2013 IEEE 26th Computer Security Foundations Symposium

Hybrid Information Flow Monitoring Against Web Tracking

Frédéric Besson, Nataliia Bielova, and Thomas Jensen
Inria
Rennes, France

***Abstract*—Motivated by the problem of stateless web tracking (fingerprinting), we propose a novel approach to hybrid information flow monitoring by tracking the knowledge about secret variables using logical formulae. This knowledge representation helps to compare and improve precision of hybrid information flow monitors. We define a generic hybrid monitor parametrised by a static analysis and derive sufficient conditions on the static analysis for soundness and relative precision of hybrid monitors. We instantiate the generic monitor with a combined static constant and dependency analysis. Several other hybrid monitors including those based on well-known hybrid techniques for information flow control are formalised as instances of our generic hybrid monitor. These monitors are organised into a hierarchy that establishes their relative precision. The whole framework is accompanied by a formalisation of the theory in the Coq proof assistant.**

I. Introduction

Web tracking refers to a collection of techniques that allow websites to create profiles of its users. While such profiles might be useful for personalized advertising, it is generally considered a problem which brings user privacy under attack. Whenever a user opens a new webpage, she has no way to know whether she is being tracked and by whom. The recent survey by Mayer and Mitchell [22] classifies the mechanisms that are used to track a user on the web. Web tracking technologies can be roughly divided into two groups: stateful and stateless. Stateful trackers store information (*e.g.*, cookies) on the user's computer. Several groups of researchers have reported on the usage of different stateful trackers on popular websites [2], [24], [31] and have found that some third-party analytics services were using these mechanisms to recreate the cookies in case they are deleted [30]. On the legal side, the European Union amendment to ePrivacy Directive 2009/136/EC was accepted and several proposals on web tracking were made [8], [16], [17], [26]. As a consequence, many web sites now explain their cookie policy but so far these regulations impose concrete restrictions only on stateful tracking technologies.

This research was partially supported by the French ANR-10-LABX-07-01 Laboratoire d'excellence CominLabs.

Stateless technologies (often called *fingerprinting*) collect information about the user's browser and OS properties, and can distinguish users by these characteristics. The calculation of the amount of identifying information is based on information theory. Eckersley demonstrated by his Panopticlick project [11] that such identification is quite effective.

A. Fingerprinting Example

For a simple illustration of fingerprinting consider the code snippet from Figure 1. A test `name = "FireFox"` schematically represents a testing of the browser name[1]. Another test `fonts = fontsSet1` schematically represents a check whether the installed fonts on the browser are the same as in some `fontsSet1`. Clearly, the information about the browser's name does not make its user uniquely identifiable. However, a precise list of fonts installed in the user's browser makes a user much more distinguishable: Eckersley [12] demonstrated in his experiments that few users share identical list of fonts.

```
1 x := 1; y := 1;
2 if (name = "FireFox") then y := 0;
3 if (y = 1) then x := 0;
4 if (fonts = fontsSet1) then
5   if (y = 1) then x := 2;
6 output x;
```

Figure 1: A possible fingerprinting code

Consider an execution of this program when `name="FireFox"` and `fonts=fontsSet1`. On line 2, `y` is assigned `0`, hence the tests `(y = 1)` fail on lines 3 and 5, and therefore this execution induces an output `x = 1`. When a tracker observes this output, she concludes that `name="FireFox"`, however she does not learn anything about the `fonts` since the code on line 5 is dead code for all executions where `name="FireFox"`. One possible protection from fingerprinting is to put a threshold on the amount of information a tracker learns about the browser features. For example, we could allow the execution described above because a tracker only learns

[1]This test corresponds to the call of `navigator.appName` browser API

DOI 10.1109/CSF.2013.23

that `name="FireFox"`, but reject the executions where a tracker learns about the installed fonts.

An efficient way to control program executions is by dynamic monitoring. Such monitors over-approximate the information leakage associated with the program output. Consider an execution described above: a dynamic monitor would only observe that lines 1, 2, 4 and 6 were executed and possibly know that the tests `(y = 1)` on lines 3 and 5 have failed. To provide a sound over-approximation of the leakage, a dynamic monitor could only conclude that an output depends on all the events in the execution trace. Hence, for this execution it would state that the tracker learns that `name ="FireFox"` and `fonts=fontsSet1`. This protection mechanism would then erroneously reject the program execution when `x = 1`, because it would conclude that the tracker also learns that `fonts=fontsSet1` whereas it is not necessarily the case: the execution where the fonts are different also result in the same output.

In this paper, we present several hybrid monitoring mechanisms that statically analyse non-executed branches of the program in order to provide a more precise approximation of the leakage. For example, the most precise of our hybrid monitors concludes that when a tracker observes that `x = 1`, she only learns that `name="FireFox"`. The protection mechanism would then allow such program execution. We present the results of our monitors for this program and show the amount of information contained in the program output in Section VII (see Program 1 of Table I).

B. Threat Model

Our web tracker extends the *gadget attacker* [4] model. Like a gadget attacker, a web tracker owns one or more web servers, where the fingerprinting scripts are located. He promotes the inclusion of these scripts into the web pages, offering a tracking or an advertisement service to websites. A web tracker does not have any special network abilities: he can only send and receive network messages from the server under his control.

Except for the gadget attacker capabilities, a web tracker has one distinctive property: he owns a database of the browser fingerprints. Therefore, a web tracker is able to compute the probability distributions for the browser properties that have been fingerprinted. These distributions could also be obtained from other sources, such as Panopticlick [11].

Another important assumption of our framework is that we shall disregard information leaks related to execution speed and termination.

Fingerprinting scripts are essentially programs, so within the program analysis realm we assume that the web tracker knows the probability distributions of the secret variables, knows the program source code and observes the output of the program.

C. Fingerprinting Protection

A straightforward counter-measure against fingerprinting is to set a threshold on the quantity of information the user agrees to leak and thus decide upon her level of anonymity. In a basic scenario, to protect from fingerprinting, we would suppress an output with a leakage above the threshold and halt the program. We discuss alternatives for protection and possible security guarantees in Section VIII-B.

Depending on the browser configuration, the same program might leak very different amounts of information. Our goal is to run a program for a browser user whose browser configuration does not incur a leakage exceeding the threshold. To achieve this goal, we need a definition of information leakage which is sensitive to the browser configuration.

D. Quantification of Information Leakage

In order to quantify identifiability of a user's browser configuration, Eckersley [12] uses the notion of *self-information* or *surprisal* from information theory. If the probability of a browser feature f to have a value v is $P(f = v)$, then the self-information is

$$I(f = v) = -\log_2 P(f = v)$$

Eckersley argues that "surprisal can be thought of as an amount of information about the identity of the object that is being fingerprinted". Consider the fingerprinting program from Figure 1. Assume for simplicity that the fonts cannot be checked. Then the only two possible outputs are `x = 0` and `x = 1`. *How much information is contained in* $\boldsymbol{x}$ *in each case?* To demonstrate self-information and discuss standard definitions for quantitative information flow (QIF), we assume that the probability of a browser name being "FireFox" is 0.21.

Self-information gives a precise answer to this question: the fact that `x = 1` (respectively, `x = 0`) means that a browser name is "FireFox" (respectively, browser name is not "FireFox"):

$$I(\texttt{x=1}) = I(\texttt{name="FireFox"}) = -\log_2 0.21 = 2.25 \text{ bits}$$
$$I(\texttt{x=0}) = I(\texttt{name}\neq\texttt{"FireFox"}) = -\log_2 0.79 = 0.34 \text{ bits}$$

This example demonstrates that the actual amount of information that the tracker learns from observing the output of a program execution can differ a lot from one execution to another.

The standard definitions of QIF (such as Shannon entropy, min-entropy, guessing entropy etc.) compute an average amount of information leakage for *all possible*

program outputs. For example, Shannon entropy based definition computes:

$$H(\texttt{name}) - H(\texttt{name}|\texttt{x}) = 0.74 \text{ bits}$$

Entropy-based definitions characterise the information flow of the program, hence providing the same (average) quantification of a program leakage for all browser users. As a consequence, even when entropy-based definitions predicts a relatively small leakage, a browser configuration of a particular user can be leaked completely to the tracker. In other words, the entropy-based approach may fail to ensure the desired privacy guarantees of a single user. We hence use self-information to quantify a leakage caused by a program execution for a concrete user.

E. Motivation for Hybrid Monitoring

Static analyses have been broadly used for QIF analysis [3], [6], [15], [21]. These static analyses approximate the leakage based on entropy-based measurements. The main challenge for applying a purely static analysis approach to fingerprinting will be to perform a precise, whole-program analysis of JavaScript programs. On the other hand, dynamic analyses have been successfully implemented in the web browsers [1], [9], [13] in order to enforce non-interference of JavaScript programs. However, purely dynamic techniques that analyse one program execution were shown to be either not precise or unsound [28] for information flow analysis.

We hence propose to investigate a hybrid approach to monitoring of QIF in which a dynamic analysis takes advantage of static analysis techniques. The static analyses will be able to retain precision because they exploit information from the execution and are applied locally. Furthermore the monitoring has strong formal guarantees thanks to the static analysis component that only analyses non-executed branches.

F. Contributions

- We propose a novel approach to hybrid information flow monitoring based on *tagging variables with the knowledge about secrets* rather than with security levels.
- We define a *generic hybrid monitor*, parametrized by a static analysis and give generic formal results on *relation between soundness and precision*.
  - We identify a soundness requirement on the static analysis which is sufficient to prove soundness of a generic hybrid monitor;
  - The genericity of the framework greatly facilitates the formal comparison of the precision of hybrid monitors.
- We instantiate a generic hybrid monitor with a combination of static dependency analysis and constant propagation, and derive three other monitors by weakening the static analyses (including monitors similar in spirit to those of Le Guernic *et al.* [19], [20]). We then prove that *our hybrid monitor is more precise* than three other monitors and establish a hierarchy of hybrid monitors, ordered by precision.

The paper is organised as follows. Section II defines the syntax and semantics of a simple programming language, designed for studying fingerprinting of browser features. Section III reviews basic definitions from quantitative information flow and derives a symbolic representation of knowledge. Section IV presents the generic hybrid monitor and Section V proves its correctness, relative to the correctness of the involved static analyses. Section VI defines a precise hybrid monitor based on constant propagation and dependency analysis and Section VII explains how other, simpler monitors can be obtained as instances of the generic monitor. Section VIII compares with related work and Section IX concludes. The correctness of the framework has been proved using the proof assistant Coq. The Coq model and the machine-checked proof of correctness can be found on an accompanying web page [27].

II. Language

We develop the monitor for a small, imperative programming language modified slightly to focus on fingerprinting of browser features. We assume an identified subset *Feat* of program variables that represents the browser features. Feature variables can be read but not assigned. We restrict the conditionals in if statements to be comparisons of features with variables and values, as these are the only tests that are relevant for the fingerprinting analysis. We will use the following notations

- Var is a set of all program variables;
- $Feat \subseteq Var$ is a set of variables that represent the browser features, ranged over by f
- Val is a set of values, including Boolean, integers and string values,
- $x \in Var \backslash Feat$ ranges over program variables that are not features;
- n is a constant: $n \in Val$ and;
- $\oplus$ is an arbitrary binary operator.

A program P is a command S followed by the output of a variable. Note that outputting a list of variables can be emulated by concatenating them using a special operator. The language's syntax is defined in Figure 3.

The semantics is defined in Figure 2 as a big-step evaluation relation $(S, \rho) \downarrow_C \rho'$. This relation evaluates a command S to be executed in an environment $\rho : Var \backslash Feat \mapsto Val$. The semantics is parametrized by a configuration $C : Feat \mapsto Val$ which remains unmodified during the evaluation. By *Config* we denote a set of all possible configurations.

$$[\textsc{skip}]\ \frac{}{(\mathsf{skip},\rho)\downarrow_C \rho} \qquad [\textsc{assign}]\ \frac{}{(x := E,\rho)\downarrow_C \rho[x \mapsto [\![E]\!]^\rho_C]} \qquad [\textsc{seq}]\ \frac{(S_1,\rho)\downarrow_C \rho' \qquad (S_2,\rho')\downarrow_C \rho''}{(S_1;S_2,\rho)\downarrow_C \rho''}$$

$$[\textsc{if}]\ \frac{[\![B]\!]^\rho_C = \alpha \qquad (S_\alpha,\rho)\downarrow_C \rho'}{(\mathsf{if}\ B\ \mathsf{then}\ S_{tt}\ \mathsf{else}\ S_{ff},\rho)\downarrow_C \rho'} \qquad [\textsc{while}]\ \frac{(\mathsf{if}\ B\ \mathsf{then}\ S;\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{else}\ \mathsf{skip},\rho)\downarrow_C \rho'}{(\mathsf{while}\ B\ \mathsf{do}\ S,\rho)\downarrow_C \rho'}$$

where

$$\begin{array}{lcl} [\![f \bowtie n]\!]^\rho_C & = & C(f) \bowtie n \\ [\![x \bowtie n]\!]^\rho_C & = & \rho(x) \bowtie n \\ [\![f \bowtie x]\!]^\rho_C & = & C(f) \bowtie \rho(x) \end{array} \qquad \begin{array}{lcl} [\![f]\!]^\rho_C & = & C(f) \\ [\![x]\!]^\rho_C & = & \rho(x) \end{array} \qquad \begin{array}{lcl} [\![n]\!]^\rho_C & = & n \\ [\![E_1 \oplus E_2]\!]^\rho_C & = & [\![E_1]\!]^\rho_C \oplus [\![E_2]\!]^\rho_C \end{array}$$

Figure 2: Language semantics

$$\begin{array}{l} P ::= S;\ \mathsf{output}\ x \\ S ::= \mathsf{skip} \mid x := E \mid S_1; S_2 \mid \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \\ \qquad \mid \mathsf{while}\ B\ \mathsf{do}\ S \\ B ::= f \bowtie n \mid x \bowtie n \mid f \bowtie x \\ \bowtie ::= \neq \mid = \\ E ::= n \mid f \mid x \mid E_1 \oplus E_2 \end{array}$$

Figure 3: Language syntax

III. Knowledge representation and quantitative leakage

A. Concrete domain of configurations

We will take as starting point the definitions of quantitative information flow from Köpf and Basin [14] and Smith [29]. Since our programs are deterministic, every program $S;\mathsf{output}\,o$ determines a partial function from a configuration C to an output v. In our notation, it means that the program has run under the configuration C: $(S,\rho_0)\downarrow_C \rho$ and produced an output v: $\rho(o) = v$. Following [14], [29], a program $S;\mathsf{output}\,o$ partitions *Config* according to the final value of o.

Definition 1 (Equivalence Class)**.** *Given a program S, a configuration C, an initial environment ρ_0 and an output variable o, an equivalence class is defined as*

$$Eq(S,C,\rho_0,o) = \left\{ C' \;\middle|\; \begin{array}{l} (S,\rho_0)\downarrow_C \rho \wedge (S,\rho_0)\downarrow_{C'} \rho') \\ \Rightarrow \rho(o) = \rho'(o) \end{array} \right\}.$$

Once the program S executes on the configuration C, a tracker can observe that the actual configuration of the user's browser is one of $Eq(S,C,\rho_0,o)$. How much does this equivalence class tell a tracker? If $Eq(S,C,\rho_0,o) = \mathit{Config}$, then the tracker has not learned anything about the actual configuration, hence no information flow has occurred. At the other extreme, if $Eq(S,C,\rho_0,o) = \{C\}$, by observing the output o, a tracker uniquely identifies C, which means *total leakage* of configuration C. All the other cases represent *partial leakage*.

Consider again the program from Figure 1. For the sake of simplicity, we assume that `name` and `fonts` are the only two browser properties we are interested in. Let the user's browser be "Opera". In this case `x = 0`, and the tracker is not able to conclude exactly the name of the user's browser. This partial leakage is precisely captured by the equivalence class of configurations with the name being different from "FireFox".

Following the definition of *self-information* (see Section I-D), we define a leakage function $\mathit{Leak} : \mathcal{P}(\mathit{Config}) \to \mathbb{R}^+$ for a set of configurations assuming that the probability of every configuration $P(C)$ is known (for example, from Panopticlick [11]).

Definition 2. *The* leakage *of a set of configurations $A \subseteq \mathit{Config}$ is defined as follows:*

$$\mathit{Leak}(A) = -\log_2 \sum_{C \in A} P(C).$$

The *Leak* function has the following properties:

- $\mathit{Leak}(\mathit{Config}) = 0$, which corresponds to the case of *non-interference*.
- For any couple of sets of configurations A_1 and A_2:

$$\text{If } A_1 \subseteq A_2 \text{ then } \mathit{Leak}(A_1) \geq \mathit{Leak}(A_2)$$

B. Symbolic representation of sets of configurations

We define an abstract domain of configurations, intended to represent a set of configurations, as follows:

$$\mathit{Config}^\sharp \ni cg ::= tt \mid ff \mid f \bowtie n \mid f \bar{\bowtie} n \mid cg \wedge cg \mid cg \vee cg$$

where $f \bar{\bowtie} n$ stands for $f \neq n$ if "$\bowtie$" is "$=$" and $f = n$ otherwise.

We write $\mathcal{M}(cg)$ for the models of the Boolean formula *cg i.e.,* the set of configurations that satisfy the Boolean formula *cg*.

During our analysis we will compute a Boolean formula *cg* for every output of the program. For example, for the output `x = 2` of the program from Figure 1, the resulting formula will be

$$(\mathit{name} \neq \text{"}\mathit{FireFox}\text{"}) \wedge (\mathit{fonts} = \mathit{fontsSet1})$$

Similar to the *leakage* function for a set of configurations, we define a leakage function $Leak^\sharp : Config^\sharp \to \mathbb{R}^+$ for Boolean formulas representing sets of configurations.

Definition 3. *The* leakage *of a Boolean formula* $cg \in Config^\sharp$ *is defined as follows:*

$$Leak^\sharp(cg) = -\log_2 \sum_{C \in \mathcal{M}(cg)} P(C).$$

The $Leak^\sharp$ function has the following properties:

- $Leak^\sharp(tt) = 0$, which corresponds to the case of *non-interference*
- For all $a_1, a_2 \in Config^\sharp$:

$$\text{If } a_1 \Rightarrow a_2 \text{ then } Leak^\sharp(a_1) \geq Leak^\sharp(a_2).$$

The last property of a $Leak^\sharp$ function is particularly important for our quantitative information flow monitors. As a result, our hybrid monitor will strive to weaken formulas: a weaker formula means that the computed leakage is smaller.

IV. Generic model of hybrid monitors

In qualitative ("high-low") information flow control, Le Guernic *et al.* [20] have shown that a dynamic information flow analysis can be improved by a static analysis of conditional branches that are not being taken. In this section, we generalise those results for quantitative information flow and define a generic hybrid monitor combining static and dynamic analysis. Moreover, we present a static analysis able to dynamically prove the non-interference of programs that were previously out of reach of existing hybrid monitors.

A. Formal Definitions

The monitor will be defined as an operational semantics, parametrized by the configuration C

$$(S, (\rho, K)) \Downarrow_C (\rho', K')$$

with a monitoring mechanism for tracking the information flow from the browser features to the output of the program. The new semantic state (ρ, K) has the following components:

- $\rho : Var \to Val$ is the environment for program variables.
- $K : Var \to Config^\sharp$ is an environment of knowledge about features stored in the non-feature variables. The knowledge is represented by a formula from the abstract domain $Config^\sharp$ (see Section III).

In traditional information flow analysis, variables are tagged with security levels, while our analysis is based on the knowledge environment K, that represents the knowledge in every program variable. This knowledge can either flow directly into the variable through an assignment or indirectly by updating the variable inside a conditional that depends on some feature value. Such a knowledge environment is thus a generalisation of a simple dependency function between variables, in that it contains additional information about the values of browser features. For example, a knowledge environment K may contain the following knowledge about the configuration: $K(x) = (name = \text{"}FireFox\text{"}) \wedge (fonts = fontsSet1)$. The initial knowledge environment K_0 is defined by $\forall x.K_0(x) = tt$, which means that no variable contains any knowledge about the browser configuration.

The monitor relies on the auxiliary function κ (defined in Figure 4) that approximates the information obtained from the evaluation of an expression. Evaluating a feature variable f will give access to its value and will therefore transmit the information $f = C(f)$ where C is the configuration of the browser. Accessing a non-feature variable provides the knowledge present in that variable as defined by the knowledge environment $K(x)$. An approximation of knowledge in an arithmetic expression $e_1 \oplus e_2$ is defined as a combination of knowledge in e_1 and in e_2.

The evaluation relation $\Downarrow_C$ defining the big-step semantics for the generic hybrid monitor parametrized by a configuration C is presented in Figure 4. The rules [SKIP], [SEQ], [IFELSE] and [WHILELOOP] correspond to the rules from the standard semantics and are straightforward. The rule [ASSIGN] updates the value environment with the new value of x. Notice that in traditional dynamic and hybrid information flow analysis [28], variable x would be assigned a "high" security level in case it is assigned within the "high" security context. In our setting, this would mean that the knowledge in variable x should be updated with the knowledge from the security context. We do not keep track of the security context, and, as we show in Section V, our monitors are sound and even more precise than the monitors that keep track of the security context.

The rule [IFTHEN] deals with the implicit flow of information due to conditionals. Assuming that the Boolean expression B evaluates to true, the semantics evaluates S_1 and statically analyse the non-executed branch S_2. The new monitor state $\{\!|B, K, s', s^\sharp|\!\}_\rho$ approximates the knowledge obtained from both branches. We explain this combination of states in Figure 5 immediately after the presentation of the static analysis.

B. The Role of the Static Analysis

The hybrid monitor is generic because it is parametrized on a static analysis providing information about the branches that are not being executed. The precision of the hybrid monitor can be improved if we know that the value of a variable, say x, after the non-executed branch is identical to the value of x after the executed branch. The static analysis computes an

$$[\textsc{skip}]\ \frac{}{(\mathsf{skip}, s) \Downarrow_C s} \qquad [\textsc{assign}]\ \frac{K' = K[x \mapsto \kappa(E)^K_C]}{(x := E, (\rho, K)) \Downarrow_C (\rho[x \mapsto [\![E]\!]^\rho_C], K')}$$

$$[\textsc{seq}]\ \frac{(S_1, s) \Downarrow_C s' \qquad (S_2, s') \Downarrow_C s''}{(S_1; S_2, s) \Downarrow_C s''} \qquad [\textsc{ifThen}]\ \frac{[\![B]\!]^\rho_C \qquad (S_1, (\rho, K)) \Downarrow_C s' \qquad (S_2, \rho) \Downarrow^\sharp s^\sharp}{(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, (\rho, K)) \Downarrow_C \ \{\!|B, K, s', s^\sharp|\!\}_\rho}$$

$$[\textsc{IfElse}]\ \frac{[\![\neg B]\!]^\rho_C \qquad (\mathsf{if}\ \neg B\ \mathsf{then}\ S_2\ \mathsf{else}\ S_1, s) \Downarrow_C s'}{(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, s) \Downarrow_C s'} \qquad [\textsc{whileLoop}]\ \frac{(\mathsf{if}\ B\ \mathsf{then}\ S; \mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{else}\ \mathsf{skip}, s) \Downarrow_C s'}{(\mathsf{while}\ B\ \mathsf{do}\ S, s) \Downarrow_C s'}$$

where

$$\kappa(f)^K_C = f = C(f) \qquad \kappa(x)^K_C = K(x) \quad \kappa(n)^K_C = tt \qquad \kappa(e_1 \oplus e_2)^K_C = \kappa(e_1)^K_C \wedge \kappa(e_2)^K_C$$

Figure 4: Semantics of a generic hybrid monitor.

abstract environment and the dependencies of a variable x, which is a set of variables needed to compute x. The analysis starts from a concrete environment ρ of values from the execution and computes an abstract state $s^\sharp = (\rho^\sharp, D)$, where $\rho^\sharp : Var \to Val \cup \{\top\}$ is an abstract environment and $D : Var \to \mathcal{P}(Var)$ is the dependency information for each variable. The role of an abstract environment $\rho^\sharp$ is to detect variables whose values are identical on both branches.

The results of the static analysis are used in the [IfThen] rule using the state combination defined in Figure 5. The auxiliary function δ is used for approximating the information coming from conditionals. The equations defining δ state that the comparison $f \bowtie n$ of a feature variable and a value will provide exactly that information. Comparing a non-feature variable x with a constant will at most provide the information about feature variables that were present in x. Finally, the comparison of a feature and a non-feature variable $f \bowtie x$ will at most transmit the information present in x and the information that f is equal to the current value of x, defined in the environment ρ.

The new environment ρ' is taken from the result of the executed branch and the new knowledge environment K'' is updated as follows.

If the values of a variable x are not the same after the execution of both branches, then x definitely obtains a complete knowledge about the conditional B. We represent this as a conjunction of the knowledge in x ($K'(x)$) with the knowledge in B ($\delta(B)^K_\rho$).

If the values of a variable x are the same after the execution of both branches, then the variable x does not contain a complete knowledge about the conditional test B. Instead, from the attacker point of view, the new knowledge in x can be obtained either from the executed branch or the non-executed branch. The formula we obtain can be understood as an abstraction of the standard *weakest precondition* of a conditional statement:

$$wp(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) = \bigwedge \begin{pmatrix} \neg B & \Rightarrow & wp(S_2) \\ B & \Rightarrow & wp(S_1) \end{pmatrix}$$

Here, the knowledge in x flowing from the non-executed branch is obtained from the knowledge of the variables used to compute x ($\bigwedge_{y \in D(x)} K(y)$) and the knowledge in x flowing from the executed branch is obtained by the monitoring mechanism. Notice that $\bar{\delta}(B)$ is not exactly the negation of $\delta(B)$ but an abstraction. It is because $\delta(B)$ is by construction an over-approximation of the knowledge of B.

Consider the following program:

```
1 x := 1; y := 0;
2 if (f = 0) then y := 1
3 else skip;
4 if (g = 0) then skip
5 else x := y;
6 output x
```

Here, `f` and `g` are feature variables that are equal to zero in the current configuration. Before the execution of the test `g = 0`, the variable `y` already contains some knowledge: $K(y) = (f = 0)$. Let's assume that a static analysis tracks the values and is able to detect that x depends on y. The resulting state of this static analysis after evaluating `x := y` is $\rho^\sharp(x) = 1, D(x) = \{y\}$. The resulting state after the execution of the `skip` branch would remain unchanged. Now, since the value of `x` would be the same and equal to 1 after the execution of either of the branches, the tracker would conclude that either `g = 0` or `f = 0`. Our combination of states computes exactly this knowledge: $(\delta(g = 0)^K_\rho \vee K(y)) \wedge K'(x) = (g = 0) \vee (f = 0)$.

Notice that there is no static analysis involved in purely dynamic monitoring, and still we can model it as a special case of our hybrid monitor. The abstract environment can be seen as $\forall x. \rho^\sharp(x) = \top$, and hence we obtain a simple dynamic monitor that does not reason about non-executed branches, but instead pessimistically decides

$\{\!|B, K, (\rho', K'), (\rho^\sharp, D)|\!\}_\rho = (\rho', K'')$, where

$$K''(x) = \begin{cases} \left(\neg\delta(B)^K_\rho \Rightarrow \bigwedge_{y \in D(x)} K(y)\right) \wedge \left(\neg\bar{\delta}(B)^K_\rho \Rightarrow K'(x)\right) & \text{if } \rho^\sharp(x) = \rho'(x) \\ \delta(B)^K_\rho \wedge K'(x) & \text{otherwise} \end{cases}$$

$$\begin{array}{llll} \delta(f \bowtie n)^K_\rho = f \bowtie n & \delta(x \bowtie n)^K_\rho = K(x) & \delta(f \bowtie x)^K_\rho = K(x) \wedge (f \bowtie \rho(x)) \\ \bar{\delta}(f \bowtie n)^K_\rho = f \bar{\bowtie} n & \bar{\delta}(x \bowtie n)^K_\rho = \mathit{ff} & \bar{\delta}(f \bowtie x)^K_\rho = \mathit{ff} \end{array}$$

Figure 5: State combination for the [IFTHEN] from Figure 4.

that all the variables will contain knowledge from the tests of the if-statements. This new knowledge in x will then contain the knowledge from the executed branch and from the test B: $\delta(B)^K_\rho \wedge K'(x)$.

V. Generic Soundness and Precision Theorems

In this section, we establish the soundness and precision theorems that hold for the generic model of hybrid monitors presented in Section IV. Comprehensive proofs can be found in the companion Coq development [27].

A. Monitor Soundness and Precision

A concrete hybrid monitor is obtained by instantiating the generic model by a given static analysis, say A. In the following, we write $\Downarrow^A$ for an hybrid monitor that uses the static analysis A. A monitor $\Downarrow^A$ is sound if after monitoring a statement S it over-approximates the knowledge about features contained in the output variable x. The formal statement of this property is given in Definition 4.

Definition 4 (Monitor soundness)**.** *A hybrid monitor $\Downarrow^A$ is sound if starting from an initial configuration C and the initial environment $(\rho, \lambda x.tt)$, it monitors a statement S and reaches a final configuration (ρ', K)*

$$(S, (\rho, \lambda x.tt)) \Downarrow^A_C (\rho', K)$$

such that for all variable x, $K(x)$ under-approximates the set of undistinguishable configurations

$$\mathcal{M}(K(x)) \subseteq Eq(S, C, \rho, x).$$

Notice that while $K(x)$ under-approximates a set of configurations, it over-approximates the knowledge of the attacker. Assume an attacker has the knowledge $K(x)$, that models a subset of actually undistinguishable configurations $Eq(S, C, \rho, x)$. Then she can more easily distinguish between the possible configurations, thus her knowledge is over-approximated.

The most precise monitor would compute $Eq(S, C, \rho, x)$ that is exactly the set of configurations indistinguishable from C by observing the value of x. In general, the closer the set $\mathcal{M}(K(x))$ is to $Eq(S, C, \rho, x)$, the more precise is the monitor.

Definition 5 (Monitor precision)**.** *A hybrid monitor $\Downarrow^A$ is more precise than a hybrid monitor $\Downarrow^B$ if for every statement S and initial configuration C, the monitor $\Downarrow^A$ always computes a bigger set of configurations corresponding to the knowledge stored in output variable x. Formally,*

$$\left.\begin{array}{l} (S, (\rho, K_0)) \Downarrow^A_C (\rho_A, K_A) \\ (S, (\rho, K_0)) \Downarrow^B_C (\rho_B, K_B) \end{array}\right\} \Rightarrow \mathcal{M}(K_B(x)) \subseteq \mathcal{M}(K_A(x)).$$

This is coherent with the definition of leakage in Section III because the leakage function is anti-monotonic in the set of configurations. Thus, a more precise monitor would estimate a smaller leakage, *i.e.*, a larger set of configurations:

$$\mathit{Leak}^\sharp(K_A(x)) \leq \mathit{Leak}^\sharp(K_B(x)).$$

In Section VI we will define a static analysis that will induce a sound monitor that is more precise than any other monitor we propose. This has the consequence that we can prove soundness of other monitors by proving that they are less precise than our hybrid monitor. This result is particularly useful when monitors are obtained by weakening the static analysis they employ, as is done when defining the hierarchy of monitors in Section VII.

B. Soundness Requirements for a Static Analysis

The generic hybrid monitor has a generic soundness proof relying only on a requirement for the static analysis. As explained in Section IV, the role of the static analysis is to extract executions within the non-executed branch that are indistinguishable from the executed branch and estimate the knowledge that is carried by the variables. Definition 6 provides the formal specification for static analyses that are compliant with our generic hybrid monitor.

Definition 6 (Sound Static Analysis)**.** *A static analysis $\Downarrow^\sharp$ is sound (for our hybrid dynamic monitor) if the*

following implication holds:

$$\left.\begin{array}{l}(S,\rho\downarrow_C\rho')\\(S,\rho_0)\Downarrow^\sharp(\rho'^\sharp,D)\\\rho'^\sharp(x)=v\\\forall y.y\in D(x)\Rightarrow\rho(y)=\rho_0(y)\end{array}\right\}\Rightarrow\rho'(x)=v.$$

Theorem 1 (Soundness)**.** *Suppose a sound static analysis $\Downarrow^\sharp$ according to Definition 6. Then the hybrid monitor $\Downarrow^{\Downarrow^\sharp}$ is sound according to Definition 4 and therefore safely approximates information leakage.*

The proof of this theorem is part of the Coq development [27].

C. Precision Requirements for a Static Analysis

The relative precision of different monitoring mechanisms is often difficult to establish, at least formally. In our generic hybrid monitor the precision of the monitor is directly linked to the strength of the static analysis: a better static analysis yields a more precise monitor.

Definition 7 (More Precise Analysis)**.** *An analysis A is more precise than an analysis B if for any result of the static analysis B, there exists a more precise result output by analysis A* i.e.*, the abstract environment is more defined and the set of variables computed is smaller.*

$$(S,\rho)\Downarrow^\sharp_B(\rho_B,D_B)\wedge(S,\rho)\Downarrow^\sharp_A(\rho_A,D_A)\Rightarrow\bigwedge\left\{\begin{array}{l}\forall x,v.\rho_B(x)=v\Rightarrow\rho_A(x)=v\\\forall x.D_A(x)\subseteq D_B(x)\end{array}\right.$$

Using the previous definition of precision, we are able to state the following generic theorem.

Theorem 2 (Relative Precision)**.** *If a static analysis A is more precise than a static analysis B (according to Definition 7) then the hybrid monitor $\Downarrow^A$ is more precise then the hybrid monitor $\Downarrow^B$ (according to Definition 5).*

The proof is by induction over the definition of the monitor semantics $\Downarrow$ and follows from the fact that all the rules are monotonic with respect to ordering of the knowledge K. This is especially the case for the IFTHEN because, as shown by Figure 5, a stronger analysis computes less spurious dependencies and therefore a weaker formula. Remember that weaker is better and that noninterference corresponds to computing the formula tt. The full proof is part of the Coq development [27].

This theorem is the key for comparing the different existing and novel hybrid monitors presented in Section VI and Section VII.

D. Where are the Security Contexts?

Security contexts are a traditional ingredient of static and dynamic information flow mechanisms. Perhaps surprisingly, our generic hybrid monitor is sound even in the absence of security context and ignoring the security context leads to a more precise monitor. Our generic hybrid monitor could incorporate a security context σ by rewriting the [ASSIGN] rule and the [IFTHEN] as

$$\frac{D'=D[x\mapsto\kappa(E)^D_C\wedge\sigma]}{(x:=E,(\rho,D,\sigma)\Downarrow_C(\rho[x\mapsto[\![E]\!]^\rho_C],D',\sigma)}$$

$$\frac{[\![B]\!]^\rho_C\qquad\sigma'=\sigma\wedge\delta(B)^D_\rho\qquad(S_1,(\rho,D),\sigma')\Downarrow_C s'\qquad(S_2,\rho)\Downarrow^\sharp s^\sharp}{(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2,(\rho,D,\sigma))\Downarrow_C(\ \{\!|B,D,s',s^\sharp|\!\}_\rho,\sigma)}$$

One explanation for this apparent paradox is that our [IFTHEN] incorporates the knowledge of the current condition and therefore includes the security context on a "lazy" basis.

Theorem 3 (Security Context)**.** *For a given (sound) static analysis, a monitor not using security contexts is always sound and more precise than a monitor using security contexts.*

This result is a direct consequence of Theorem 1 and the fact that the assignment rule with security context computes a stronger formula. The proof is also part of the Coq development [27].

For a big-step semantics, this reasoning is very natural but we believe the same precision can be achieved for a small-step operational semantics at the cost of some bureaucracy *e.g.,* an explicit stack of security contexts. What is crucial for precision is to never incorporate the knowledge of conditions during the [ASSIGN] rule. If programs were allowed to output values at any time, even our big-step semantics would require a security context. A simple approach would then consist in incorporating the knowledge of the security context only when a value is output.

It is also worth noting that ignoring the security context does not improve the purely dynamic monitor: the security context will eventually be included. However, the improvement is visible for hybrid monitors and allows to prove the absence of information flow in programs like `if C then x:=1 else x:=1; output x`.

VI. A HYBRID MONITOR WITH CONSTANT PROPAGATION AND DEPENDENCY ANALYSIS

In this section we define a hybrid monitor that employs a static analysis which can take full advantage of the concrete values available to the dynamic part of the hybrid monitor. Our static analysis is a combination of constant propagation and dependency analysis. As explained in Section IV-B, the hybrid monitor can take advantage of the fact that a variable has the same value on both branches of a conditional to make a more accurate estimation of the knowledge about features contained in that variable.

$$[\text{ASkip}]\ \frac{}{(\mathsf{skip}, s) \Downarrow^\sharp s} \qquad [\text{AAssignVal}]\ \frac{D' = D[x \mapsto \kappa^\sharp(E)^D]}{(x := E, (\rho, D)) \Downarrow^\sharp (\rho[x \mapsto [\![E]\!]^\sharp_\rho], D')}$$

$$[\text{ASeq}]\ \frac{(S_1, s_1) \Downarrow^\sharp s_2 \qquad (S_2, s_2) \Downarrow^\sharp s_3}{(S_1; S_2, s_1) \Downarrow^\sharp s_3} \qquad [\text{AIfTop}]\ \frac{[\![B]\!]^\sharp_\rho = \top \qquad (S_1, s) \Downarrow^\sharp s_1 \qquad (S_2, s) \Downarrow^\sharp s_2}{(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, s) \Downarrow^\sharp s_1 \sqcup s_2}$$

$$[\text{AIfComb}]\ \frac{[\![B]\!]^\sharp_\rho = tt \qquad (S_1, (\rho, D)) \Downarrow^\sharp s_1 \qquad (S_2, (\rho, D)) \Downarrow^\sharp s_2}{(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, (\rho, D)) \Downarrow^\sharp \ \{\!|B, D, s_1, s_2|\!\}^\sharp_\rho}$$

$$[\text{AIfElse}]\ \frac{[\![B]\!]^\sharp_\rho = ff \qquad (\mathsf{if}\ \neg B\ \mathsf{then}\ S_2\ \mathsf{else}\ S_1, s) \Downarrow^\sharp s'}{(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, s) \Downarrow^\sharp s'} \qquad [\text{AWhile}]\ \frac{(S, s') \Downarrow^\sharp s_1 \qquad s_1 \sqsubseteq s' \qquad s \sqsubseteq s'}{(\mathsf{while}\ B\ \mathsf{do}\ S, s) \Downarrow^\sharp s'}$$

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
\kappa^\sharp(n)^D & = & \emptyset & [\![n]\!]^\sharp_\rho & = & n & [\![f \bowtie n]\!]^\sharp_\rho & = & \top \\
\kappa^\sharp(f)^D & = & \emptyset & [\![f]\!]^\sharp_\rho & = & \top & [\![f \bowtie x]\!]^\sharp_\rho & = & \top \\
\kappa^\sharp(x)^D & = & D(x) & [\![x]\!]^\sharp_\rho & = & \rho(x) & [\![x \bowtie n]\!]^\sharp_\rho & = & [\![x]\!]^\sharp_\rho \bowtie^\sharp [\![n]\!]^\sharp_\rho \\
\kappa^\sharp(e \oplus e')^D & = & \kappa^\sharp(e)^D \cup \kappa^\sharp(e')^D & [\![e \oplus e']\!]^\sharp_\rho & = & [\![e]\!]^\sharp_\rho \oplus^\sharp [\![e']\!]^\sharp_\rho & & &
\end{array}$$

$$\text{with } v \odot^\sharp v' = \begin{cases} \top & \text{if } v = \top \vee v' = \top \\ v \odot v' & \text{otherwise} \end{cases} \quad \text{where } \odot \in \{\oplus, \bowtie\}.$$

Figure 6: Constant propagation and dependency analysis.

An abstract state $(\rho^\sharp, D) \in State^\sharp$ is a pair of:

- an abstract environment $\rho^\sharp : Var \to Val \cup \{\top\}$ and $\top$ represents an arbitrary value,
- a dependency function $D : Var \to \mathcal{P}(Var)$ such that the computation of x depends upon a set of variables $D(x)$.

Abstract states are equipped with a partial order $\sqsubseteq$ obtained as the Cartesian product of the ordering of abstract values: $\forall x, y. x \sqsubseteq y$ iff $x = y \vee y = \top$ and the point-wise lifting of the standard set inclusion $\mathcal{P}(Var)$. The join operator $\sqcup$ is the least upper bound induced by the ordering $\sqsubseteq$.

The static analysis is specified in Figure 6 as a syntax-directed set of inference rules that generate constraints over abstract states. The static analysis of a program S is defined as a function between abstract states, written $(S, s) \Downarrow^\sharp s'$, such that s' is the least abstract state solution to the constraints. The intented meaning is that s' is a valid abstraction of the result obtained when running program S in an initial state that is modelled by an abstract state s.

The [AIfComb] rule combines the states after the analysis of two branches in case the test of the if-statement can be evaluated in the given abstract environment. The state combination $\{\!|B, D, s_1, s_2|\!\}^\sharp_\rho$ from Figure 7 is the abstraction of the combination of the states from executed and non-executed branches that we defined in Figure 5. In disjunctive normal form, the logical formula for obtaining K'' is of the form

$$(\delta(B)^K_\rho \wedge K'(x)) \ \vee\ (\bigwedge_{y \in D(x)} K(y) \wedge K'(x)) \vee \ldots \quad (1)$$

In the rest of this section, we explain why the set D'' (see Figure 7) represents an under-approximation of this formula. Note that the static analysis can safely ignore the other terms of the formula, here represented by $\ldots$. If the values of x are possibly different after both branches, we combine the knowledge possibly obtained by reading x in the executed branch and in the test B. If the values of x are the same after both branches, then x gets the knowledge either from the executed branch and the test B or just from both branches.

The state combination for static analysis in Figure 7, uses auxiliary sets of variables: $D_{true}(x)$ and $D_{both}(x)$. The set $D_{true}(x)$ is the set of variables in the test B and the set of variables, on which x depends after the potential execution of the true branch. This set represents the same idea that was used in the state combination of hybrid monitor: it corresponds to the knowledge in the formula $(\delta(B)^K_\rho \wedge K'(x))$. The set $D_{both}(x)$ computes a set of variables on which computation of x depends in both branches. This set corresponds to the knowledge in the formula $\bigwedge_{y \in D(x)} K(y) \wedge K'(x)$.

Now, when we construct a new dependency set $D''(x)$, in case the values of x are different the $D_{true}(x)$ set is taken. This case is a straightforward translation of the same condition in Figure 5. In case the values of x are different, we would like to approximate the knowledge we computed in formula (1) by the set of variables. Since

$\{\!|B, D, (\rho_t, D_t), (\rho_f, D_f)|\!\}^\sharp_\rho = (\rho_t, D'')$, where

$$D''(x) = \begin{cases} D_{true}(x) \bar{\vee} D_{both}(x) & \text{if } \rho_t(x) = \rho_f(x) \\ D_{true}(x) & \text{otherwise} \end{cases} \qquad X \bar{\vee} X' = \begin{cases} X & X \subseteq X' \\ X' & \text{otherwise} \end{cases}$$

$$D_{true}(x) = \mathcal{V}^D(B) \cup D_t(x) \qquad \mathcal{V}^D(f \bowtie n) = \emptyset$$
$$D_{both}(x) = D_f(x) \cup D_t(x) \qquad \mathcal{V}^D(x \bowtie n) = D(x)$$
$$\mathcal{V}^D(f \bowtie x) = D(x)$$

Figure 7: State combination for the [AIfComb] rule of the static analysis from Figure 6.

the resulting set of variables $D(x)$ will later be used by a hybrid monitor to compute a conjunction $\bigwedge_{y \in D(x)} K(y)$, we cannot approximate the disjunction with the set of variables. Hence, we propose to choose one set, either D_{true} or D_{both}, which is more precise than the other.

Notice, that if $X \subseteq X'$, then the formula for set X that is computed by hybrid monitor, is weaker than the formula for set X', because it is a conjunction of a smaller set of variables. Hence, the leakage computed from X is smaller than the leakage computed form X'.

We prove the soundness requirement for the static analysis presented in Figure 6.

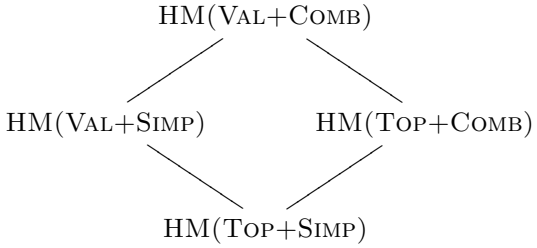
Theorem 4 (Static Analysis Soundness)**.** *The static analysis $\Downarrow^\sharp$ is sound according to Definition 6.*

The proof of this theorem is part of the Coq development [27].

Consider an example of a non-interferent program 4 from Table I: when `A` is true, `x = 1` and otherwise `x = 1` because `y = 1`. The knowledge about the value of `C` is contained in `z`, however it does not influence the value of `x` because there is no execution where `x` would be assigned to `z`. To explain the static analysis, let's consider the case when `A` is true. The static analysis starts from the branch `if (y = 1) then x := y; else x := z` and since the test `y = 1` can be evaluated, the rule [AIfComb] is applied. The resulting states from the branches `x := y` and `x := z` are combined according to the static analysis state combination, where the auxiliary sets are: $D_{true}(x) = \{y\}$ and $D_{both}(x) = \{y, z\}$. Here, the value of `z` does not influence the decision of the new dependency set because $D_{true}(x) \subset D_{both}(x)$. Hence, $D''(x) = D_{true}(x) = \{y\}$.

Then, the hybrid monitor combines the results of the static analysis and of the executed branch in the [IfThen] rule from Figure 4. In this rule, $D(x) = \{y\}$. Since our monitor does not track the security context, the knowledge in `x` after the execution of the branch `skip` is $K'(x) = tt$ and `y` does not contain any knowledge: $K(y) = tt$. Therefore, $K''(x) = (\delta(B)^K_\rho \vee K(y)) \wedge (\bar{\delta}(B)^K_\rho \vee K'(x)) = tt$. Formula tt corresponds to no knowledge, and the leakage of this program is 0 bits.

This example clearly shows that our hybrid monitor will recognise the non-interference of this program, however other dynamic and hybrid information flow techniques would mark `x` with "high" security label, since it has been assigned under the security context of a secret condition `A`.



where

Val = [AAssignVal] Comb = [AIfComb]
Top = [AAssignTop] Simp = [AIfSimple]

Figure 8: The hierarchy of hybrid monitors

VII. A hierarchy of hybrid monitors

Next, we examine three variants of the monitor from the previous section, obtained by modifying the constant propagation and dependency analyses. These modifications are defined by replacing the rules for assignment and conditionals in the definition of the static analyses (Figure 6). We shall name each monitor by HM(X+Y) where X is the name of the rule for assignment and Y is the rule for conditionals used. The systematic way in which these monitors are derived makes it easy to organise them into a hierarchy of relative precision, depicted in Figure 8. All the precision theorems in this section are a direct consequence of Theorem 2 stating that more precise static analysis induces more precise hybrid monitoring. The proofs of the theorems have been left out for lack of space—see [27].

Table I presents examples of 2 programs that leaks some information about the secrets and 2 non-interfering programs. To simplify the examples, the secrets `A, B` and `C` denote the tests on the browser features that were represented by $f \bowtie n$ in the original syntax. Notice that program 1 represents the original example

Table I: Quantification of leakage by different hybrid monitors when `A, B` and `C` are true.

	Program 1	**Program 2**	**Program 3**	**Program 4**

	`x := 1; y := 1;`
	`if A then y := 0;`
	`if (y = 1) then`
	`  x := 0;`
	`if B then`
	`  if (y = 1) then`
	`  x := 2`
	`output x;`
		`x :=1; y := 0; z := 1;`
		`if A then skip`
		`else y := 1;`
		`if B then skip`
		`else`
		`  if (y = 1) then`
		`  z := 0`
		`if C then skip`
		`else x := z;`
		`output x;`
			`x := 1; y := 1; z := 0;`
			`if A then z := 1;`
			`if B then skip`
			`else`
			`  if (y = 1) then skip`
			`  else x := z;`
			`output x;`
				`x := 1; y := 1; z := 0;`
				`if C then z := 1;`
				`if A then x := 1`
				`else`
				`  if (y = 1) then`
				`  x := y`
				`  else x := z;`
				`output x;`

Actual leakage	A - **2.25** bits	$A \vee B \vee C$ – **1.61** bits	tt – **0** bits	tt – **0** bits
HM(Val+Comb)	A – **2.25** bits	$A \vee B \vee C$ – **1.61** bits	tt – **0** bits	tt – **0** bits
HM(Val+Simp)	$A \wedge B$ – **8.89** bits	$B \vee C$ – **2.75** bits	$A \vee B$ – **2.20** bits	$A \vee C$ – **1.64** bits
HM(Top+Comb)	A – **2.25** bits	C – **2.84** bits	tt – **0** bits	A – **2.25** bits
HM(Top+Simp)	$A \wedge B$ – **8.89** bits	C – **2.84** bits	B – **6.64** bits	A – **2.25** bits

of fingerprinting code from Figure 1. We only substitute a test `(name = "FireFox")` with `A` and a test `(fonts = fontsSet1)` with `B`.

These programs illustrate the difference in precision of hybrid monitors. For every monitor and program we specify a formula that represents the knowledge in `x` in the end of the execution (when `A`, `B`, and `C` are true) and a corresponding amount of leakage in bits computed from the obtained formula.

To provide the estimation of leakage we assume the corresponding probabilities for `A`, `B`, and `C` to be true: $P(A) = 0.21$ (test on the "FireFox" browser name), $P(B) = 0.01$ (test on a concrete list of fonts), $P(C) = 0.14$ (test on a time zone). We also assume that the browser features represented by `A`, `B`, and `C` are independent. We then compute probabilities for events in its usual sense, for example

$$\begin{aligned} P(A \wedge B) &= P(A) \cdot P(B) = 0.0021 \\ P(A \vee B \vee C) &= 1 - P(\neg A)P(\neg B)P(\neg C) = 0.327 \end{aligned}$$

Then the leakage computed as a self-information (a logarithm of an event), for example the leakage of information in formula $A \wedge B$ is $-\log_2 P(A \wedge B) = -\log_2 0.0021 = 8.89$ bits, while in formula $A \vee B \vee C$ it is $-\log_2 P(A \vee B \vee C) = 1.61$ bits.

Notice that Table I also presents examples of programs 3 and 4 that are non-interferent. These programs illustrate the difference in precision of hybrid monitors. When a monitor computes that the output variable `x` contains 0 bits of information, the monitor recognises the program as non-interferent. Hence this monitor is more precise than other monitors that compute some leakage different than 0 bits.

A. The HM(Val+Simp) monitor

The precise treatment of conditionals in the static analysis from Figure 6 attempts to determine the actual value of the Boolean conditional by the constant propagation analysis. A simpler analysis would abandon this idea and just assume that both branches might be executed. Instead of [AIfTop], [AIfComb] and [AIfElse] rules, this analysis uses one simple rule for if-statements:

$$[\text{AIfSimple}]\ \frac{(S_1, s) \Downarrow^\sharp s_1 \qquad (S_2, s) \Downarrow^\sharp s_2}{(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, s) \Downarrow^\sharp s_1 \sqcup s_2}$$

Theorem 5. *The monitor* HM(Val+Comb) *is more precise than the monitor* HM(Val+Simp).

To illustrate the difference in precision, consider a program 4 from Table I. This program is non-interferent and our HM(Val+Comb) monitor correctly computes 0 bits of leakage (see section Section VI for more details).

We consider the case when `A` and `C` are true. Then, `z` is updated to 1 and it contains knowledge $K(z) = C$. The static analysis of HM(Val+Simp) monitor ignores the Boolean conditional `(y = 1)`, and hence it computes a set of variables $D(x) = \{y, z\}$. [IfThen] rule of the hybrid monitor computes

$$\begin{aligned} K''(x) &= (\delta(A)^K_\rho \vee (K(y) \wedge K(z))) \wedge (\bar{\delta}(A)^K_\rho \vee K'(x)) \\ &= (A \vee (tt \wedge C)) \wedge (\neg A \vee tt) = A \vee C \end{aligned}$$

The amount of leakage in this case is 0.83 bits, that clearly shows that HM(Val+Simp) monitor is less precise than HM(Val+Comb) monitor that computes 0 bits of leakage for this program execution.

B. The HM(Top+Simp) monitor

Le Guernic *et al.* [20] proposed a hybrid information flow monitor that uses static analysis for non-executed branches. The idea of the analysis is to compute a set of variables *modified* that might be assigned to some value in the non-executed branch. Then, all the variables in *modified* are tagged with a "high" label in case the test of the if-statement contained some "high" (secret) variables. In a later work, Russo and Sabelfeld [28] define a generic framework of hybrid monitors where such syntactic checks are proposed as well.

To compare our monitors with the monitor of Le Guernic et al. [20], we propose a static analysis that sets the abstract value of a variable to $\top$ as soon as it gets assigned. By doing so, $\rho^\sharp(x) = \top$ means that $x \in$ *modified*. Concretely, we take the static analysis from the HM(Val+Simp) monitor and substitute the [AAsignVal] rule with the following rule:

$$[\text{AAssignTop}]\ \frac{\rho' = \rho[x \mapsto \top]}{(x := E, (\rho, D)) \Downarrow^\sharp (\rho', tt)}$$

With this static analysis in mind, the idea of syntactic checks is already considered in our generic hybrid monitor. Whenever x is in *modified*, its value will be $\top$, and hence according to the state combination procedure in Figure 5, the knowledge of the test will be added to the knowledge of x.

Theorem 6. *The* HM(Val+Simp) *monitor* is more precise *than the* HM(Top+Simp) *monitor* .

All the programs from Table I illustrate that the HM(Val+Simp) monitor evaluates the leakage more precisely than the HM(Top+Simp) monitor.

C. The HM(Top+Comb) monitor

In a later work, Le Guernic [19] proposed a more generic framework of hybrid monitors that use static analysis. One of the novelties of this work is that the static analysis should ignore the branch that will not be executed (according to the current environment) if the test before the branch does not contain any secret variables. The formalization of this principle in our approach is a static analysis that uses the [AAssignTop] rule for assignment and [AIfComb] rule for if-statements.

Theorem 7. *The* HM(Top+Comb) *monitor* is more precise *than the* HM(Top+Simp) *monitor.*

In his PhD thesis [18], Le Guernic has proven that if a monitor on which we base HM(Top+Simp) monitor concludes that a variable x does not contain secret information, then the monitor similar to HM(Top+Comb) also concludes that variable x does not contain any secret information. Our framework generalises this proof since our notion of precision is based on the *amount of knowledge* in a variable. Programs 1 and 3 in Table I illustrate this difference in precision.

Theorem 8. *The* HM(Val+Comb) *monitor* is more precise *than the* HM(Top+Comb) *monitor.*

To illustrate this precision result, consider the program 4 from Table I. The static analysis used by the HM(Top+Comb) monitor marks all the assigned variables as $\top$ because of the syntactic nature of [AAssignTop] rule. Hence, $\rho^\sharp(x) = \top$, and so `x` will contain some knowledge about `A`.

Notice that programs 1 and 2 from Table I illustrate that the HM(Top+Comb) and HM(Val+Simp) monitors are incomparable in a sense of their relative precision.

VIII. Discussion and Related work

A. Hybrid Information Flow Monitoring

Hybrid monitors for information flow control that combine static and dynamic techniques have recently become popular [19], [20], [25], [28]. One of the first techniques was proposed by Le Guernic *et al.* [20] where the static analysis only performs syntactic checks on non-executed branches. This approach fits into our framework as HM(Top+Simp) monitor and it is proven to be less precise than the other monitors we propose. Russo and Sabelfeld [28] introduced a generic framework of hybrid monitors, where non-executed branches are also analysed only syntactically. In the follow-up work Le Guernic [19] presented a more permissive static analysis, that ignores possible branches that depend only on public variables. Inspired by this approach, we introduced HM(Top+Comb) monitor that is proven to be less precise than HM(Val+Comb) monitor.

Devriese and Piessens [10] proposed a *secure multi-execution* (SME) technique which falls outside of the static, dynamic, or hybrid classification. The basic principle is to multi-execute the program for every security level while filtering inputs and outputs and thus enforcing non-interference. The approach was shown to be efficient in a web browser environment [1], [9], when the security lattice consists of two levels: secret and public.

In our setting, each secret variable (browser feature) has a different security level (different knowledge), and combination of variables yields a creation of a new security level. In this case the security lattice grows exponentially (with the growth of a boolean formula) and SME approach would not be an efficient solution.

B. Protection against Excessive Leakage

Our hybrid monitor computes an over-approximation of the knowledge extracted from the observation of the program output. To protect herself against excessive leakage, a user can set a threshold on the maximum number of bits she agrees to leak. The hybrid monitor would estimate the leakage and either halt the program execution, or perform other actions so that the leakage would not exceed the threshold.

One of such actions can be a suppression of program output, another possibility could be editing the output. Such enforcement actions can be modelled by edit automata [5] that are designed to enforce desired security policies without halting the program.

In our current model, a program only outputs a single final value and we assume that non-termination is not observable. In this setting, the hybrid monitor can turn a potential excessive leakage into an absence of leakage by halting the program just before the output statement. A consequence is that our hybrid monitor can enforce *termination-insensitive non-interference* by suppressing any output which leakage is different from zero.

If termination is observable, halting the program might leak information. To cope with this issue, Mardziel *et al.* [21] perform a worst-case static analysis which computes an over-approximation of the leakage of all executions. If the over-approximation is above the leakage threshold, the program is not executed. In our fingerprinting context, this approach would likely be very pessimistic as the program would not be executed as soon as there exists one single user for which the program can learn too much information. Future work will investigate how our hybrid monitor can estimate the information leakage due to halting the execution, and how to decrease it below the threshold (*e.g.*, by lying about the browser configuration).

Another important property to consider is *correction soundness* introduced by Le Guernic [19]. A monitor is correction sound if on two executions that agree on public inputs, the low outputs get the same security level in the end of the execution. If the monitor enforcing non-interference is not correction-sound, it introduces new information leaks due to different enforcement reaction on different secret inputs. Our hybrid monitor would not obey this property, because when the program is non-interferent, there might be some secret inputs, for which the output would contains some information according to our monitor. The question of correction soundness is worth a deeper investigation for monitors that track the knowledge flow of the program.

C. Quantitative Information Flow Analysis

There are several approaches to quantify the information learned by a public observer about the secret program inputs. Existing work based on static analysis *e.g.*, [3], [6], [15], [21] aim at quantifying the information flow of a program and therefore rely on metrics that summarise the information flow of *all the executions*. Our hybrid monitoring technique aims at estimating the leakage of a single execution. As the leakage can be very different from one execution to the other, an advantage of our hybrid monitor technique is that it can potentially take more informed counter-measures based on the estimated amount of leaked information.

Clarkson, Myers, and Schneider [7] define the belief of the observer about secret inputs as a probability distribution, and show how to refine this belief by observing concrete executions of the program. A strength of this model is that it accommodates for inaccurate beliefs. Our threat model is simpler and assumes that the initial belief of the attacker *i.e.,* the probability distribution of browser properties, is accurate. Clarkson, Myers and Schneider also show that *self-information* [7, Section 4.1] is the adequate notion to quantify the amount of information leaked by the observation of the output of a single execution. Based on this belief tracking approach, Mardziel *et al.* [21] propose an enforcement mechanism for knowledge-based policies. The knowledge of the observer is a probability distribution of secret variables, and the static analysis of the program makes a decision to run or reject the program. In case there exists a value of some secret variable that *may increase the knowledge* of the observer above some predefined threshold, the program is rejected. The approach is static and could reject a program whereas a specific concrete execution could actually leak very little information. Also, Mardziel *et al.* keep history of the knowledge gained by the observer. This knowledge is updated whenever an observer sees more program outputs. Modelling a sequence of outputs is currently out of reach of our model. We shall consider such extension and incorporation of history in the future work.

Backes *et al.* [3] compute the number of equivalence classes and their sizes by statically analysing the program, and evaluate the leakage using entropy-based measurements. Like us they use a symbolic representation of equivalence classes and leakage computation also requires model enumeration. Our hybrid monitor restricts the expressiveness of the logic used to represent symbolically equivalence classes. This is done at the cost of precision. For instance, using arithmetic reasoning, it is straightforward to deduce that the expression $x - x$ does not leak any knowledge. Our hybrid monitors considers that $\kappa(x - x) = \kappa(x) \wedge \kappa(x) = \kappa(x)$. However, the advantages are twofold: the hybrid monitor is fast which is mandatory for online monitoring; the hybrid monitor is language agnostic which allows to deal with arbitrary language operators.

Köpf and Rybalchenko [15] bound the leakage of a program by combining the over- and under-approximation of the leakage of randomised concrete executions. Lower bounds of leakage are obtained by instantiating a relational static analysis with concrete values. Upper bounds are obtained by a symbolic backward analysis of the concrete execution path. Our hybrid monitor is using a tighter combination of static and dynamic analysis. In terms of precision, the techniques are not comparable. A symbolic backward analysis of a concrete execution path would have the precision of a purely dynamic monitor that would not abstract the knowledge of expressions. Our hybrid monitor might be less precise because it abstracts the knowledge of expressions. However, it gains precision over a symbolic execution because its static analysis explores (infinitely) many paths and refines the leakage in case it can prove they produce the same output.

To the best of our knowledge, the only dynamic analysis for quantitative information flow was proposed by McCamant and Ernst [23]. It uses a channel capacity metrics for information leakage. The channel capacity defines the smallest probability distribution, and hence puts an upper bound on the amount of leaked information. This approach is not precise enough in our setting since the probability distributions are known a-priori.

IX. Conclusions

Fingerprinting of browsers is a technique for tracking users on the web without storing data in their browser. By running fingerprinting scripts, web trackers can learn about specific features of the user's browser configuration and thereby effectively identify the user (more precisely, her browser). However, the effectiveness of a script highly depends on the web browser configuration.

We propose to evaluate the amount of information a web tracker learns by observing the output of a fingerprinting script for a particular browser configuration. To quantify the leakage precisely for a specific user, *i.e.* for a specific browser configuration, we propose a hybrid analysis technique computing a symbolic representation of the knowledge that a script obtains about the browser configuration.

We have developed a generic framework for modeling hybrid monitors that are parametrized by static analyses. The framework most notably proposes a generic soundness requirement on the static analysis which is sufficient to prove soundness of a hybrid monitor. This generic framework can be used to prove relative precision of hybrid information flow monitors.

We have instantiated the generic monitor with a combined static constant propagation and dependency analysis. This analysis provides more precise results for non-executed branches than in previous works. Moreover, our symbolic representation of knowledge allows us to benefit from the constant propagation analysis and to model the tracker's knowledge about a browser configuration more precisely. Concretely, our approach gains precision in those cases where a tracker will observe the same value for a given variable after the execution of either of the branches. We have proved that our monitor is more precise than the other hybrid information flow monitors found in the literature.

The entire theory has been modelled and verified [27] using the Coq proof assistant. Using Coq has been very productive to explore this rather new area on the frontier between security monitors and static analysis in a semantically correct way.

For further work, Section VIII already discussed extensions towards correction soundness, threshold-based enforcement and the security guarantee that it can provide. In addition, our hybrid analyses are defined for a simple programming language with focus on the principles behind the mechanism. We would have to scale to such languages as JavaScript for real deployment. Hedin and Sabelfeld [13] have shown it possible to analyse JavaScript using a purely dynamic information flow technique. Their system seems an ideal candidate to instrument with our monitor in order to track and quantify the information a tracker can deduce about possible configurations by observing the program outputs.

Acknowledgements: The authors are grateful to Boris Köpf, Alan Schmitt and the anonymous reviewers for valuable comments on earlier versions of this paper. We also thank Michael Hicks for fruitful discussions about this work.

References

[1] Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proc. of the 39th Symposium of Principles of Programming Languages. ACM (2012)

[2] Ayenson, M., Wambach, D.J., Soltani, A., Good, N., Hoofnagle, C.J.: Flash cookies and privacy II: Now with HTML5 and ETag respawning. In: SSRN eLibrary (2011).

[3] Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proc. of the 2009 Symposium on Security and Privacy. pp. 141–153 (2009)

[4] Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. Communications of the ACM 52, 83–91 (2009)

[5] Bauer, L., Ligatti, J., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. International Journal of Information Security 4(1-2), 2–16 (2005)

[6] Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15(3), 321–371 (2007)

[7] Clarkson, M.R., Myers, A.C., Schneider, F.B.: Quantifying information flow with beliefs. Journal of Computer Security 17(5), 655–701 (2009)

[8] Commission, E.: Commission proposes a comprehensive reform of the data protection rules, Online: http://ec.europa.eu/justice/newsroom/data-protection/news/120125_en.htm

[9] De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: Flowfox: a Web Browser with Flexible and Precise Information Flow Control. In: Proc. of the 19th ACM Conference on Communications and Computer Security. pp. 748–759 (2012)

[10] Devriese, D., Piessens, F.: Non-interference through secure multi-execution. In: Proc. of the 2010 Symposium on Security and Privacy. pp. 109–124. IEEE (2010)

[11] Eckersley, P.: The Panopticlick project, https://panopticlick.eff.org

[12] Eckersley, P.: How unique is your browser? In: Proc. of the 2010 Privacy Enhancing Technologies Symposium. LNCS, vol. 6205, pp. 1–18. Springer (2011)

[13] Hedin, D., Sabelfeld, A.: Information-flow security for a core of javascript. In: Proc. of the 25th Computer Security Foundations Symposium. pp. 3–18. IEEE (2012)

[14] Köpf, B., Basin, D.: An information-theoretic model for adaptive side-channel attacks. In: Proc. of the 14th ACM conference on Computer and communications security. pp. 286–296. ACM (2007)

[15] Köpf, B., Rybalchenko, A.: Approximation and randomization for quantitative information-flow analysis. In: Proc. of the 23rd Computer Security Foundations Symposium. pp. 3–14. IEEE (2010)

[16] Kroes, N.: Privacy online: USA jumps aboard the Do-Not-Track standard (February 23, 2012), online: http://blogs.ec.europa.eu/neelie-kroes/usa-do-not-track/

[17] Kroes, N.: Why we need a sound Do-Not-Track standard for privacy online (January 20, 2012), online: http://blogs.ec.europa.eu/neelie-kroes/donottrack/

[18] Le Guernic, G.: Confidentiality Enforcement Using Dynamic Information Flow Analyses. Ph.D. thesis, Kansas State University (2007)

[19] Le Guernic, G.: Precise Dynamic Verification of Confidentiality. In: Proc. of the 5th International Verification Workshop. CEUR Workshop Proc., vol. 372, pp. 82–96 (2008)

[20] Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-based Confidentiality Monitoring. In: Proc. of the Annual Asian Computing Science Conference. LNCS, vol. 4435, pp. 75–89. Springer (2006)

[21] Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic Enforcement of Knowledge-based Security Policies. In: Proc. of the Computer Security Foundations Symposium. pp. 114–128. IEEE (2011)

[22] Mayer, J.R., Mitchell, J.C.: Third-party web tracking: Policy and technology. In: Proc. of the 2012 Symposium on Security and Privacy. pp. 413–427. IEEE (2012)

[23] McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: Proc. of the ACM 2008 Conf. on Programming Language Design and Implementation. pp. 193–205. ACM (2008)

[24] McDonald, A., Cranor, L.: A survey of the use of adobe flash local shared objects to respawn http cookies. Tech. Rep. 11-001, Carnegie Mellon CyLab (January 2011)

[25] Moore, S., Chong, S.: Static analysis for efficient hybrid information-flow control. In: Proc. of the 24th Computer Security Foundations Symposium. pp. 146–160 (2011)

[26] Party, A..D.P.W.: Letter to the online advertisement industry (August 3, 2011), Available online: http://ec.europa.eu/justice/data-protection/article-29/documentation/other-document/files/2011/20110803_letter_to_oba_annexes.pdf

[27] QIF project: Coq proof of monitor correctness, http://www.irisa.fr/celtique/ext/QIF/

[28] Russo, A., Sabelfeld, A.: Dynamic vs. Static Flow-Sensitive Security Analysis. In: Proc. of the 23rd Computer Security Foundations Symposium. pp. 186–199. IEEE (2010)

[29] Smith, G.: On the Foundations of Quantitative Information Flow. In: Foundations of Software Science and Computational Structures. LNCS, vol. 5504, pp. 288–302. Springer (2009)

[30] Soltani, A.: Respawn redux (August 2011). Online: http://ashkansoltani.org/docs/respawn_redux.html

[31] Soltani, A., Canty, S., Mayo, Q., Thomas, L., Hoofnagle, C.J.: Flash cookies and privacy. In: AAAI Spring Symposium: Intelligent Information Privacy Management (2010)