On-the-Fly Inlining of Dynamic Dependency Monitors for Secure Information Flow

Luciano Bello[1,2] and Eduardo Bonelli[2,3,4]

[1] Si6 Labs - CITEDEF - Inst. de Investigac. Cient. y Técnicas para la Defensa
lbello@citedef.gob.ar
[2] ITBA - Instituto Tecnológico Buenos Aires
[3] CONICET - Consejo Nacional de Investigaciones Científicas y Técnicas
[4] UNQ - Univesidad Nacional de Quilmes
ebonelli@unq.edu.ar

Abstract. Information flow analysis (IFA) in the setting of programming languages is steadily veering towards the adoption of dynamic techniques. This is particularly attractive for scripting languages for web applications programming. A common manifestation of dynamic techniques is that of run-time monitors, which should block program execution in the presence of an insecure run. Significant efforts are still required before practical, scalable monitors for secure IFA of industrial scale languages such as JavaScript can be achieved. Such monitors ideally should compensate for the absence of the traces they do not track, should not require modifications of the VM and should provide a fair compromise between security and usability among other things. This paper discusses on-the-fly inlining of monitors that track dependencies as a prospective candidate.

1 Introduction

Secure IFA in the setting of programming languages [1] is steadily veering towards the adoption of dynamic techniques [2,3,4,5,6,7,8,9]. There are numerous reasons for this among which we can mention the following. First they are attractive from the perspective of scripting languages for the web such as JavaScript which are complex subjects of study for static-based techniques. Second, they allow dealing with inherently run-time issues such as dynamic object creation and `eval` run-time code evaluation mechanism. Last but not least, recent work has suggested that a mix of both static and dynamic flavors of IFA will probably strike the balance between correct, usable and scalable tools in practice.

Language-based secure IFA is achieved by assigning variables a security level such as public or secret and then determining whether those that are labeled as secret affect the contents of public ones during execution. This security property is formalised as *noninterference*. In this paper, we are concerned in particular with *termination-insensitive noninterference*[1,10]: starting with two identical run-time states that only differ in the contents of secret variables, the final states attained after any given pair of terminating runs differ at most in the contents of the secret variables. Thus in this paper we ignore covert channels.

G. Barthe, A. Datta, and S. Etalle (Eds.): FAST 2011, LNCS 7140, pp. 55–69, 2012.

IFA Monitors. Dynamic IFA monitors track the security level of data during execution. If the level of the data contained in a variable may vary during execution we speak of a *flow-sensitive* analysis [12]. Flow-sensitivity provides a more flexible setting than the flow-insensitive one when it comes to practical enforcement of security policies. Purely dynamic flow-sensitive monitors can leak information related to control flow [11]. Such monitors keep track of the security label of each variable and update these labels when variables are assigned. Information leak occurs essentially because these monitors cannot track traces that are not taken (such as branches that are not executed). Consider the example in Fig. 1 taken from [7] (the subscripts may be ignored for now). Assume that `sec` is initially labeled as secret. The monitor labels variables `tmp` and `pub` as public (since constants are considered public values) after executing the first two assignments. If `sec` is nonzero, the label of `tmp` is updated to secret since the assignment in line 3 depends on the value of `sec`. The "then" branch of the second conditional is not executed. If `sec` is zero, then the "then" branch of the second conditional is executed. Either way, the value of `sec`, a secret variable, leaks to the returned value and the monitor is incapable of detecting it.

```
1 tmp := 1; pub := 1;
2 if_p1 sec then
3     tmp := 0;
4 if_p2 tmp then
5     pub := 0;
6 ret_p3 (pub)
```

Fig. 1. Monitor attack, from [11]

Purely dynamic flow-sensitive monitors must therefore be supplied with additional information in order to compensate for this deficiency. One option is to supply the monitor with information on the branches not taken. This is the approach taken for example in [11]. In the example of Fig. 1, when execution reaches the conditional in line 4, although the "then" branch is not taken the label of `pub` would be updated to secret since this variable would have been written in the branch that was not taken and that branch depends on a secret variable. In order to avoid the need for performing static analysis [13] proposed the *no-sensitive upgrade* scheme where execution gets stuck on attempting to assign a public variable in a secret context. Returning to our example, when `sec` is nonzero and execution reaches the assignment in line 3, it would get stuck. A minor variant of that scheme is the *permissive upgrade* [14] scheme where, although assignment of public variables in a secret contexts is allowed, branching on expressions that depend on such variables is disallowed. In our example, when `sec` is nonzero and execution reaches the assignment in line 3, it would be allowed. However, execution would get stuck at line 4. As stated in [15], not only can these schemes reject secure programs, but also their practical applicability is yet to be determined.

Dynamic Dependency Tracking. An alternative to supplying a monitor that is flow-sensitive with either static information or resorting to the *no-sensitive upgrade* or *permissive upgrade* schemes is *dependency analysis* [5]. Shroff et al. introduce a run-time IFA monitor that assigns program points to branches and maintains a cache of dependencies of *indirect flows* towards program points and

a cache of *direct flows* towards program points. These caches are called κ and δ, respectively. The former is persistent over successive runs. Indeed, when execution takes a branch which has hitherto been unexplored, the monitor collects information associated with it and adds it to the current indirect dependencies. Thus, although an initial run may not spot an insecure flow, it will eventually be spotted in subsequent runs.

In order to illustrate this approach, we briefly revisit the example of Fig. 1 (further details are supplied in Sec. 2). We abbreviate the security level "secret" with the letter H and "public" with L, as is standard. Values in this setting are tagged with both a set of dependencies (set of program points p, p_i, etc.) and a security level. When the level is not important but the dependency is, we annotate the value just with the dependency: e.g. 0^p (in our example dependencies are singletons, hence we write p rather that $\{p\}$). Likewise, when it is the security level that is relevant we write for e.g. 0^L or 0^H. After initialization of the variables and their security levels, the guard in line 2 is checked. Here two operations take place. First the level of program point $p1$ is set to H reflecting a direct dependency of $p1$ with `sec`. This is stored in δ, the cache of direct dependencies. The body of the condition is executed (since the guard is true) and `tmp` is updated to 0^{p1}, indicating that the assigned value depends on the guard in $p1$. When the guard from the fourth line is evaluated, in κ (the cache of indirect dependencies, which is initially empty) the system stores that $p2$ depends on $p1$ (written $p_2 \mapsto p_1$), since the value of the variable involved in the condition depends on $p1$. At this point `pub` has the same value, namely 1, as `sec`, and hence leaks this fact. The key of the technique is to retain κ for future runs. Suppose that in a successive run `sec` is 0^H. The condition from line 2 is evaluated and the direct dependency $p1 \mapsto H$ is registered in δ. The third line is skipped and the condition pointed by $p2$ is checked. This condition refers to `tmp` whose value is 1^L. The body in line 5 is executed and `pub` is updated with 0^{p2}. At this point, it is possible to detect that `pub` depends on H as follows: variable `pub` depends on $p2$ (using the cache κ); $p2$ depends on $p1$; and the level of the latter program point is H according to the direct dependency cache. Table 1 summarizes both runs as explained above.

Inlining Monitors. An alternative to implementing a monitor as part of a custom virtual machine or modifying the interpreter [16,17] is to resort to *inlining* [2,15,18,19]. The main advantage behind this option is that no modification of the host run-time environment is needed, hence achieving a greater degree of portability. This is particularly important in web applications. Also, such an inlining can take place either at the browser level or at the proxy level, thus allowing dedicated hardware to inline system wide. Magazinius et al. [19] introduce the notion of *on-the-fly* inlining. The monitor in charge of enforcing the security policy uses a function *trans* to inline a monitored code. This function is also available at run-time and can be used to transform code only known immediately before its execution. The best example of this dynamic source is the `eval` primitive.

Table 1. Dependency tracking on two runs of Fig. 1

	First run					Second run				
line	1	2	3	4	6	1	2	4	5	6
`sec`	1^H	1^H	1^H	1^H	1^H	0^H	0^H	0^H	0^H	0^H
`tmp`	1^L	1^L	0^{p1}	0^{p1}	0^{p1}	1^L	1^L	1^L	1^L	1^L
`pub`	1^L	1^L	1^L	1^L	1^L	1^L	1^L	1^L	0^{p2}	0^{p2}
$p1$		H	H	H	H		H	H	H	H
$p2$				L	L			L	L	L
$p3$					L					L
ret					1^{p3}					0^{p3}
κ				$p1$ $\uparrow$ $p2$	$p1$ $\uparrow$ $p2$	$p1$ $\uparrow$ $p2$	$p1$ $\uparrow$ $p2$	$p1$ $\uparrow$ $p2$	$p1$ $\uparrow$ $p2$	$p1$ $\uparrow$ $p2$ $\uparrow$ $p3$

Contribution. This paper takes the first steps in *inlining* the dependency analysis [5] as a viable alternative to supplying a flow-sensitive monitor with either static information or resorting to the *no-sensitive upgrade* or *permissive upgrade* schemes. Given that we aim at applying our monitor to JavaScript, we incorporate `eval` into our analysis. Since the code evaluated by `eval` is generated at run-time and, at the same time, the dependency tracking technique requires that program points be persisted, we resort to hashing to associate program points to dynamically generated code. We define and prove correct an on-the-fly inlining transformation, in the style of [19], of a security monitor which is based on dependency analysis that incorporates these extensions.

Paper Structure. Sec. 2 recasts the theory of [5] originally developed for a lambda calculus with references to a simple imperative language. Sec. 3 briefly describes the target language of the inlining transformation and defines the transformation itself. Sec. 4 extends the transformation to `eval`. The properties of the transformation are developed in Sec. 5. Finally, we present conclusions and possible lines of additional work. A prototype in Python is available at `http://tpi.blog.unq.edu.ar/~ebonelli/inlining.tar.gz`.

2 Dependency Analysis for a Simple Imperative Language

We adapt the dependency analysis framework of Shroff et al. [5] to a simple imperative language $\mathcal{W}^{deps}$ prior to considering an inlining transformation for it. Its syntax is given in Fig. 2. There are two main syntactic categories, *expressions* and *commands*. An expression is either a variable, a labeled value, a binary expression, an application (of a user-defined function to an argument expression) or a case expression. A *labeled value* is a tuple consisting of a value (an integer or

$$
\begin{array}{rll}
P, \pi & ::= & \{\overline{p}\} & \text{(set of ppids, program counter)} \\
v & ::= & i \mid s & \text{(value)} \\
\sigma & ::= & \langle v, P, L \rangle & \text{(labeled value)} \\
e & ::= & x \mid \sigma \mid e \oplus e \mid f(e) \mid \ \texttt{case}\ e\ \texttt{of}\ (e:e)^+ & \text{(expression)} \\
c & ::= & \texttt{skip} \mid x := e \mid \texttt{let}\, x = e\, \texttt{in}\, c \mid c; c \mid \texttt{while}_p\, e\, \texttt{do}\, c & \text{(command)} \\
 & \mid & \texttt{if}_p\, e\, \texttt{then}\, c\, \texttt{else}\, c \mid \texttt{ret}_p(e) \mid \mathit{stop} & \\
E & ::= & \emptyset \mid f(x) \doteq e; E & \text{(expr. environment)} \\
\mu & ::= & \{\overline{x \mapsto \sigma}\} & \text{(memory)} \\
\kappa & ::= & \{\overline{p \mapsto P}\} & \text{(cache of dependencies)} \\
\delta & ::= & \{\overline{p \mapsto L}\} & \text{(cache of direct flows)}
\end{array}
$$

Fig. 2. Syntax of $\mathcal{W}^{deps}$

a string), a set of program points and a security level. We assume a set of *program points* $p_1, p_2, \ldots$. *Security levels* are taken from a lattice $(\mathcal{L}, \sqsubseteq)$. We write $\sqcup$ for the supremum. Commands are standard. For technical purposes, it is convenient to assume that the program to be executed ends in a return command `ret`, and that moreover this is the unique occurrence of `ret` in the program. Note however that this assumption may be dropped at the expense of slightly complicating the statement of *information leak* (Def. 1) and *delayed leak detection* (Prop. 1). The `while`, `if` and `ret` commands are sub-scripted with a program point.

The operational semantics of $\mathcal{W}^{deps}$ is defined in terms of a binary relation over *configurations*, tuples of the form $\langle E, \kappa, \delta, \pi, \mu, c \rangle$ where E is an *expression environment*, κ is a *cache of indirect flows*, δ is a *cache of direct flows*, π is the *program counter* (a set of program points), μ is a (partial) function from variables to labeled values and c is the current command. We use $\mathcal{D}, \mathcal{D}_i, etc$ for configurations. We write $\mu[x \mapsto \sigma]$ for the memory that behaves as μ except on x to which it associates σ. Also, $\mu \setminus x$ undefines μ on x. The domain of μ includes a special variable *ret* that holds the return value. The expression environment declares all available user-defined functions. We omit writing it in configurations and assume it is implicitly present. *Expression evaluation* is introduced in terms of *closed expression evaluation* and then *(open) expression evaluation*. *Closed expression evaluation* is defined as follows,

$$
\begin{array}{rcl}
\mathcal{I}(\langle v, P, L \rangle) & \stackrel{def}{=} & \langle v, P, L \rangle \\
\mathcal{I}(f(e)) & \stackrel{def}{=} & \hat{f}(\mathcal{I}(e)) \\
\mathcal{I}(\texttt{case}\ e\ \texttt{of}\ \boldsymbol{e : e'}) & \stackrel{def}{=} & \mathit{c\hat{a}se}\ \mathcal{I}(e)\ \mathit{of}\ \boldsymbol{e'_i : e'} \\
\mathcal{I}(e_1 \oplus e_2) & \stackrel{def}{=} & \mathcal{I}(e_1) \hat{\oplus} \mathcal{I}(e_2)
\end{array}
$$

where we assume $\hat{f}(\langle v, P, L \rangle) \stackrel{def}{=} \mathcal{I}(e[x := \langle v, P, L \rangle])$, if $f(x) \doteq e \in E$; $\mathit{c\hat{a}se}\ \langle u, P, L \rangle\ \mathit{of}\ \boldsymbol{e : e'} \stackrel{def}{=} \langle v, P \cup P', L \sqcup L' \rangle$ if u matches[1] e_i with substitution σ and $\mathcal{I}(\sigma e'_i) = \langle v, P', L' \rangle$; and $\langle i_1, P_1, L_1 \rangle \hat{\oplus} \langle i_2, P_2, L_2 \rangle \stackrel{def}{=} \langle i_1 \oplus i_2, P_1 \cup P_2, L_1 \sqcup L_2 \rangle$.

[1] Here we mean the standard notion of matching of a closed term e_1 against an algebraic pattern e_2; if successful, it produces a substitution σ for the variables of e_2 s.t. $\sigma(e_2) = e_1$.

We assume that in a case-expression exactly one branch applies. Moreover, we leave it to the user to guarantee that user-defined functions are terminating.

Given a memory μ, the *variable replacement* function, also written μ, applies to expressions: it traverses expressions replacing variables by their values. It is defined only if the free variables of its argument are in the domain of μ. Finally, *open expression evaluation* is defined as $\mathcal{I} \circ \mu$, the composition of $\mathcal{I}$ and μ, and abbreviated $\hat{\mu}$.

The *reduction judgement* $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ states that the former configuration *reduces* to the latter. This judgement is defined by means of the *reduction schemes* of Fig. 3. It is a mixed-step semantics in the sense that it mixes both small and big-step semantics. Thus $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ may be read as $\mathcal{D}_2$ may be obtained from $\mathcal{D}_1$ in some number of small reduction steps. We write $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ for the n-fold composition of $\rightarrowtail$. Rule Skip is straightforward; *stop* is a run-time command to indicate the end of execution. The Let scheme is standard; we resort to $[x := e]$ for capture avoiding substitution of all occurrences of the free variable x by e. The Assign scheme updates memory μ by associating x with the labeled value of e, augmenting the indirect dependencies with the program counter π. We omit the description of While-T and While-F and describe the schemes for the conditional (which are similar). If the condition is true (the reduction scheme when the condition is false, namely If-F, is identical except that it reduces c_2, hence it is omitted), then before executing the corresponding branch the configuration is updated. First the program counter is updated to include the program point p. A new dependency is added to the cache of indirect dependencies for p, namely $\pi \cup P$, indicating that there is an *indirect* flow from the current security context under which the conditional is being reduced and the condition e (via its dependencies). The union operator $\kappa \uplus \kappa'$ is defined as κ'' iff κ'' is the smallest cache such that $\kappa, \kappa' \leq \kappa''$. Here the ordering relation on caches is defined as $\kappa \leq \kappa'$ iff $\forall p \in dom(\kappa).\kappa(p) \subseteq \kappa'(p)$. Finally, the security level L of the condition is recorded in δ', reflecting the *direct* dependency of the branch on e. The scheme for `ret` updates the cache of indirect dependencies indicating that there is an *indirect* flow from the program counter and e (via its dependencies) towards the value that is returned. Finally, we note that $\langle \kappa, \delta, \pi, \mu, c\rangle \rightarrowtail \langle \kappa', \delta', \pi', \mu', c'\rangle$ implies $\kappa \leq \kappa'$ and $\pi' = \pi$.

2.1 Properties

Delayed leak detection (Prop. 1), the main property that the monitor enjoys, is presented in this section. Before doing so however, we require some definitions. The transitive closure of cache look-up is defined as $\kappa(p) \overset{def}{=} P \cup \kappa(P)^+$, where $\kappa(p) = P$. Suppose $P = \{p_1, \ldots, p_k\}$. Then $\kappa(P) \overset{def}{=} \bigcup_{i\in 1..k} k(p_i)$ and $\kappa(P)^+ \overset{def}{=} \bigcup_{i\in 1..k} k(p_i)^+$. We define $\mathsf{secLevel}^{\kappa,\delta} P \overset{def}{=} \delta(P \cup \kappa(P)^+)$, the join of all security levels associated to the transitive closure of P according to the direct dependencies recorded in δ. We write $\mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_{high}\rangle}]$ for $\mu[x_1 \mapsto \langle v_1, \emptyset, L_{high}\rangle] \ldots [x_k \mapsto \langle v_k, \emptyset, L_{high}\rangle]$. We fix L_{low} and L_{high} to be any two distinct levels. A terminating run leaks information via its return value, if

$$\frac{}{\langle \kappa, \delta, \pi, \mu, \texttt{skip}\rangle \rightarrowtail \langle \kappa, \delta, \pi, \mu, \mathit{stop}\rangle}\ \textsc{Skip}$$

$$\frac{\langle \kappa, \delta, \pi, \mu[z \mapsto \hat{\mu}(e)], c[x := z]\rangle \overset{n}{\rightarrowtail} \langle \kappa', \delta', \pi, \mu', \mathit{stop}\rangle \quad z \text{ fresh}}{\langle \kappa, \delta, \pi, \mu, \texttt{let}\ x = e\ \texttt{in}\ c\rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu' \setminus z, \mathit{stop}\rangle}\ \textsc{Let}$$

$$\frac{\langle \kappa, \delta, \pi, \mu, c_1\rangle \overset{n}{\rightarrowtail} \langle \kappa', \delta', \pi, \mu', \mathit{stop}\rangle}{\langle \kappa, \delta, \pi, \mu, c_1; c_2\rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu', c_2\rangle}\ \textsc{Seq}$$

$$\frac{\hat{\mu}(e) = \langle v, P, L\rangle \quad \mu' = \mu[x \mapsto \langle v, P \cup \pi, L\rangle]}{\langle \kappa, \delta, \pi, \mu, x := e\rangle \rightarrowtail \langle \kappa, \delta, \pi, \mu', \mathit{stop}\rangle}\ \textsc{Assign}$$

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle i, P, L\rangle \quad i \neq 0 \quad \pi' = \pi \cup \{p\} \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \langle \kappa', \delta', \pi', \mu, c\rangle \overset{n}{\rightarrowtail} \langle \kappa'', \delta'', \pi', \mu'', \mathit{stop}\rangle\end{array}}{\langle \kappa, \delta, \pi, \mu, \texttt{while}_p\, e\, \texttt{do}\, c\rangle \rightarrowtail \langle \kappa'', \delta'', \pi, \mu', \texttt{while}_p\, e\, \texttt{do}\, c\rangle}\ \textsc{While-T}$$

$$\frac{\hat{\mu}(e) = \langle 0, P, L\rangle \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\}}{\langle \kappa, \delta, \pi, \mu, \texttt{while}_p\, e\, \texttt{do}\, c\rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu, \mathit{stop}\rangle}\ \textsc{While-F}$$

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle i, P, L\rangle \quad i \neq 0 \quad \pi' = \pi \cup \{p\} \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \langle \kappa', \delta', \pi', \mu, c_1\rangle \overset{n}{\rightarrowtail} \langle \kappa'', \delta'', \pi', \mu', \mathit{stop}\rangle\end{array}}{\langle \kappa, \delta, \pi, \mu, \texttt{if}_p e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle \rightarrowtail \langle \kappa'', \delta'', \pi, \mu', \mathit{stop}\rangle}\ \textsc{If-T}$$

$$\frac{\hat{\mu}(e) = \langle v, P, L\rangle \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\}}{\langle \kappa, \delta, \pi, \mu, \texttt{ret}_p(e)\rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu[\mathit{ret} \mapsto \langle v, P \cup \pi, L\rangle], \mathit{stop}\rangle}\ \textsc{Ret}$$

Fig. 3. Mixed-step semantics for $\mathcal{W}^{deps}$

this return value is visible to an attacker as determined by the schemes in Fig. 3 and there is another run of the same command, whose initial memory differs only in secret values w.r.t. that of the first run, that produces a different return value. Moreover, this second run has the final cache of indirect dependencies of the first run (κ_1) as its *initial* cache of indirect dependencies.

Definition 1 (Information Leak [5]). *Let* $\mu_0 \overset{def}{=} \mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_{high}\rangle}]$ *for some memory* μ*. A run* $\langle \kappa_0, \delta_0, \pi, \mu_0, c\rangle \overset{n_1}{\rightarrowtail} \langle \kappa_1, \delta_1, \pi, \mu_1, \mathit{stop}\rangle$ leaks information w.r.t. security level L_{low}*, with* $L_{high} \not\sqsubseteq L_{low}$ *iff*

1. $\mu_1(\mathit{ret}) = \langle i_1, P_1, L_1\rangle$*;*
2. $(\mathsf{secLevel}^{\kappa_1,\delta_1} P_1) \sqcup L_1 \sqsubseteq L_{low}$*; and*
3. *there exists* k *labeled values* $\overline{\langle v'_k, \emptyset, L_{high}\rangle}$ *s.t.* $\mu'_0 = \mu[\overline{x_k \mapsto \langle v'_k, \emptyset, L_{high}\rangle}]$ *and* $\langle \kappa_1, \delta_0, \pi, \mu'_0, c\rangle \overset{n_2}{\rightarrowtail} \langle \kappa_2, \delta_2, \pi, \mu_2, \mathit{stop}\rangle$ *and* $\mu_2(\mathit{ret}) = \langle i_2, P_2, L_2\rangle$ *with* $i_1 \neq i_2$.

Delayed leak detection is proved in [5] in the setting of a higher-order functional language and may be adapted to our simple imperative language.

Proposition 1. *If*

- $\mu_0 = \mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_k\rangle}]$;
- *the run* $\langle \kappa_0, \delta_0, \pi, \mu_0, c\rangle \overset{n_1}{\rightarrowtail} \langle \kappa_1, \delta_1, \pi, \mu_1, \mathit{stop}\rangle$ *leaks information w.r.t. security level* L_{low}*; and*
- $\mu_1(\mathit{ret}) = \langle i_1, P_1, L_1\rangle$

then there exists $\overline{\langle v'_k, \emptyset, L'_k\rangle}$ *s.t.*

- $\mu'_0 = \mu[\overline{x_k \mapsto \langle v'_k, \emptyset, L'_k\rangle}]$;
- $\langle \kappa_1, \delta_0, \pi, \mu'_0, c\rangle \overset{n_2}{\rightarrowtail} \langle \kappa_2, \delta_2, \pi, \mu_2, \mathit{stop}\rangle$*; and*
- $\mathsf{secLevel}^{\kappa_2,\delta_1} P_1 \not\sqsubseteq L_{low}$.

The labeled values $\overline{\langle v'_k, \emptyset, L'_k\rangle}$ may be either public or secret since, if the first run leaks information, then appropriate input values of any required level must be supplied in order for the second run to gather the necessary dependencies that allow it to detect the leak.

3 Inlining the Dependency Analysis

The inlining transformation *trans* inserts code that allows dependencies to be tracked during execution. The target of the transformation is a simple imperative language we call $\mathcal{W}$ whose syntax is defined as follows:

$$
\begin{array}{rll}
v & ::= i \mid s \mid P \mid L & \text{(value)} \\
e & ::= x \mid v \mid e \oplus e \mid f(e) \mid \ \texttt{case}\ e\ \texttt{of}\ (e:e)^+ & \text{(expression)} \\
c & ::= \texttt{skip} \mid c;c \mid \texttt{let}\, x = e\, \texttt{in}\, c \mid x := e \mid \texttt{while}\, e\, \texttt{do}\, c \mid & \text{(command)} \\
 & \quad \mid \texttt{if}\, e\, \texttt{then}\, c\, \texttt{else}\, c \mid \texttt{ret}(e) \mid \mathit{stop} & \\
M & ::= \{\overline{x \mapsto v}\} & \text{(memory)}
\end{array}
$$

In contrast to $\mathcal{W}^{deps}$, it operates on standard, unlabeled values and also includes sets of program points and security levels as values, since they will be manipulated by the inlined monitor. Moreover, branches, loops and return commands are no longer decorated with program points. *Expression evaluation* is defined similarly to $\mathcal{W}^{deps}$. A *$\mathcal{W}$-(run-time) configuration* is an expression of the form $\langle E, M, c\rangle$ (as usual E shall be dropped for the sake of readability) denoted with letters $\mathcal{C}, \mathcal{C}_i$, etc. The small-step[2] semantics of $\mathcal{W}$ commands is standard and hence omitted. We write $\mathcal{C} \to \mathcal{C}'$ when $\mathcal{C}'$ is obtained from $\mathcal{C}$ via a reduction step. The transformation *trans* is a user-defined function that resides in E; when applied to a string it produces a new one. We use double-quotes for string constants and $+\!\!+$ for string concatenation.

We now describe the inlining transformation depicted in Fig. 4 and Fig. 5. The inlining of `skip` is immediate. Regarding assignment $x := e$, the transformation

[2] Hence not mixed-step but rather the standard notion.

```
1  trans(y) =
2   case y of
3      "skip": "skip"
4      "x:=e":
5           "x_L:= lev(" ++ vars("e") ++ ");" ++
6           "x_P:= dep(" ++ vars("e") ++ ") | pc;" ++
7           "x :=e"
8      "let x=e in c":
9           "let x=e in " ++
10            "x_L:= lev(" ++ vars("e") ++ ");" ++
11            "x_P:= dep(" ++ vars("e") ++ ") | pc;" ++
12            trans(c)
13     "c_1;c_2":
14          trans(c_1) ++ ";" ++ trans(c_2)
15     # continued below
```

Fig. 4. Inlining transformation (1/2)

introduces two shadow variables x_P and x_L. The former is for tracking the indirect dependencies of x while the latter is for tracking its security level. As may be perceived from the inlining of assignment, the transformation *trans* is in fact defined together with three other user-defined functions, namely *vars*, *lev* and *dep*. The first extracts the variables in a string returning a new string listing the comma-separated variables. Eg. $vars("x \oplus f(2 \oplus y)")$ would return, after evaluation, the string "x, y". The second user-defined function computes the least upper bound of the security levels of the variables in a string and the last computes the union of the implicit dependencies of the variables in a string. The level of e and its indirect dependencies are registered in x_L and x_P, respectively. In the case of x_P, the current program counter is included by means of the variable *pc*. The binary operator `|` denotes the union between sets. In contrast to $vars("e")$, which is computed at inlining time, *lev* and *dep* are computed when the inlined code is executed. We close the description of the inlining of assignment by noting that the transformed code adopts *flow-sensitivity* in the sense that the security level of the values stored in variables may vary during execution. It should also be noted that rather than resort to the *no sensitive upgrade* discipline of Austin and Flanagan [13] to avoid the attack of Fig. 1 (which is also adopted by [19] in their inlining transformation), the dependency monitor silently tracks dependencies without getting stuck.

The let construct is similar to assignment but also resorts to the let construct of $\mathcal{W}$. Here we incur in an abuse of notation since in practice we expect x_L and x_P to be implemented in terms of dictionaries $L[x]$ and $P[x]$. Hence we assume that the declared variable x also binds the x in x_L and x_P. The inlining of command composition is simply the inlining of each command. In the case of `while` (Fig. 5) first we have to update the current indirect dependencies cache and the cache of direct flows (lines 3 and 4, respectively). This is because evaluation of e will take place at least once in order to determine whether program execution skips

```
1  # continued from above
2  "while_p e do c":
3       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
4       "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
5       "while e do " ++
6           "(let pc'= pc in " ++
7             "pc := pc | {p};" ++
8              trans(c) ++
9              "pc := pc';" ++
10             "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
11             "d_p := d_p | lev(" ++ vars("e") ++ "));"
12 "if_p e then c_1 else c_2":
13       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
14       "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
15       "let pc'= pc in " ++
16        "pc := pc | {p};" ++
17         "if e then " ++ trans(c_1) ++ "else" ++ trans(c_2)++ ";" ++
18         "pc := pc'"
19 "ret_p (e)":
20        "k_p := k_p | dep(" ++ vars("e") ++ texttt") | pc;" ++
21        "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
22         "ret (e)"
```

Fig. 5. Inlining transformation (2/2)

the body of the while-loop or enters it. For that purpose we assume that we have at our disposal global variables k_p and d_p, for each program point p in the command to inline. Once inside the body, a copy of the program counter is stored in pc' and then the program counter is updated (line 7) with the program point of the condition of the `while`. Upon completing the execution of $trans(c)$, it is restored and then the dependencies are updated reflecting that a new evaluation of e takes place. The clause for the conditional is similar to the one for `while`. The clause for `ret` follows a similar description.

4 Incorporating `eval`

This section considers the extension of $\mathcal{W}^{deps}$ with the command `eval`(e). Many modern languages, including JavaScript, perform dynamic code evaluation. IFA studies have recently begun including it [9,19,20].

The argument of `eval` is an expression that denotes a string that parses to a program and is generated at run-time. Therefore its set of program points may vary. Since the monitor must persist the cache of indirect flows across different runs, we introduce a new element to $\mathcal{W}^{deps}$-configurations, namely a family of caches indexed by the codomain of a hash function: $\mathcal{K}$ is a mapping from the hash of the source code to a cache of indirect flows (i.e. $\mathcal{K} ::= \{\overline{h \mapsto \kappa}\}$ where h are elements of the codomain of the hash function). $\mathcal{W}^{deps}$-configurations thus

```
1  "eval_p(e)":
2    "let pc' = pc in " ++
3       "pc := pc | {p}" ++
4       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc'" ++
5       "d_p := d_p | lev(" ++ vars("e") ++ ")" ++
6       "let h = hash(e) in " ++
7          "k := k | K_h;" ++
8          "eval(trans(e));" ++
9          "K_h := K_h | depsIn(k, e);" ++
10      "pc := pc'"
```

Fig. 6. Inlining of $\texttt{eval}_p(e)$

take the new form $\langle \mathcal{K}, \kappa, \delta, \pi, \mu, c \rangle$. The reduction schemes of Fig. 3 are extended by (inductively) dragging along the new component; the following new reduction scheme, EVAL, will be in charge of updating it. A quick word on notation before proceeding: we write $\mathcal{K}(h)$ for the cache of indirect dependencies of s, where s is a string that parses to a command and $\mathsf{hash}(s) = h$. Also, given a cache κ and a command c, the expression $\kappa|_c$ is defined as follows (where $\mathsf{programPoints}(c)$ is the set of program points in c): $\kappa|_c \stackrel{def}{=} \{p \mapsto P \mid p \in \mathsf{programPoints}(c) \wedge \kappa(p) = P\}$. The EVAL reduction scheme is as follows:

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle s, P, L \rangle \quad \pi' = \pi \cup \{p\} \quad h = \mathsf{hash}(s) \\ \kappa' = \kappa \uplus \mathcal{K}(h) \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\} \\ \langle \mathcal{K}, \kappa', \delta', \pi', \mu, \mathsf{parse}(s) \rangle \stackrel{n}{\rightarrowtail} \langle \mathcal{K}', \kappa'', \delta'', \pi', \mu'', stop \rangle\end{array}}{\langle \mathcal{K}, \kappa, \delta, \pi, \mu, \texttt{eval}_p(e) \rangle \rightarrowtail \langle \mathcal{K}'[h \mapsto \mathcal{K}'(h) \uplus \kappa''|_{\mathsf{parse}(s)}], \kappa'', \delta'', \pi, \mu'', stop \rangle} \text{ EVAL}$$

This reduction scheme looks up the cache for the hash of s (that is $\mathcal{K}(h)$) and then adds it to the current indirect cache. Also added to this cache is the dependency of the code to be evaluated on the level of the context and the dependencies of the expression e itself. The resulting cache is called κ'. After reduction, $\mathcal{K}'$ is updated with any new dependencies that may have arised (recursively[3]) for s (written $\mathcal{K}'(h)$ above) together with the set of program points affected to parse(s) by the outermost (i.e. non-recursive) reduction (written $\kappa''|_{\mathsf{parse}(s)}$ above). EVAL may be inlined as indicated in Fig. 6 where $dep(k, e)$ represents the user-defined function that computes $\kappa|_c$. Note that c here is the code that results from parsing the value denoted by e.

This approach has a downside. When the attacker has enough control over e, she can manipulate it in order to always generate different hashes. This affects the accumulation of dependencies (the cache of indirect flows will never be augmented across different runs) and hence the effectiveness of the monitor in

[3] When $\mathsf{parse}(s)$ itself has an occurrence of `eval` whose argument evaluates to s.

```
1  "eval_p(e)":
2    "let pc' = pc in:" ++
3       "pc := pc | {p}" ++
4       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc'" ++
5       "d_p := d_p | lev(" ++ vars("e") ++ ")" ++
6       "let h = hash(e) in:" ++
7          "k := k | K_h;" ++
8          "eval(trans(e));" ++
9          "d_p := d_p | secLevel(k,d,dom(depsIn(k,e));" ++
10         "K_h := K_h | depsIn(k,e);" ++
11      "pc := pc'"
```

Fig. 8. External anchor for $\texttt{eval}_p(e)$

identifying leaks. Since the monitor can leak during early runs, this may not be desirable. The following code exemplifies this situation:

```
1 tmp := 1; pub := 1;
2 eval_p(x ++ " if_q1 sec then tmp := 0;
3             if_q2 tmp then pub := 0");
4 ret_q3 (pub)
```

The attacker may have control over `x`, affecting the hash and, therefore, avoid indirect dependencies from accumulating across different runs. Fig 7 represents a dependency chain of this code. The shaded box represents the `eval` context. Notice that q_1 and q_2 point to p because π had been extended with the latter. The edges a and b are created separately in two different runs, when `sec` is *1* or *0* respectively. The monitor should be able to capture the leak by accumulating both edges in κ, just like in the example in Fig. 1, because there is a path that connects q_3 with the high labeled q_1. But, since the attacker may manipulate the hash function output via the variable `x`, it is possible to avoid the accumulative effect in κ thus a and b will not exist simultaneously in any run.

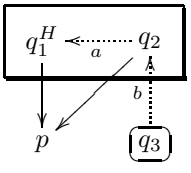

Fig. 7. Edges a and b are *both* needed to detect the leak in q_3

One approach to this situation is to allow the program point p in the $\texttt{eval}_p(e)$ command to absorb all program points in the code denoted by e. Consequently, if a high node is created in the `eval` context, p will be raised to *high* just after the execution of `eval`. The reduction scheme Eval would have to be replaced by Eval':

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle s, P, L\rangle \quad h = \mathsf{hash}(s) \quad \pi' = \pi \cup \{p\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \kappa' = \kappa \uplus \mathcal{K}(h) \uplus \{p \mapsto \pi \cup P\} \\ \delta''' = \delta''[p \mapsto \mathsf{secLevel}^{\kappa'',\delta''} dom(\kappa''|_{\mathsf{parse}(s)})] \\ \langle \mathcal{K}, \kappa', \delta', \pi', \mu, \mathsf{parse}(s)\rangle \overset{n}{\rightarrowtail} \langle \mathcal{K}', \kappa'', \delta'', \pi', \mu'', stop\rangle\end{array}}{\langle \mathcal{K}, \kappa, \delta, \pi, \mu, \mathtt{eval}_p(e)\rangle \rightarrowtail \langle \mathcal{K}'[h \mapsto \mathcal{K}'(h) \uplus \kappa''|_{\mathsf{parse}(s)}], \kappa'', \delta''', \pi, \mu'', stop\rangle} \text{ EVAL'}$$

Intuitively, every node associated to the program argument of `eval` passes on to p its level which hence works as an external anchor. In this way, if any node has the chance to be in the path of a leak, every low variable depending on them is considered dangerous. The new dependency chain for the above mentioned example is shown in Fig. 9, where the leak is detected. More precisely, when $\mathtt{eval}_p(e)$ concludes, δ'' is upgraded to $\mathsf{secLevel}^{\kappa,\delta} dom(\kappa''|_c)$ (where *dom* is the domain of the mapping). Since $q1$ is assigned level secret by δ'', this bumps the level of p to secret. The proposed inlining is given in Fig. 8. In this approach the `ret` statement should not be allowed inside the `eval`, since the bumping of the security level of p is produced *a posteriori* to the execution of the argument of `eval`.

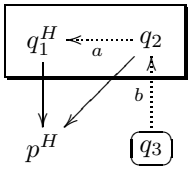

Fig. 9. Dependency chain with external anchor for $\mathtt{eval}_p(e)$

5 Properties of the Inlining Transformation

This section addresses the *correctness* of the inlined transformation. We show that the inlined transformation of a command c simulates the execution of the monitor. First we define what it means for a $\mathcal{W}$-configuration to *simulate* a $\mathcal{W}^{deps}$-configuration. We write $\mathsf{trans}(c)$ for the result of applying the *recursive function* determined by the code for *trans* to the argument "c" and then parsing the result. Two sample clauses of trans are: $\mathsf{trans}(c_1; c_2) \overset{def}{=} \mathsf{trans}(c_1); \mathsf{trans}(c_2)$ for command composition and $\mathsf{trans}(\mathtt{eval}(e)) \overset{def}{=} \mathtt{let}\ h = \ hash(e)\ \mathtt{in}\ (k := k \mid K_h;\ \mathtt{eval}(trans(e));\ K_h := K_h \mid depsIn(k, e))$ for `eval`. We also extend this definition with the clause: $\mathsf{trans}(stop) \overset{def}{=} stop$.

Definition 2. *A $\mathcal{W}$-configuration $\mathcal{C}$ simulates a $\mathcal{W}^{deps}$-configuration $\mathcal{D}$, written $\mathcal{D} \prec \mathcal{C}$, iff*

1. $\mathcal{D} = \langle \mathcal{K}, \kappa, \delta, \pi, \mu, c\rangle$;
2. $\mathcal{C} = \langle M, \mathsf{trans}(c)\rangle$;
3. $M(K) = \mathcal{K}$, $M(k) = \kappa$, $M(d) = \delta$, $M(pc) = \pi$; *and*
4. $\mu(x) = \langle M(x), M(x_P), M(x_L)\rangle$, *for all* $x \in dom(\mu)$.

In the expression '$M(K) = \mathcal{K}$' by abuse of notation we view $M(K)$ as a "dictionary" and therefore understand this expression as signifying that for all

$h \in dom(\mathcal{K})$, $M(K_h) = \mathcal{K}(h)$. Similar comments apply to $M(k) = \kappa$ and $M(d) = \delta$. In the case of $M(pc) = \pi$, both sets of program points are tested for equality.

The following correctness property is proved by induction on an appropriate notion of *depth* of the reduction sequence $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$.

Proposition 2. *If (1)* $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c \rangle$*; (2)* $\mathcal{C}_1 = \langle M_1, \mathsf{trans}(c) \rangle$*; (3)* $\mathcal{D}_1 \prec \mathcal{C}_1$*; and (4)* $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$*,* $n \geq 0$*; then there exists* $\mathcal{C}_2$ *s.t.* $\mathcal{C}_1 \twoheadrightarrow \mathcal{C}_2$ *and* $\mathcal{D}_2 \prec \mathcal{C}_2$*.*

$$\begin{array}{ccc} \mathcal{D}_1 & \prec & \mathcal{C}_1 \\ {\scriptstyle n}\big\downarrow & & \big\vdots\big\downarrow \\ \mathcal{D}_2 & \prec & \mathcal{C}_2 \end{array}$$

Remark 1. A converse result also holds: modulo the administrative commands inserted by trans, reduction from $\mathcal{C}_1$ originates from corresponding commands in c. This may be formalised by requiring the inlining transformation to insert a form of labeled `skip` command to signal the correspondence of inlined commands with their original counterparts (cf. Thm.2(b) in [15]).

6 Conclusions and Future Work

We recast the dependency analysis monitor of Shroff et al. [5] to a simple imperative language and propose a transformation for inlining this monitor on-the-fly. The purpose is to explore the viability of a completely dynamic inlined dependency analysis as an alternative to other run-time approaches that either require additional information from the source code (such as branches not taken [15]) or resort to rather restrictive mechanisms such as *no sensitive upgrade* [13] (where execution gets stuck on attempting to assign a public variable in a secret context) or *permissive upgrade* [14] (where, although assignment of public variables in a secret contexts is not allowed, branching on expressions that depend on such variables is disallowed).

This paper reports work in progress, hence we mention some of the lines we are currently following. First we would like to gain some experience with a prototype implementation of the inlined transformation as a means of foreseeing issues related to usability and scaling. Second, we are considering the inclusion of an output command and an analysis of how the notion of progress-sensitivity [9] adapts to the dependency tracking setting. Finally, inlining declassification mechanisms will surely prove crucial for any practical tool based on IFA.

Acknowledgements. To the referees for supplying helpful feedback.

References

1. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications

2. Venkatakrishnan, V.N., Xu, W., Duvarney, D.C., Sekar, R.: Provably correct runtime enforcement of non-interference properties. In: International Conference on Information and Communication Security, pp. 332–351 (2006)
3. Guernic, G.L., Banerjee, A., Jensen, T.P., Schmidt, D.A.: Automata-Based Confidentiality Monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)
4. Guernic, G.L.: Automaton-based confidentiality monitoring of concurrent programs. In: Computer Security Foundations Workshop, pp. 218–232 (2007)
5. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium, pp. 203–217. IEEE Computer Society, Washington, DC, USA (2007)
6. Mccamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 193–205 (2008)
7. Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research. In: Ershov. Memorial Conf., pp. 352–365 (2009)
8. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 113–124 (2009)
9. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Computer Security Foundations Workshop, pp. 43–59 (2009)
10. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4, 167–188
11. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium, CSF 2010, pp. 186–199. IEEE Computer Society, Washington, DC, USA (2010)
12. Hunt, S., Sands, D.: On flow-sensitive security types. In: Morrisett, J.G., Jones, S.L.P. (eds.) POPL, pp. 79–90. ACM (2006)
13. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. SIGPLAN Not. 44, 20–31 (2009)
14. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2010, pp. 3:1–3:12. ACM, New York (2010)
15. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: Computer Security Foundations Workshop, pp. 200–214 (2010)
16. Futoransky, A., Gutesman, E., Waissbein, A.: A dynamic technique for enhancing the security and privacy of web applications. In: Black Hat USA 2007 Briefings, August 1-2, Las Vegas, NV, USA (2007)
17. Dhawan, M., Ganapathy, V.: Analyzing information flow in javascript-based browser extensions. In: Annual Comp. Sec. App. Conference, pp. 382–391 (2009)
18. Erlingsson, U.: The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Department of Computer Science, Cornell University (2003) TR 2003-1916
19. Magazinius, J., Russo, R., Sabelfeld, A.: On-the-fly inlining of dynamic security monitors. In: Proc. IFIP International Information Security Conference (2010)
20. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for javascript. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 50–62 (2009)