K^{-1}(v)$ for any possible output v. Recall that a knowledge analysis may always safely under-approximate the knowledge set $K^{-1}(v)$ of output v, so having $K(\rho) = \top$ for some ρ makes the knowledge analysis more conservative than an analysis that is capable of determining the exact output for ρ.

A. Monitor semantics

We now define a hybrid knowledge monitor that combines a dynamic monitor with a static analysis. The hybrid monitor executes the program and, at the same time, computes an over-approximation of the knowledge of the initial state that can be deduced from the current state at a given point in the execution. The semantic state of the hybrid monitor is thus a pair (ρ, κ), where the first component is either an environment Env containing the current values of the variables, or an empty environment, $\cdot$, that will be used by the static analysis. The second component $\kappa \in \mathit{Var} \to \mathbf{K}$ is an environment containing the knowledge present in each variable. At branching points, the monitor will execute one branch and will statically analyse the other, non-executed branch. This static analysis will help refining the computation of the actual knowledge stored in variables.

Given a concrete initial environment ρ, the initial state of the hybrid monitor $\mathit{init}(\rho)$ is such that each variable x has the knowledge of the current value of x in the initial environment: $\mathit{init}(\rho) = (\rho, \kappa_0)$ with $\kappa_0 = \lambda x.\lambda\rho'.\rho'(x)$. To see this, suppose that the program immediately outputs the value v of variable x *i.e.* $v = \rho(x)$. The set of environments $\{\rho' \mid \rho'(x) = v\}$ that produce v is modeled exactly by $\kappa_0(x)^{-1}(v)$.

From the initial state, the monitor executes according to the rules of Figure 3. The concrete execution and the static analysis are combined into one reduction relation $\Downarrow$. The rules SKIP and SEQ are standard. For the other language constructs, there are two rules: a dynamic rule describing the monitored execution of the construct, and a static rule describing the static analysis of it. The dynamic rules will operate on environments with the actual values of the variables. The static analysis, on the other hand, is intended to provide information about all other possible executions so it will not have information about concrete values. In the formalization, this means that the static rules apply only when the environment is undefined (denoted by $\cdot$).

The two rules ASSIGNDYN and ASSIGNSTAT for assignment use the function $(\!|\_|\!)_\kappa$ to evaluate the knowledge about the initial environment contained in the value of the expression e. The function takes the current knowledge environment κ as parameter. In addition, the dynamic rule updates the value of x in the environment ρ.

The rule IFDYN describes the monitored execution of conditional statements of form $\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}$. The outcome of the test α is the value of the expression e computed in the environment ρ and the appropriate branch is executed with that environment, producing a new environment ρ' and a new knowledge environment κ_α. The non-executed branch $c_{\bar{\alpha}}$ is statically analysed, using an undefined environment of values and the current knowledge environment. The knowledge environment $\kappa_{\bar{\alpha}}$ obtained from this static analysis must be combined with the knowledge environment from the execution κ_α. To this end, we construct the function $\mathbb{IF}((\!|e|\!)_\kappa, \kappa_{tt}, \kappa_{ff})$ that uses a conditional operator $\mathit{IF}(c, f, g)$ from Section III-B. We later show that in programs without loops the IF operator allows us to precisely model the attacker's knowledge.

For the while loop, the dynamic rules WHILEDYNTRUE and WHILEDYNFALSE are standard unfolding semantic rules that apply when the environment is defined. The static rule WHILESTAT states that any s' whose knowledge safely approximates the knowledge before entering the loop (condition $s \preccurlyeq s'$) as well as the knowledge after executing the body of the loop (condition $s_1 \preccurlyeq s'$) is a valid result of the static analysis of the loop. The rule leaves room for an actual implementation to compute more or less precise approximations of the attacker knowledge after a loop. Our implementation (Section VII) employs an iterative fixpoint computation to this end.

We do not define a rule for analysing the output command $\mathsf{output}\ x$, because it does not change the knowledge of any variable. The $\mathsf{output}\ x$ command is important because it is at this point that we must decide what to output and, hence, what security property to enforce. In Section V, we shall propose rules for the output command that will enforce enforce non-interference.

B. Examples

To illustrate the expressive power of our hybrid monitor, we provide a number of examples. They show when the monitor computes precise knowledge but also limitations due to the static analysis of loops. They also illustrate the role played by the static detection of termination.

$$(\textsc{skip})\ \frac{}{(\mathsf{skip}, s) \Downarrow s} \qquad (\textsc{seq})\ \frac{(c_1, s) \Downarrow s' \qquad (c_2, s') \Downarrow s''}{(c_1; c_2, s) \Downarrow s''}$$

$$(\textsc{assignDyn})\ \frac{[\![e]\!]_\rho = v \quad (\!|e|\!)_\kappa = e^\sharp}{(x := e, (\rho, \kappa)) \Downarrow (\rho[x \mapsto v], \kappa[x \mapsto e^\sharp])} \qquad (\textsc{assignStat})\ \frac{(\!|e|\!)_\kappa = e^\sharp}{(x := e, (\cdot, \kappa)) \Downarrow (\cdot, \kappa[x \mapsto e^\sharp])}$$

$$(\textsc{assumeDyn})\ \frac{C[\![e]\!]_\rho = tt}{(\mathsf{assume}(e), (\rho, \kappa)) \Downarrow (\rho, \mathbb{A}((\!|e|\!)_\kappa, \kappa))} \quad (\textsc{assumeStat})\ \frac{}{(\mathsf{assume}(e), (\cdot, \kappa)) \Downarrow (\cdot, \mathbb{A}((\!|e|\!)_\kappa, \kappa))}$$

$$(\textsc{ifDyn})\ \frac{C[\![e]\!]_\rho = \alpha \qquad (c_\alpha, (\rho, \kappa)) \Downarrow (\rho', \kappa_\alpha) \qquad (c_{\bar{\alpha}}, (\cdot, \kappa)) \Downarrow (\cdot, \kappa_{\bar{\alpha}})}{(\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}, (\rho, \kappa)) \Downarrow (\rho', \mathbb{IF}((\!|e|\!)_\kappa, \kappa_{tt}, \kappa_{ff}))}$$

$$(\textsc{ifStat})\ \frac{(c_{tt}, (\cdot, \kappa)) \Downarrow (\cdot, \kappa_{tt}) \qquad (c_{ff}, (\cdot, \kappa)) \Downarrow (\cdot, \kappa_{ff})}{(\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}, (\cdot, \kappa)) \Downarrow (\cdot, \mathbb{IF}((\!|e|\!)_\kappa, \kappa_{tt}, \kappa_{ff}))}$$

$$(\textsc{whileDynTrue})\ \frac{C[\![e]\!]_\rho = tt \qquad (\mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}, s) \Downarrow s' \qquad s = (\rho, \kappa)}{(\mathsf{while}\ e\ \mathsf{do}\ c, s) \Downarrow s'}$$

$$(\textsc{whileDynFalse})\ \frac{C[\![e]\!]_\rho = ff \qquad (\mathsf{if}\ e\ \mathsf{then}\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}, s) \Downarrow s' \qquad s = (\rho, \kappa)}{(\mathsf{while}\ e\ \mathsf{do}\ c, s) \Downarrow s'}$$

$$(\textsc{whileStat})\ \frac{(\mathsf{assume}(e); c, s') \Downarrow s_1 \qquad s_1 \preccurlyeq s' \qquad s \preccurlyeq s' \qquad (\mathsf{assume}(\neg e), s') \Downarrow s'' \qquad s = (\cdot, \kappa)}{(\mathsf{while}\ e\ \mathsf{do}\ c, s) \Downarrow s''}$$

where $(\!|x|\!)_\kappa = \kappa(x) \quad (\!|n|\!)_\kappa = \lambda\rho.\lfloor n \rfloor \quad (\!|e_1 \oplus e_2|\!)_\kappa = \lambda\rho.(\!|e_1|\!)_\kappa(\rho) \oplus^\sharp (\!|e_2|\!)_\kappa(\rho)$

$$\mathbb{IF}(c, \kappa_1, \kappa_2)(x) = IF(c, \kappa_1(x), \kappa_2(x)) \qquad \mathbb{A}(c, \kappa)(x) = Assume(c, \kappa(x))$$

$$(\rho, \kappa) \preccurlyeq (\rho', \kappa') \text{ iff } \kappa \preccurlyeq \kappa' \wedge \rho = \rho'$$

Fig. 3: Hybrid knowledge analysis semantics.

```
1 if h then z := x + y                    Program 5
2 else z := y - x;
3 output z
```

Example 1 (Precise knowledge computation)**.** For Program 5, the hybrid monitor computes $\kappa(z) = \lambda\rho.if(C[\![h]\!]_\rho, [\![x+y]\!]_\rho, [\![y-x]\!]_\rho)$. The program is loop-free and therefore $\kappa(z)$ is a function which encodes exactly the function computing the final value of z from the initial environment. Suppose that the final value of z is 1, the knowledge of the output 1 is obtained by $\kappa(z)^{-1}(1) = \{\rho \mid if(C[\![h]\!]_\rho, [\![x+y]\!]_\rho, [\![y-x]\!]_\rho) = 1\}$.

Suppose that initially $h = true$, $x = 0$ and $y = 1$. As a result, L-equivalent environments are $\{\rho \mid \rho(x) = 0 \wedge \rho(y) = 1\}$[3]. This program is indeed secure for the given initial environment (see Definition 3) since all L-equivalent environments output the value 1. Notice that all of them are included in $\kappa(z)^{-1}(1)$: if $x = 0$ and $y = 1$ then the condition $if(h, x+y, y-x) = 1$ always holds: $if(h, 0+1, 1-0) = 1 \Leftrightarrow if(h, 1, 1) = 1 \Leftrightarrow 1 = 1$.

[3]As z is set in both branches, its initial value is irrelevant.

Notice that both dynamic monitors and the standard hybrid monitors block all executions of this program either because there is a low assignment under a high security context in both branches, or because an output variable z explicitly depends on h. We will show in Section VI-C that this power of proving non-interference allows our monitor to be more permissive than other monitors.

Example 2 (Detection of loop non-termination)**.** Interestingly, our monitor may be more precise than other monitors even in the presence of loops in a high security context. Consider Program 6.

```
1 l := 0;                                 Program 6
2 if h then skip
3 else while true do l := 1;
4 output l
```

When h is $true$, the purely dynamic monitors would accept this execution, while the previous hybrid monitors [7], [18] would block it since the hybrid monitor would detect that a value of l might change in the non-executed branch.

However, our monitor is able to detect the nontermination of the while loop. On line 2, we apply the IfDyn

rule, and compute $\kappa(l) = \lambda\rho.if(C[\![h]\!]_\rho, \kappa_0(l), \kappa''(l))$, where $\kappa_0(l) = \lambda\rho.0$ and κ'' is computed by the WhileStat rule. The first three premises of this rule ensure that in state $s' = (\bot, \kappa')$, we have $\kappa'(l) = \lambda\rho.\top$ since l is updated in the loop body. However, the forth premise $((\mathsf{assume}(\neg e), s') \Downarrow s'')$ ensures that $\kappa''(l) = \lambda\rho.\bot$ thus being able to conclude that whenever h is $false$, the program does not terminate.

Example 2 shows that our static analysis allows us to detect non-termination of the loops in some cases. Notice that this capability does not give us soundness for termination-sensitive noninterference, but gives more precision for termination-insensitive noninterference. In contrast, the knowledge monitor in our previous work [7] is not able to prove termination-insensitive noninterference for Example 2 since its static analysis only determines that the output `l` may depend on the secret `h`.

C. Limitations

Our hybrid monitor is not always capable of computing the exact knowledge. A fundamental reason is that the knowledge is computed for each variable independently. Therefore, it cannot express a relation, e.g., the equality of variables.

Example 3 (Imprecise knowledge computation)**.** For Program 7 and its execution when $h = true$, our static analysis does not infer that, at the end of the loop y is equal to x. Hence, it fails at deducing that in the other branch y is 0.

Program 7

```
1 y = 1; x = N;
2 if h then skip;
3 else while x > 0 do x = x-1; y = x;
4 output y
```

When h is true, we statically analyse the while-loop. The loop invariant is $s' = (\cdot, \kappa')$, where κ' defines a knowledge for each variable. To model the fact that x is decremented at each iteration, the static analysis computes

$$\kappa'(x) = \bigvee_{0 \le n \le N} \lambda\rho.n = \lambda\rho.\top.$$

This information is propagated towards y by the assignment $y = x$ and we get $\kappa'(y) = \lambda\rho.\top$. At the end of the loop, the test $\neg x > 0$ *i.e.,* $x = 0$ – providing x is a natural integer – allows to recover the fact that x is necessarily 0. However, as the equality between x and y is not propagated, the value of y cannot be recovered. As a result, the final knowledge in y is $\kappa(y) = \lambda\rho.if(C[\![h]\!]_\rho, 1, \top)$, while the real attacker knowledge is $\lambda\rho.if(C[\![h]\!]_\rho, 1, 0)$.

As a result of the imprecise knowledge computation, the static analysis is not always capable to detect the loop termination. If we have $\kappa(x)(\rho) = \bot$, we know for certain that the program does not terminate for initial environment ρ. However, if $\kappa(x)(\rho) = v$ for some v, there is no certainty. The program either terminates and the value is indeed v or the program does not terminate. Said otherwise, any non-terminating execution from initial environment ρ can soundly approximated by $\kappa(x)(\rho) = v$.

Example 4 (Non-detection of loop non-termination)**.** Consider the program obtained as the sequential composition of Program 7 followed by Program 8. This new composed program is noninterferent (TINI) since it either outputs 1 or does not terminate.

Program 8

```
1 x = 1;
2 while y = 0 do x = 1;
3 output x
```

After an execution of the Program 7 in the initial environment where $h = true$, the computed knowledge in variable y is $\kappa(y) = \lambda\rho.if(C[\![h]\!]_\rho, 1, \top)$.

The static analysis of the while loop detects that the value of x in the loop body is always 1 and the knowledge in variable x is $\kappa(x) = \lambda\rho.1$. However, since the knowledge in y is not precise, the static analysis is unable to determine the non-termination of the loop.

V. Correctness and Soundness

Given a monitor's knowledge κ, a variable x and its value v, we can express the set of possible environments that can produce v as the inverse of κ: $\kappa(x)^{-1}(v)$. The Monitor Correctness Theorem 1 states that this set of environments intersected with the low-equivalence class $[\rho_i]_\mathsf{L}$ is a correct approximation of the attacker knowledge for environment ρ_i, as defined in Definition 1.

Theorem 1 (Monitor Correctness)**.** *Let $c \in \mathbb{C}, \rho, \rho' \in Env, \kappa \in Var \to \mathbf{K}$ and assume that $(c, init(\rho)) \Downarrow (\rho', \kappa)$. Then for all $v \in \mathbb{V}$, $x \in Var$ and $\rho_i \in Env$,*

$$\kappa(x)^{-1}(v) \cap [\rho_i]_\mathsf{L} \subseteq \mathcal{K}(c; \mathsf{output}\ x, v, \rho_i).$$

Proof sketch. The Correctness Theorem is a consequence of a more general, inductive invariant. It states that if we are given sound knowledge κ about a program c_0 executed in environment ρ then monitoring another program c with κ as initial knowledge will produce final knowledge κ' which is sound for the sequential composition $c_0; c$ when executed in environment ρ. To state this invariant, we define the predicate *soundK* which states that the knowledge $\kappa \in Var \to \mathbf{K}$ is sound for an execution of program c in initial environment ρ_i. Formally, we write $soundK(c, \rho_i, \kappa)$ if for all $v \in \mathbb{V}$, $x \in Var$ and $\rho_f \in Env$,

$$\left.\begin{array}{c} \rho_i \in \kappa(x)^{-1}(v) \\ \wedge \\ (c, \rho_i) \downarrow \rho_f \end{array}\right\} \Rightarrow \rho_f(x) = v.$$

The invariant can then be stated as follows:

$$\left.\begin{array}{c} (c, (\rho_0, \kappa)) \Downarrow (\rho_1, \kappa') \\ \wedge \\ soundK(c_0, \rho, \kappa) \end{array}\right\} \Rightarrow soundK(c_0; c, \rho, \kappa').$$

The proof of this invariant is by structural induction over the relation $\Downarrow$ and by case analysis over the hybrid

monitoring rules of Figure 3. Instantiating this invariant with $c_0 = \mathsf{skip}$ and $\kappa = \kappa_0$, we get that

$$(c, \text{init}(\rho)) \Downarrow (\rho', \kappa) \Rightarrow \forall \rho_i. soundK(c, \rho_i, \kappa).$$

Theorem 1 then follows from the observation that all knowledge that is sound according to the predicate *soundK* is a *subset* of the attacker knowledge, as defined in Definition 1. Formally, if $soundK(c, \rho, \kappa))$ then

$$\kappa(x)^{-1}(v) \cap [\rho]_{\mathsf{L}} \subseteq \mathcal{K}(c; \mathsf{output}\ x, v, \rho). \qquad \square$$

We complete the semantics of a hybrid monitor with an additional rule to deal with outputs, presented below. The output rule uses the $NI(\rho, K, v)$ predicate that uses the knowledge K to check whether all low-equal initial environments would either produce the same value v or would not terminate (indicated by a value $\perp$). In Section VII-D we describe how to efficiently implement the computation of predicate *NI*. This predicate is defined by

$$NI(\rho, K, v) \quad \triangleq \quad [\rho]_{\mathsf{L}} \subseteq K^{-1}(v) \cup K^{-1}(\perp).$$

The rule for output is then given by:

$$(\textsc{outNI})\ \frac{(c, init(\rho)) \Downarrow (\rho', \kappa) \quad \rho'(x) = v \quad NI(\rho, \kappa(x), v)}{(c; \mathsf{output}\ x, \rho) \Downarrow v}$$

With this output rule, we have the property of a knowledge-based hybrid monitor that the monitor either accepts the output of a program, or blocks.

Lemma 2. *If a knowledge-based hybrid monitor produces a value v for a program P from an initial environment ρ, then the original program P computes the same value:*

$$(P, init(\rho)) \Downarrow v \Rightarrow (P, \rho) \downarrow v.$$

We can then prove (using Theorem 1) that the knowledge-based hybrid monitor completed with an output rule outNI enforces knowledge-based security.

Theorem 2 (Monitor Soundness)**.** *A program P, monitored by a knowledge-based hybrid monitor with output rule* outNI*, is TINI under the monitor semantics $\Downarrow$: whenever $\rho^1_{\mathsf{L}} = \rho^2_{\mathsf{L}}$ and $(P, init(\rho^1)) \Downarrow v^1$ and $(P, init(\rho^2)) \Downarrow v^2$, then $v^1 = v^2$.*

Example 5 (Permissiveness)**.** Program 6 of Example 2 demonstrates when our analysis is able to detect non-termination of the program. Our monitor computes the knowledge in variable l as $\kappa(l) = \lambda\rho.if(C[\![h]\!]_\rho, 0, \perp)$. Then, when a variable l is to be output, the outNI rule ensures that on all possible low-equal environments, either the program outputs 0 or does not terminate – the predicate *NI* holds.

Given that our monitor is sometimes able to model when the program does not terminate, it might be tempting to enforce termination-sensitive noninterference (TSNI). To achieve it, one could substitute the *NI* predicate with a *TSNI* predicate requiring that in all the low-equal memories either the program terminates producing v, or it does not terminate:

$$TSNI(\rho, K, v) \triangleq [\rho]_{\mathsf{L}} \subseteq K^{-1}(v) \vee [\rho]_{\mathsf{L}} \subseteq K^{-1}(\perp).$$

The problem with this approach is that whenever $\kappa(x)(\rho) = v$, there is no certainty that the program terminates, since v approximates $\perp$.

Example 6 (TSNI counterexample)**.** The composed program from Example 4 is TINI but not TSNI since it terminates on $h = true$ and does not terminate when $h = false$. The hybrid monitor computes the knowledge in x as $\kappa(x) = \lambda\rho.1$ that would satisfy *TSNI* predicate, however this would not be a sound enforcement of TSNI.

VI. Combination with other monitors

We now show how an existing dynamic monitor based on security levels can be combined with our knowledge-based hybrid monitor. The combined monitor will admit more executions than each of the monitors taken in separation, and will still be secure. To compare the precision of monitors, Hedin et al. [12] propose the notion of "permissiveness" that compares a set of program executions accepted by two monitors and defines a monitor to be more permissive if it accepts a strictly bigger set of executions.

Hedin et al. [12] observe that purely dynamic monitors (e.g., NSU [4]) and simple hybrid monitors (e.g., Le Guernic et al. [18]) are not necessarily comparable with respect to their permissiveness. For example, the execution of Program 1 in environment $h = false$ is accepted by a dynamic monitor NSU because the test is false, but it is rejected by a hybrid monitor since the static analysis concludes that there might be a leak on the non-executed branch. On the other hand, NSU rejects an execution of Program 2 when $h = true$ (because of a sensitive upgrade on line 1), while a hybrid monitor accepts it (because the security level of l is downgraded to L on line 2).

A. Hybrid monitor reusing an inlined monitor

We assume that a monitor based on security levels (for example, a purely dynamic monitor), is inlined in the program following the inlining technique simultaneously proposed by Chudnov and Naumann [10] and Russo and Sabelfeld [20]. Here, a program c is transformed into a program $\widetilde{c}$, where each variable x has a shadow variable $\widetilde{x}$ representing the security label of x. The monitoring is not intrusive in the sense that the values of x are the same for c and $\widetilde{c}$. In other words, the computation of security levels has no impact on the computed values. We will present the instantiation to NSU in Section VI-B below.

Given a program $P = c; \mathsf{output}\ x$, the monitor usually decides to output x if the label of x is lower or equal than the level L[4]. A hybrid monitor can choose to mimic the

[4]Usually, this condition is $pc \sqcap \Gamma(x) \sqsubseteq \mathsf{L}$, however the program counter *pc* is at the lowest level L because our programs only produce output outside conditionals and while-loops.

$$(\text{T-SKIP})\ G, S \vdash \mathsf{skip} \rhd \mathsf{skip} \qquad (\text{T-SEQ})\ \frac{\forall i = 1..2. G, S \vdash c_i \rhd \widetilde{c}_i}{G, S \vdash c_1; c_2 \rhd \widetilde{c}_1; \widetilde{c}_2}$$

$$(\text{T-IF})\ \frac{S(x) = \widetilde{x} \qquad y \notin dom(S) \cup rng(S) \wedge y \notin G \qquad \forall i = 1..2. (y :: (pc :: G)), S \vdash c_i \rhd \widetilde{c}_i}{(pc :: G), S \vdash \mathsf{if}\ x\ \mathsf{then}\ c_1 \mathsf{else}\ c_2 \rhd y := \widetilde{x} \sqcup pc; \mathsf{if}\ x\ \mathsf{then}\ \widetilde{c}_1 \mathsf{else}\ \widetilde{c}_2}$$

$$(\text{T-WHILE})\ \frac{S(x) = \widetilde{x} \qquad y \notin dom(S) \cup rng(S) \wedge y \notin G \qquad (y :: (pc :: G)), S \vdash c \rhd \widetilde{c}}{(pc :: G), S \vdash \mathsf{while}\ x\ \mathsf{do}\ c \rhd y := \widetilde{x} \sqcup pc; \mathsf{while}\ x\ \mathsf{do}\ (\widetilde{c}; y := \widetilde{x} \sqcup y)}$$

$$(\text{T-ASSIGN})\ \frac{S \vdash e \rhd \widetilde{e} \qquad S(x) = \widetilde{x} \qquad \widetilde{c} = \mathsf{if}\ pc \sqsubseteq \widetilde{x}\ \mathsf{then}\ \widetilde{x} = \widetilde{e} \sqcup pc\ \mathsf{else}\ \forall y \in Var. S(y) = B}{pc :: G, S \vdash x := e \rhd \widetilde{c}; x := e}$$

Fig. 4: NSU inlining transformation

original behaviour of the inlined monitor by introducing the following output rule OUTL:

$$(\text{OUTL})\ \frac{(\widetilde{c}, init(\rho)) \Downarrow (\rho', \kappa) \quad \rho'(x) = v \quad \rho'(\widetilde{x}) \sqsubseteq \mathsf{L}}{(\widetilde{c}; \mathsf{output}\ x), \rho \Downarrow v}$$

This rule ensures that the inlined monitor only outputs a value v when the security level of the variable x is below L. We make one assumption about the inlined monitor *viz.*, that it correctly computes the security labels of the variables that can be used to enforce noninterference.

Assumption 1 (Correct labels of inlined monitor)**.** Consider a program $P = \widetilde{c}; \mathsf{output}\ x$. If it outputs a value v according to the following output rule:

$$(\text{OUTL})\ \frac{(\widetilde{c}, \rho) \downarrow \rho' \quad \rho'(x) = v \quad \rho'(\widetilde{x}) \sqsubseteq \mathsf{L}}{(\widetilde{c}; \mathsf{output}\ x), \rho \downarrow v}$$

Then the label of variable x is computed correctly in the final environment ρ', meaning

$$\mathcal{K}(P, v, \rho) = \mathcal{I}(P, \rho).$$

As a consequence, a hybrid monitor only using the OUTL output rule soundly enforces noninterference.

To gain more precision, the hybrid monitor can first use its own knowledge about the output variable x (applying the rule OUTNI), and only if it is unable to prove noninterferfence, it can then try to apply the default output rule OUTL of the inlined monitor.

We can gain even more precision by reasoning about the knowledge contained in the shadow variables. This will allow the monitor to output certain values, even if neither the rule OUTNI nor the rule OUTL holds. The idea is to provide the knowledge-based monitor with the extra information that certain variables would never be output because the dynamic monitor would block them. Environments that lead to an output that is certain to be blocked by the dynamic monitor can be disregarded when computing the set of possible outcome of the program. Concretely, we add a new security level B ("will be blocked by the monitor"), such that $H \sqsubset B$. An additional output rule OUTH exploits this new label. It requires that:

- the value of x can be output (because $\rho(\widetilde{x}) \sqsubset B$), and
- all the environments that will not be blocked by a monitor, written $NotB(\widetilde{K})$, and low-equal to ρ would produce the same value v.

Formally, we get the following output rule OUTH that combines information computed by the hybrid monitor and the inlined dynamic monitor.

$$(\text{OUTH})\ \frac{\begin{array}{c}(\widetilde{c}, init(\rho)) \Downarrow (\rho', \kappa) \quad \rho'(x) = \lfloor v \rfloor \\ \rho'(\widetilde{x}) \not\sqsubseteq \mathsf{L} \quad \rho'(\widetilde{x}) \sqsubset B \quad single(\rho, \kappa(x), \kappa(\widetilde{x}), v)\end{array}}{(\widetilde{c}; \mathsf{output}\ x), \rho \Downarrow v}$$

The predicate *single* ensures that we only produce a single unique value. It exploits the new *blocked* level B and is defined as follows.

$$\begin{array}{lcl} single(\rho, K, \widetilde{K}, v) & \triangleq & ([\rho]_{\mathsf{L}} \cap NotB(\widetilde{K})) \subseteq K^{-1}(v) \\ NotB(\widetilde{K}) & \triangleq & (Env \backslash \widetilde{K}^{-1}(B)). \end{array}$$

With this rule, the combination of the hybrid monitor and a dynamic monitor can be strictly more permissive than either of them.

B. Application to NSU

In the No-Sensitive Upgrade monitor (NSU) [4], [25] an assignment to a variable is allowed only if the program counter level *pc* is lower than the level of the assigned variable l. Otherwise, the execution is stopped.

Figure 4 presents an inlining transformation for NSU. The inlining technique uses the transformation judgement $G, S \vdash c \rhd \widetilde{c}$, where $S : vars(c) \to Var$ maps each variable of c into its *shadow variable* $S(x)$ which contains the current security level of x; and G is a finite list of variables that store the current *pc*. $S \vdash e \rhd \widetilde{e}$ means that $\widetilde{e}$ is an expression for the level join of shadow variables of the variables in e (see [10, Fig. 9]).

NSU enforces termination-insensitive noninterference [4], so the proposed transformation indeed satisfies Assumption 1. The only difference in our presentation

of inlining is that we do not write an explicit divergence operation (such as `while (true) skip`), but rather assign the highest security level B to all the variables (see T-ASSIGN rule); and still allow the computation $x := e$.

The following short example demonstrates in a concise manner how the knowledge-based hybrid monitor that reuses NSU is more permissive than NSU.

Program 9

```
1 l = 1; if h then l = 0;
2 output h
```

Example 7. All the executions of Program 9 are blocked by NSU since the program tries to output a high variable h on a low channel. However our knowledge-based hybrid monitor combined with NSU accepts its execution when $h = \mathit{false}$. Following the idea of NSU, the monitor transforms an information channel into a termination channel. When $h = \mathit{false}$, the hybrid monitor is able to compute that the other initial environment ρ', where $h = \mathit{true}$ will get a B label ($\kappa(\widetilde{h})(\rho') = B$) and therefore will not be output (unless the rule OUTNI holds). Therefore, the output is accepted by the OUTH rule because the *single* predicate ensures that all the executions not blocked by the monitor (where $h = \mathit{false}$) will compute the same value.

C. Soundness and Permissiveness

We now prove the main soundness result for the hybrid monitor that executes a program with an inlined security monitor. In the following, we write $HM(\textsc{Output})$ for our hybrid monitor using a *set* of output rules $\textsc{Output} \subseteq \{\textsc{outNI}, \textsc{outL}, \textsc{outH}\}$.

Theorem 3. *All the executions of a program $P = (\widetilde{c}; \mathsf{output}\ x)$, monitored by a knowledge-based hybrid monitor $HM(\{\textsc{outNI}, \textsc{outL}, \textsc{outH}\})$ are secure for all environments ρ.*

Remark that a hybrid monitor using a smaller set of output rules is also sound. This fact will be useful to prove the relative precision of hybrid monitors.

A monitor A is more permissive than a monitor B if it stops less monitored executions (suppresses less outputs). Following Hedin *et al.* [12], a *productive* environment for a monitor M and a program P, written $E_M(P)$ is a set of environments for which the monitor does not stop, i.e., $E_M(P) = \{\rho \mid \exists v.(P, \rho) \Downarrow_M v\}$. Thus, E_M is a family of productive sets indexed by programs for the monitor M.

Definition 6. A monitor A is *at least as permissive* as a monitor B if $E_B \subseteq E_A$.

Theorem 4 (Hierarchy of hybrid monitors)**.** *Consider a program $P = c; \mathsf{output}\ x$ and a program $\widetilde{P} = \widetilde{c}; \mathsf{output}\ x$ that is instrumented by a sound dynamic monitor for noninterference. The following precision results hold:*

$$
\begin{array}{lcl}
E_{HM(\{\textsc{outNI}\})}(P) & = & E_{HM(\{\textsc{outNI}\})}(\widetilde{P}) \\
E_{HM(\{\textsc{outNI}\})}(\widetilde{P}) & \subseteq & E_{HM(\{\textsc{outNI},\textsc{outL}\})}(\widetilde{P}) \\
E_{HM(\{\textsc{outL}\})}(\widetilde{P}) & \subseteq & E_{HM(\{\textsc{outNI},\textsc{outL}\})}(\widetilde{P}) \\
E_{HM(\{\textsc{outNI},\textsc{outL}\})}(\widetilde{P}) & \subseteq & E_{HM(\{\textsc{outNI},\textsc{outL},\textsc{outH}\})}(\widetilde{P})
\end{array}
$$

Theorem 4 shows that the combination of our hybrid monitor with a dynamic monitor is more precise than each of them taken separately. Moreover, as the output rule OUTH requires a cooperation between both, our best hybrid monitor $HM(\{\textsc{outNI}, \textsc{outL}, \textsc{outH}\})$ is more precise than a direct parallel composition of both, which can be obtained as $HM(\{\textsc{outNI}, \textsc{outL}\})$.

Example 8. Consider again a Program 9. When $h = \mathit{false}$, the knowledge-based hybrid monitor ($E_{HM(\{\textsc{outNI}\})}$) is not able to prove noninterference and the level of h is H, and hence the output is blocked. However, the more precise monitor $E_{HM(\{\textsc{outNI}, \textsc{outL}, \textsc{outH}\})}$ ensures that h is output since the *single* predicate ensures that the monitor always produces the same output.

VII. Implementation

The knowledge-based hybrid monitor has been implemented and proved correct [1] using the Coq proof assistant. Here, we describe the main data structures and algorithms used for computing knowledge.

Knowledge about initial environments is formalized using knowledge functions $K \in \mathbf{K}$. One algorithmic challenge is to find an efficient algorithm for extracting the knowledge from such a function K *i.e.* computing $K^{-1}(v)$ for some v. We present a concrete representation for a class of knowledge functions that is closed under all the operations needed by the hybrid monitor. With this representation, a logical formula representing the knowledge of an output v is computed in linear time.

A. Concrete domain of knowledge functions

The domain of the monitor $\mathbf{K} = \mathit{Env} \to \mathbb{V}^\sharp$ is encoded with the symbolic domain $\mathbf{K}^\flat \subset \mathcal{P}(\mathbb{F} \times \mathbb{E}) \times \mathbb{F}$ where $\mathbb{E}$ is the set of program expressions and $\mathbb{F}$ is the following set of boolean formulae: $\mathbb{F} \ni \phi ::= \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid e \mid \mathit{tt} \mid \mathit{ff}$ (here $e \in \mathbb{E}$ is a program expression seen as a boolean, see Section III-A).

A pair $(f, e) \in \mathbb{F} \times \mathbb{E}$ denotes a knowledge which returns the value of the expression e when the formula f holds in the initial environment and $\top$ otherwise. The last element $\phi \in \mathbb{F}$ of $\mathbf{K}^\flat$ specifies when the knowledge is $\bot$. Given an element $K = (S, \phi) \in \mathbf{K}^\flat$, we write K^S for the set of pairs S and K^ϕ for the formula ϕ. The denotation of $K \in \mathbf{K}^\flat$ in the knowledge domain $\mathbf{K}$ is then obtained by:

$$
\begin{array}{rcl}
\{\!|K|\!\} & = & \left(\bigwedge_{(f,e)\in K^S} f \mapsto e\right) \curlywedge K^\phi \mapsto \bot \\
\text{where } \psi \mapsto e & = & \lambda\rho.\text{if } [\![\psi]\!]_\rho \text{ then } [\![e]\!]_\rho \text{ else } \top
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{true}[K] & = & \bigvee_{(f,e)\in K^S}(f \wedge e)\\
\mathit{false}[K] & = & \bigvee_{(f,e)\in K^S}(f \wedge \bar{e})\\
\mathit{top}[K] & = & \neg(\mathit{true}[K] \vee \mathit{false}[K] \vee K^\phi)\\
K@\psi & = & (\{(\psi \wedge f, e) \mid (f,e) \in K^S\}, K^\phi \vee \neg\psi)\\
K_1 \curlyvee K_2 & = & (K^= \cup K_{12} \cup K_{21}, K_1^\phi \wedge K_2^\phi), \text{ where}\\
K^= & = & \left\{(f_1 \wedge f_2 \wedge e_1 = e_2, e_1) \middle| \begin{array}{l}(f_1,e_1)\in K_1^S\\(f_2,e_2)\in K_2^S\end{array}\right\}\\
K_{ij} & = & \{(f_i \wedge K_j^\phi, e_i) \mid (f_i,e_i) \in K_i^S\}
\end{array}
$$

Fig. 5: Implementation of the conditional operator.

Our domain $\mathbf{K}^\flat$ also requires well-formedness conditions. Given $(S,\phi) \in \mathbf{K}^\flat$, we add the constraint that any two pairs (f,e) and (f',e') from S must satisfy that if both f and f' hold then the expressions e and e' evaluate to the same value. Moreover, for any $(f,e) \in S$, the conjunction of f and ϕ does not hold. All the operators needed by the hybrid monitor preserve this property.

Example 9. We illustrate below the symbolic encoding of basic knowledge functions.

$$
\{\!|(\emptyset, \mathit{ff})|\!\} = \lambda\rho.\top \qquad \{\!|(\emptyset, \mathit{tt})|\!\} = \lambda\rho.\bot
$$
$$
\{\!|\{(\mathit{true}, n)\}, \mathit{ff})|\!\} = \lambda\rho.n
$$

For Program 3, we get the following knowledge in l in the end of the program:

$$
(\{(h,1);(\neg h, x);(x=1,x)\}, \mathit{ff})
$$

The knowledge is well-formed since if any two formulae among $h, \neg h$ or $x = 1$ hold at the same time, the corresponding expressions evaluate to the same value.

B. Implementation of operators

The assignment rules ASSINDYN and ASSIGNSTAT compute the knowledge of an expression using a function $(\!|\_|\!)_\kappa$. We show how to implement this function for a new domain of knowledge $\mathbf{K}^\flat$ below:

$$
\begin{array}{rcl}
(\!|x|\!)_\kappa = \kappa(x) \qquad (\!|n|\!)_\kappa = (\{(\mathit{tt}, n)\}, \mathit{ff})\\
(\!|k_1 \oplus k_2|\!)_\kappa & = & (S, (\!|k_1|\!)_\kappa^\phi \vee (\!|k_2|\!)_\kappa^\phi) \text{ where}\\
S & = & \left\{(f_1 \wedge f_2, e_1 \oplus e_2) \middle| \begin{array}{l}(f_1,e_1)\in (\!|k_1|\!)_\kappa^S\\(f_2,e_2)\in (\!|k_2|\!)_\kappa^S\end{array}\right\}
\end{array}
$$

One key operator is the conditional operator *IF* which combines the knowledge from different execution paths. The *IF* and *assume* operators can be rewritten using more basic operators, defined in Figure 5:

$$
\begin{array}{lcl}
\mathit{Assume}(c,l) & = & l@\mathit{true}[c] \curlyvee l@\mathit{top}[c]\\
\mathit{IF}(c,l,r) & = & l@\mathit{true}[c] \curlyvee r@\mathit{false}[c] \curlyvee (l \curlyvee r)@\mathit{top}[c]
\end{array}
$$

Theorem 5. *The operators over $\mathbf{K}^\flat$ exactly model the operators over $\mathbf{K}$. In particular, we have*

$$
\begin{array}{lcl}
\{\!|(\!|e|\!)_\kappa|\!\} & = & (\!|e|\!)_{\lambda x.\{\!|\kappa(x)|\!\}}\\
\{\!|K_1 \curlyvee K_2|\!\} & = & \{\!|K_1|\!\} \curlyvee \{\!|K_2|\!\}\\
\{\!|\mathit{IF}(c,e_1,e_2)|\!\} & = & \mathit{IF}(\{\!|c|\!\}, \{\!|e_1|\!\}, \{\!|e_2|\!\})
\end{array}
$$

C. Static analysis of loops

The only place where the specification of the hybrid monitor is not directly executable is the rule WHILESTAT. The implementation of this rule requires an iterative fixpoint computation in order to infer an invariant of the loop body. The domain $\mathbf{K}^\flat$ does not satisfy the finite ascending chain condition. Therefore, a widening operator is needed to ensure convergence and speed up computations. Our widening limits the number of distinct expressions which can occur in formulae. To remove an expression e from a formula f, we compute the formula $f^+ \wedge f^-$ where f^+ is obtained by substituting e for tt and f^- is obtained by substituting e for ff. The obtained formula is by construction stronger which ensures the soundness of the transformation. Remember that $(f \mapsto e)$ returns $\top$ when f does not hold. By bounding the number of expressions to some fixed constant k, and because formulae have a normal, our fixpoint iteration operates over a finite domain of boolean formulae. This ensures convergence.

D. Effective proof of non-interference

To get an effective enforcement and implement the rules OUTNI and OUTH, we need a decision procedure for the predicates *NI* and *single*. These predicates can be encoded as logic formulae $f \in \mathbb{F}$. To get a decidable enforcement, the logic needs to be decidable. As propositional logic is decidable, the decidability depends only on the language of expressions $\mathbb{E}$. It will hold, for instance, for decidable fragment of arithmetic such as linear integer arithmetics or bit-vector arithmetics.

The logic translation is syntax-directed and defined by the function $\langle\cdot\rangle$:

$$
\begin{array}{lcll}
\langle K^{\text{-}1}(v)\rangle & = & \bigvee_{(f,e)\in K^S} f \wedge e = v & \langle K^{\text{-}1}(\bot)\rangle = K^\phi\\
\langle [\rho]_\mathsf{L}\rangle & = & \bigwedge_{\{x \mid \Gamma(x) \sqsubseteq \mathsf{L}\}} x = \rho(x) &
\end{array}
$$

The inverse function $K^{\text{-}1}(v)$ represents the knowledge of the output v. Given the output value v and a symbolic knowledge $K \in \mathbf{K}^\flat$, $\langle K^{\text{-}1}(v)\rangle$ builds a disjunction where all the expressions are constrained to be equal to v. The equivalence class $[\rho]_\mathsf{L}$ of a initial environment ρ is obtained by a conjunction of equality constraints stating that a low variable x, should have the value of $\rho(x)$. Set operations $\cup$, $\cap$ and $\setminus$ have a standard encoding and set inclusion can be done by checking entailment. Theorem 6 states that the security predicates *NI* and *single* can be checked by checking that their logic encoding is a tautology.

Theorem 6. *The logic translation of security predicates is sound and complete.*

$$
\begin{array}{lll}
\mathit{NI}(\rho, K, v) & \mathit{iff} & \langle [\rho]_\mathsf{L}\rangle \Rightarrow \langle K^{\text{-}1}(v)\rangle \vee \langle K^{\text{-}1}(\bot)\rangle\\
\mathit{single}(\rho, K, \widetilde{K}, v) & \mathit{iff} & \langle [\rho]_\mathsf{L}\rangle \wedge \langle \mathit{NotB}(\widetilde{K})\rangle \Rightarrow \langle K^{\text{-}1}(v)\rangle
\end{array}
$$

E. Experiments

From our Coq development, we have extracted an Ocaml proof-of-concept implementation [1]. We have extracted

five programs from this paper and used our implementation to compute the approximated knowledge. These programs were selected to demonstrate the difference in monitors with respect to the attacker knowledge approximation and permissiveness. For these five programs, we provide the actual knowledge gained by an attacker by observing an output and the approximation computed by different monitors, including the best hybrid monitor of Besson *et al.* [7], called HM(Val+Comb). For enforcement of noninterference, we compare the permissiveness of a knowledge-based hybrid monitor HM (Section IV), the standard NSU and their combination (Section VI).

For each program, we analyse an execution for a given initial environment presented in column 2 of Figure 6a. The actual knowledge of the attacker is a formula over the high variables, that we present in column $\mathcal{K}(P_i, v, \rho)$. For columns HM(Val+Comb) and HM, we highlight in light grey the knowledge that was not computed precisely.

For the majority of programs the hybrid monitor HM is able to compute the exact knowledge of the attacker. Only for Program P_7, our monitor approximates the knowledge of the attacker due to the its static analysis limitation. For Programs P_4 and P_5, our hybrid monitor is strictly more precise than the hybrid monitor of Besson *et al.*

Figure 6b gives an insight into permissiveness of the monitors, where HM stands for $HM(\{\text{OUTNI}\})$, NSU stands for $HM(\{\text{OUTL}\})$ and HM+NSU stands for $HM(\{\text{OUTNI}, \text{OUTL}, \text{OUTH}\})$. Given a program execution described in the second column of Figure 6a, we write a ✓ if the value is output and a ✗ if the monitor blocks the execution. All the presented program executions are rejected by NSU except for P_1 and P_7. The execution of P_5 is proved noninterferent by the hybrid monitor HM, while rejected by NSU. Program P_9 illustrates the case where neither HM nor NSU alone is able to ensure termination-insensitive noninterference whereas their combination HM+NSU does. As explained in Section VI-B, the key insight is to exploit the knowledge that other interfering executions would be blocked.

VIII. Discussion

1) Scalability: The formal development and implementation of our hybrid monitor are given for a minimalistic imperative language. A relevant question is whether our core monitor could be efficiently implemented for a full-fledged language. Core dynamic monitors have already been adapted to very dynamic languages. For instance, the JSFlow project [13] implements a dynamic information flow monitor for JavaScript. Hedin *et al.* [12] also proposed a hybrid monitor that covers a large subset of JavaScript. For hybrid monitors, the main challenge is to mitigate the overhead incurred by the static analysis. Note that it is always possible to trade precision for efficiency – for instance by making an aggressive use of widening operators (see Section VII-C). At the extreme, static analysis can even be momentarily switched off if it is deemed too costly or unfeasible. In that case, the computed knowledge for the non-executed branch would be $\lambda\rho.\top$ i.e. the absence of knowledge which is sound but imprecise. Regarding functions, a reasonable trade-off could be to limit the static analysis to the current function boundaries i.e intra-procedural analysis. Yet, getting the desired trade-off between precision and efficiency requires more investigation.

2) Extension to programs with I/O and strategies: The proposed approach can be extended to programs with I/O and strategies [8]. We could define a special global variable that contains all the knowledge of the previous outputs and each new input would immediately contain that knowledge. Like this, we could track the knowledge that would be an upper bound for any possible strategy. The current representation would not change in this case.

IX. Related work

Zdancewic [25] proposed the *no-sensitive-upgrade* principle for dynamic information flow control that halts execution if a program assigns to low variables under secret control. Austin and Flanagan [4], [5] extended this to *permissive upgrade* which takes the future use of the assigned variable into account before halting the execution. *Hybrid monitors* for information flow control combine static and dynamic program analysis [17], [18], [21], [23]. One of the first techniques was proposed by Le Guernic *et al.* [18] where the static analysis only performs syntactic checks on non-executed branches. Russo and Sabelfeld [23] introduced a generic framework of hybrid monitors, where non-executed branches are analysed syntactically and formally proved that the permissiveness of such monitors is incomparable with the purely dynamic monitors. In a follow-up work, Le Guernic [17] presented a more permissive static analysis, that ignores possible branches that depend only on public variables. Besson *et al.* [7] enhance a dynamic monitor with static constant propagation and dependency analysis, and show how this leads to a hierarchy of increasingly more permissive hybrid monitors. Their knowledge is represented by the domain $\mathbb{F}\times\mathbb{V}$. As explained in Section I, the present work improves permissiveness of the hybrid monitor from [7]: 1) we have a strictly more expressive domain: an element $(f, v) \in \mathbb{F} \times \mathbb{V}$ is exactly modelled in our domain by the knowledge $(\{(f, v)\}, \mathit{ff})$; 2) we have the advantage of capturing certain forms of non-termination. With respect to dynamic monitors, permissiveness of the proposed monitor is incomparable (see Fig. 6b), however it has the power to achieve a strictly higher level of permissiveness by combination with the dynamic monitors.

Chudnov *et al.* [9] propose a hybrid monitor for relational logic. An interesting feature of the work is that the monitor is obtained from a constructive soundness proof. In this work, we consider a specific property (namely termination insensitive non-interference). Yet, our knowledge analysis is not geared to noninterference and could help discharging more general assertions of relational logic.

	$(P_i, \rho) \downarrow v$	$\mathcal{K}(P_i, v, \rho)$	HM(Val+Comb)	HM
P_1	$(l = 0, h = ff) \downarrow 1$	$\neg h$	$\neg h$	$\neg h$
P_4	$(h1 = ff, h2 = tt) \downarrow 1$	$h1 \vee h2$	$\neg h1 \wedge h2$	$h1 \vee h2$
P_5	$(h = tt, x = 0, y = 1) \downarrow 1$	tt	h	tt
P_7	$(h = tt) \downarrow 1$	tt	h	h
P_9	$(h = ff) \downarrow ff$	h	h	h

(a) Computation of Knowledge

	HM	NSU	HM + NSU
P_1	✗	✓	✓
P_4	✗	✗	✗
P_5	✓	✗	✓
P_7	✗	✓	✓
P_9	✗	✗	✓

(b) Permissiveness of Enforcement

Fig. 6: Experimental results

In a recent paper, Hedin *et al.* [12] extend a dynamic information flow monitor for core JavaScript with a static points-to analysis that can approximate the potential write targets in regions with a high security context. The dynamic monitor is based on NSU and prevents implicit flows by forbidding all side effects with targets that are below the security context. The static analysis is used to raise the security labels of the potential write targets to the level of the context before entering this context. This prevents the monitor from stopping when writing to a low target. Interestingly, the static analysis need not be sound or complete, as the dynamic monitor ensures that the hybrid monitor is sound. Precision only affects the permissiveness of the monitor. Their hybrid monitor is more precise than a static information flow analysis such as that of Hunt and Sands [15]. However, they also make the observation that "with the above definition of relative permissiveness, a hybrid monitor cannot subsume a purely dynamic monitor" [12, Thm. 3]. This is not contradictory with our findings because they only consider a particular static analysis. We conjecture that their "NSU + points-to" monitor can benefit from our extension with knowledge computation in order to obtain a monitor that subsumes their dynamic monitor.

The notion of attacker knowledge was first proposed by Askarov and Sabelfeld [3] and then used by Askarov and Chong [2] to study enforcement of noninterference when the security policy changes over time, and for different kind of attackers. The notion of knowledge here is used to state the security conditions but the enforcement mechanism does not compute the knowledge explicitly.

In the area of purely static information flow analysis, Hunt ans Sands [15] proposed a flow-sensitive type system for non-interference that was later proven to be less permissive than a standard hybrid monitor [23, Thm. 3]. Müller *et al.* [22] generalize the type system of Hunt and Sands using the technique of self-composition. They define an abstract weakest precondition calculus for self-composed program that computes logical formulae describing dependencies and equalities between variables.

X. Conclusions

We propose a hybrid monitor to compute the knowledge that an attacker obtains by observing a program output. The monitor is hybrid since it statically analyses non-executed branches. Our symbolic representation of attacker knowledge is powerful and subsumes existing hybrid monitoring approaches. We show that a knowledge-based monitor can be combined with any dynamic monitor for noninterference resulting in an enforcement mechanism that is more permissive than each mechanism taken separately. Therefore, our monitor is able to reach a level of permissiveness that was deemed impossible for the previous hybrid monitors [23].

In this paper we have laid the foundations for designing knowledge-based hybrid monitors. There are several ways in which this work can be further expanded.

The language studied here is voluntarily kept minimalistic and there are interesting semantic questions linked to how to monitor knowledge for more advanced programming languages with features such as objects, arrays and higher-order functions.

Our monitor statically analyses non-executed branches and the theory explains how this integration is designed. However, the current development can go much further and integrate traditional static analyses. In particular, more precise numeric analyses, ranging from constant propagation to polyhedral analysis, would allow the monitor to prove more equalities between variables and, hence, improve permissiveness. Other analyses such as points-to analysis would be required for the extension to the language features mentioned above.

References

[1] Formalisation of the Hybrid Monitor in Coq. Supplementary material.

[2] A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *CSF'12*, pages 308–322. IEEE, 2012.

[3] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *S&P'07*, pages 207–221. IEEE, 2007.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09*, pages 113–124, 2009.

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS'10*, pages 3:1–3:12. ACM, 2010.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *CACM*, 52:83–91, 2009.

[7] F. Besson, N. Bielova, and T. Jensen. Hybrid information flow monitoring against web tracking. In *CSF'13*, pages 240–254. IEEE, 2013.

[8] S. Chong. Required information release. *Journal of Computer Security*, 20(6):637–676, 2012.

[9] A. Chudnov, G. Kuan, and D. A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *CSF'14*, pages 48–62. IEEE, 2014.

[10] A. Chudnov and D. A. Naumann. Information Flow Monitor Inlining. In *CSF'10*, pages 200–214. IEEE, 2010.

[11] J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.

[12] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive Hybrid Information Flow Control for a JavaScript-like Language. In *CSF'15*. IEEE, 2015.

[13] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in javascript and its apis. In *SAC'14*, pages 1663–1671. ACM, 2014.

[14] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF'12*, pages 3–18. IEEE, 2012.

[15] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06*, pages 79–90. ACM, Jan. 2006.

[16] J. Landauer and T. Redmond. A lattice of information. In IEEE, editor, *CSFW'93*, pages 65–70, 1993.

[17] G. Le Guernic. Precise Dynamic Verification of Confidentiality. In *Proc. of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proc.*, pages 82–96, 2008.

[18] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *ASIAN'06*, volume 4435 of *LNCS*, pages 75–89. Springer, 2006.

[19] G. Le Guernic and T. Jensen. Monitoring Information Flow. In A. Sabelfeld, editor, *Workshop on Foundations of Computer Security - FCS'05*, Proceedings of the 2005 Workshop on Foundations of Computer Security (FCS'05), pages 19–30. DePaul University, 2005.

[20] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *SEC'10*, pages 173–186, 2010.

[21] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *CSF'11*, pages 146–160, 2011.

[22] C. Müller, M. Kovács, and H. Seidl. An analysis of Universal Information Flow based on Self-composition. In *CSF'15*. IEEE, 2015.

[23] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *CSF'10*, pages 186–199. IEEE, 2010.

[24] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

[25] S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.