On-the-fly Inlining of Dynamic Security Monitors

Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld

Dept. of Computer Science and Engineering, Chalmers University of Technology
412 96 Gothenburg, Sweden, Fax: +46 31 772 3663

Abstract. Language-based information-flow security considers programs that manipulate pieces of data at different sensitivity levels. Securing information flow in such programs remains an open challenge. Recently, considerable progress has been made on understanding dynamic monitoring for secure information flow. This paper presents a framework for inlining dynamic information-flow monitors. A novel feature of our framework is the ability to perform inlining on the fly. We consider a source language that includes dynamic code evaluation of strings whose content might not be known until runtime. To secure this construct, our inlining is done on the fly, at the string evaluation time, and, just like conventional offline inlining, requires no modification of the hosting runtime environment. We present a formalization for a simple language to show that the inlined code is secure: it satisfies a noninterference property. We also discuss practical considerations and preliminary experimental results.

1 Introduction

Language-based approach to security gains increasing popularity [16, 36, 45, 33, 19, 27, 7, 11] because it provides natural means for specifying and enforcing application and language-level security policies. Popular highlights include Java stack inspection [45], to enforce stack-based access control, Java bytecode verification [19], to verify bytecode type safety, and web language-based mechanisms such as Caja [27], ADsafe [7], and FBJS [11], to enforce sandboxing and separation by program transformation and language subsets.

Language-based information-flow security [33] considers programs that manipulate pieces of data at different sensitivity levels. For example, a web application might operate on sensitive (secret) data such as credit card numbers and health records and at the same time on insensitive (public) data such as third-party images and statistics. A key challenge is to secure *information flow* in such programs, i.e., to ensure that information does not flow from secret inputs to public outputs. There has been much progress on tracking information flow in languages of increasing complexity [33], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [28, 38, 39].

While the above tools are mostly based on static analysis, considerable progress has been also made on understanding monitoring for secure information flow [12, 43, 41, 18, 17, 37, 25, 34, 2, 1]. Mozilla's ongoing project FlowSafe [9] aims at empowering Firefox with runtime information-flow tracking, where dynamic information-flow reference monitoring [2, 3] lies at its core. The driving force for using the dynamic techniques is expressiveness: as more information is available at runtime, it is possible to

use it and accept secure runs of programs that might be otherwise rejected by static analysis.

Dynamic techniques are particularly appropriate to handle the dynamics of web applications. Modern web application provide a high degree of dynamism, responding to user-generated events such as mouse clicks and key strokes in a fine-grained fashion. One popular feature is auto-completion, where each new character provided by the user is communicated to the server so that the latter can supply an appropriate completion list. Features like this rely on scripts in browsers that are written in a reactive style. In addition, scripts often utilize dynamic code evaluation to provide even more dynamism: a given string is parsed and evaluated at runtime.

With a long-term motivation of securing a scripting language with dynamic code evaluation (such as JavaScript) in a browser environment without modifying the browser, the paper turns attention to the problem of *inlining* information security monitors. *Inlined reference monitors* [10] are realized by modifying the underlying application with inlined security checks. Inlining security checks are attractive because the resulting code requires no modification of the hosting runtime environment. In a web setting, we envisage that the kind of inlining transformation we develop can be performed by the server or a proxy so that the client environment does not have to be modified.

We present a framework for inlining dynamic information-flow monitors. For each variable in the source program, we deploy a *shadow variable* (auxiliary variable that is not visible to the source code) to keep track of its security level. Additionally, there is a special shadow variable *program counter* pc to keep track of the *security context*, i.e., the least upper bound of security levels of all guards for conditionals and loops that embody the current program point. The pc variable helps tracking *implicit flows* [8] via control-flow constructs.

A novel feature of our framework is the ability to perform inlining on the fly. We consider a source language that includes dynamic code evaluation (popular in languages as JavaScript, PHP, Perl, and Python). To secure dynamic code evaluation, our inlining is performed on the fly, at the string evaluation time, and, just like conventional offline inlining, requires no modification of the hosting runtime environment. The key element of the inlining is providing a small library packaged in the inlined code, which implements the actual inlining.

Our approach stays clear of the pitfalls of dynamic information-flow enforcement. Indeed, dynamic information-flow tracking is challenging because the source of insecurity may be the fact that a certain event has *not* occurred in a monitored execution. However, we draw on recent results on dynamic information-flow monitoring [34, 2] that show that security can be enforced purely dynamically. The key is that the execution of a program that attempts to leak a secret either explicitly, by an assignment, or implicitly, by public side effects once inside a conditional or loop that branches on secret data, is simply blocked by the monitor. This gives us a great advantage for treating dynamic code evaluation: the inlined monitor needs to perform no static analysis for the dynamically evaluated code.

We present a formalization for a simple language to show that the result of the inlining is secure: it satisfies the baseline policy of *termination-insensitive noninterference* [6, 13, 44, 33]: whenever two runs of a program that agree on the public part of the

$$P ::= (\mathbf{def}\ f(x) = e;)^*\ c \qquad e ::= s \mid \ell \mid x \mid e \oplus e \mid f(e) \mid \mathbf{case}\ e\ \mathbf{of}\ (e\ :\ e)^+$$
$$c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{let}\ x = e\ \texttt{in}\ c \mid \texttt{eval}(e) \mid \mathit{stop}$$

Fig. 1. Language

initial memory terminate, then the final memories must also agree on the public part. We conclude by discussing practical considerations and preliminary yet encouraging experimental results.

We remark that it is known that noninterference is not a safety property [26, 40]. *Precise* characterizations of what can be enforced by monitoring have been studied in the literature (e.g., [35, 14]), where noninterference is discussed as an example of a policy that cannot be enforced precisely by dynamic mechanisms. However, the focus of this paper is on enforcing *permissive yet safe approximations* of noninterference. The exact policies that are enforced might just as well be safety properties (or not), but, importantly, they must guarantee noninterference.

2 Inlining transformation

We present an inlining method for a simple imperative language with dynamic code evaluation. The inlined security analysis has a form of *flow sensitivity*, i.e., confidentiality levels of variables can be sometimes relabeled during program execution. Our source-to-source transformation injects purely dynamic security checks.

$$\textsc{Eval}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow s \qquad \mathit{parse}(s) = c}{\langle \texttt{eval}(e) \mid m, \Sigma\rangle \rightarrow \langle c \mid m, \Sigma\rangle}$$

Fig. 2. Semantics for dynamic code evaluation

Language Figure 1 presents a simple imperative language enriched with functions, local variables, and dynamic code evaluation. A program P is a possibly empty sequence of function definitions ($\mathbf{def}\ f(x) = e$) followed by a command c. Function bodies are restricted to using the formal parameter variable only ($\mathit{FV}(e) \subseteq \{x\}$, where $\mathit{FV}(e)$ denotes the free variables that occur in e). Expressions e consist of strings s, security levels ℓ, variables x, composite expressions $e \oplus e$ (where $\oplus$ is a binary operation), function calls $f(e)$, and non-empty case analysis ($\mathbf{case}\ e\ \mathbf{of}\ (e\ :\ e)^+$). We omit explanations for the standard imperative instructions appearing in the language [46]. Command *stop* signifies termination. Command $\texttt{let}\ x = e\ \texttt{in}\ c$ binds the value of e to the local *read-only* variable x, which is only visible in c. Command $\texttt{eval}(e)$ takes a string e, which represents a program, and runs it.

Semantics A program P, memory m, and *function environment* Σ form a *program configuration* $\langle P \mid m, \Sigma\rangle$. A memory m is a mapping from global program variables Vars to values Vals. A function environment Σ consists of a list of definitions of the form $\mathbf{def}\ f(x) = e$. A small semantic step has the form $\langle P \mid m, \Sigma\rangle \rightarrow \langle P' \mid m', \Sigma'\rangle$ and

$$
\begin{array}{l}
\mathit{trans}(y) = \\
\quad \textbf{case}\ y\ \textbf{of} \\
\quad\quad \text{"}\texttt{skip}\text{"}\ :\ \text{"}\texttt{skip}\text{"} \\
\quad\quad \text{"}x := e\text{"}\ :\ \text{"}\texttt{let}\ ex = \text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"})\ +\!\!+ \\
\quad\quad\quad\quad \text{"}\ \texttt{in if}\ pc \sqsubseteq x'\ \texttt{then}\ x' := \mathit{lev}(ex) \sqcup pc; x := e\ \texttt{else}\ \mathit{loop}\text{"} \\
\quad\quad \text{"}c_1; c_2\text{"}\ :\ \mathit{trans}(c_1) +\!\!+ \text{";"} +\!\!+ \mathit{trans}(c_2) \\
\quad\quad \text{"}\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\text{"}\ :\ \text{"}\texttt{let}\ ex = \text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"})\ +\!\!+ \\
\quad\quad\quad\quad\quad\quad \text{"}\ \texttt{in let}\ pc' = pc\ \texttt{in}\ pc := pc \sqcup \mathit{lev}(ex); \texttt{if}\ e\ \texttt{then}\ \{\text{"} +\!\!+ \\
\quad\quad\quad\quad\quad\quad \mathit{trans}(c_1) +\!\!+ \text{"}\}\ \texttt{else}\ \{\text{"} +\!\!+ \mathit{trans}(c_2) +\!\!+ \text{"}\};\ pc := pc'\text{"} \\
\quad\quad \text{"}\texttt{while}\ e\ \texttt{do}\ c\text{"}\ :\ \text{"}\texttt{let}\ ex = \text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"})\ +\!\!+ \\
\quad\quad\quad\quad\quad \text{"}\texttt{let}\ pc' = pc\ \texttt{in}\ pc := pc \sqcup \mathit{lev}(ex); \texttt{while}\ e\ \texttt{do}\ \{\text{"} +\!\!+ \mathit{trans}(c) \\
\quad\quad\quad\quad\quad +\!\!+ \text{"}\};\ pc := pc'\text{"} \\
\quad\quad \text{"}\texttt{let}\ x = e\ \texttt{in}\ c\text{"}\ :\ \text{"}\texttt{let}\ ex = \text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"})\ +\!\!+ \\
\quad\quad\quad\quad\quad\quad \text{"}\texttt{let}\ x' = \mathit{lev}(ex) \sqcup pc\ \texttt{in let}\ x = e\ \texttt{in}\ \text{"} +\!\!+ \mathit{trans}(c) \\
\quad\quad \text{"}\texttt{eval}(e)\text{"}\ :\ \text{"}\texttt{let}\ ex = \text{"} +\!\!+\ \mathit{vars}(\text{"}e\text{"})\ +\!\!+ \\
\quad\quad\quad\quad\quad \text{"}\texttt{let}\ pc' = pc\ \texttt{in}\ pc := pc \sqcup \mathit{lev}(ex); \texttt{eval}(\mathit{trans}(e)); pc := pc'\text{"}
\end{array}
$$

Fig. 3. Inlining transformation

$$
\frac{ex, pc, pc', x'_1, \ldots, x'_n \in \mathit{Fresh}(c)}{\begin{array}{c}\Gamma \vdash \textbf{def}\ f_1(x) = e_1; \ldots; \textbf{def}\ f_k(x) = e_k; c \rightsquigarrow \textbf{def}\ f_1(x) = e_1; \ldots; \textbf{def}\ f_k(x) = e_k; \\ \textbf{def}\ \mathit{vars}(y) = \ldots; \textbf{def}\ \mathit{lev}(y) = \ldots; \textbf{def}\ \mathit{trans}(y) = \ldots; \\ pc := L; x'_1 := \Gamma(x_1); \ldots; x'_n := \Gamma(x_n); \texttt{eval}(\mathit{trans}(c))\end{array}}
$$

Fig. 4. Top-level transformation

possibly updates the command, memory, and function environment. The semantics for dynamic code evaluation is shown in Figure 2. Dynamic code evaluation occurs when expression e evaluates, under the current memory and function environment, to a string s ($\langle e \mid m, \Sigma\rangle \Downarrow s$), and that string is successfully parsed to a command ($\mathit{parse}(s) = c$). For simplicity, we assume that executions of programs get stuck when failing to parse. In a realistic programming language, however, failing to parse would result in a runtime error. Semantics rules for the other commands are standard [46] and thus we omit them.

Inlining transformation Figure 3 contains the transformation for inlining. At the core of the monitor is a combination of the *no sensitive upgrade* discipline by Austin and Flanagan [2] and a treatment of dynamic code evaluation from a flow-insensitive monitor by Askarov and Sabelfeld [1]. String constants are enclosed by double-quote characters (e.g., "`text`"). Operator $+\!\!+$ concatenates strings (e.g., "`conc`" $+\!\!+$ "`atenation`" results in "`concatenation`"). Since *trans* only works on strings that represent programs, we consider programs and strings as interchangeable terms. In order for the transformation to work, variables x' (for any global variable x), as well as ex, pc', and pc must not occur in the string received as argument. The selection of names for these variables must avoid collisions with the source program variables, which is particularly important in the presence of dynamic code evaluation. In an implementation, this can be accomplished by generating random variable names for auxiliary variables. We differ this discussion until Section 4.

Before explaining how the transformation works, we state our assumptions regarding the security model. For convenience, we only consider the security levels L (*low*) and H (*high*) as elements of a security lattice, where $L \sqsubseteq H$ and $H \not\sqsubseteq L$. Security levels L and H identify public and secret data, respectively. We assume that the attacker can only observe public data, i.e., data at security level L. The lattice join operator $\sqcup$ returns the least upper bound over two given levels.

We now explain our inlining technique in detail. Given the source code src (as a string) and a mapping Γ (called *security environment*) that maps global variables to security levels, the inlining of the program is performed by the top-level rule in Figure 4. The rule has the form $\Gamma \vdash src \rightsquigarrow trg$, where, under the initial security environment Γ, the source code src is transformed into the target code trg. This rule defines three auxiliary functions $vars(\cdot)$, $lev(\cdot)$, and $trans(\cdot)$ for extracting the variables in a given string, for computing the least upper bound on the security level of a given string, and for on-the-fly transformation of a given string, respectively. We discuss the definition of these functions below. The top-level rule also introduces an auxiliary *shadow* variable pc, setting it to L, and a shadow variable x' for each source program global variable x, setting it to the initial security level, as specified by the security environment Γ. This is done to keep track of the current security levels of the context and of the global variables (as detailed below). The shadow variables are *fresh*, i.e., their set is disjoint from the variables that may occur in the configuration during the execution of the source program. We denote by $x \in \mathit{Fresh}(c)$, whenever variable x never occurs in the configuration during the execution of program c. With these definitions in place, the inlined version of src is simply $\texttt{eval}(trans(src))$, which has the effect of on-the-fly application of function $trans$ to the string src at runtime.

On-the-fly inlining We now describe the definition of function $trans(y)$ according to the cases on y. Since functions are side-effect free, the inlining of function declarations is straightforward: they are simply propagated to the result of the transformation. The inlining of command `skip` requires no action. As foreshadowed above, special shadow variable pc is used to keep track of the *security context*, i.e., the join of security levels of all guards for conditionals and loops that embody the current program point. The pc variable helps to detect implicit flows [8] of the form $\texttt{if}\ h\ \texttt{then}\ l := 0\ \texttt{else}\ l := 1$, where h and l are secret and public variables, respectively. In addition, Austin and Flanagan [2] use pc to restrict updates of variables' security levels: changes of variables' security levels are not allowed when the security context (pc) is set to H. This restriction helps to prevent attackers from turning flow sensitivity into a channel for laundering secrets [34, 31]. With this in mind, the inlining of $x := e$ demands that $pc \sqsubseteq x'$ before updating x'. In this manner, public variables ($x' = L$) cannot change their security level in high security contexts ($pc = H$). The security level of x is updated to the join of pc and the security level of variables appearing in e.

Function $lev(s)$ returns the least upper bound of the security levels of variables encountered in the string s. The formal specification of $lev(s)$ is given as $\sqcup_{x' \in FV(s)} x'$. Observe that directly calling $lev(e)$ does not necessarily returns the confidentiality level of e because the argument passed to lev is the result of evaluating e, which is a constant string. To illustrate this point, consider $w = "text"$, $w' = H$, and $e = w$. In this case, calling $lev(e)$ evaluates to $lev("text")$, which is defined to be L since $"text"$ does not

involve any variable. Clearly, setting $lev("text") = L$ is not acceptable since the string is formed from a secret variable. Instead, the transformation uses function $vars$ to create a string that involves all the variables appearing in an expression e ($vars("e")$). Observe that such string is not created at runtime, but when inlining commands. Function $vars$ returns a string with the shadow variables of the variables appearing in the argument string. For instance, assuming that $e = "text" +\!\!+\ w +\!\!+\ y$, we have that $vars("e") = "w' +\!\!+\ y'"$. Shadow variable x' is then properly updated to $pc \sqcup lev(ex)$, where $ex = vars("e")$. When $pc \not\sqsubseteq x'$, the transformation forces the program to diverge ($loop$) in order to preserve confidentiality. We define $loop$ as simply `while` 1 `do skip`. This is the only case in the monitor where the execution of the program might be interrupted due to a possible insecurity.

The inlining for sequential composition $c_1; c_2$ is the concatenation of transformed versions of c_1 and c_2. The inlining of `if` e `then` c_1 `else` c_2 produces a conditional, where their branches are transformed. The current value of pc is stored in the local variable pc', to be restored after the execution of the branch ($pc := pc'$). This manner to manipulate the pc, similar to traditional security type systems [44], avoids over-restrictive enforcement. The inlining of `while` e `do` c is similar to the one for conditionals. The inlining of `let` $x = e$ `in` c determines the security level of the new local variable x ($x' = lev(ex) \sqcup pc$) and transforms the body of the let ($trans(c)$).

The inlining of dynamic code evaluation is the most interesting feature of the transformation. Similarly to conditionals, the inlining of `eval`(e) saves the value of pc ($pc' = pc$) before updating it. Observe that the execution of commands depends on the confidentiality level of e as well as the current value of pc ($pc := pc \sqcup lev(ex)$). The value of pc is then restored by using pc' after the execution of `eval` ($pc := pc'$). In the transformed code, the transformation wires itself before executing calls to `eval` (`eval`$(trans(e))$). As a consequence, the transformation performs inlining on-the-fly, i.e., at the application time of the `eval`.

3 Formal results

This section presents the formal results. We prove the soundness of the transformation. Soundness shows that transformed programs respect a policy of *termination-insensitive noninterference* [6, 13, 44, 33]. Informally, the policy demands that whenever two runs of a program that agree on the public part of the initial memory terminate, then the final memories must also agree on the public part. Two memories m_1 and m_2 are *Γ-equal* (written $m_1 =_\Gamma m_2$) if they agree on the variables whole level is L according to Γ ($m_1 =_\Gamma m_2 \stackrel{\text{def}}{=} \forall x \in Vars.\, \Gamma(x) = L \Longrightarrow m_1(x) = m_2(x)$). The formal statement of noninterference is as follows.

Definition 1. *For initial and final security environments Γ and Γ', respectively, a program P satisfies* noninterference *(written $\models \{\Gamma\}\, c\, \{\Gamma'\}$) if and only if whenever $m_1 =_\Gamma m_2$, $\langle P \mid m_1, \cdot\rangle \longrightarrow^* \langle stop \mid m_1', \Sigma_1\rangle$, and $\langle P \mid m_2, \cdot\rangle \longrightarrow^* \langle stop \mid m_2', \Sigma_2\rangle$, then $m_1' =_{\Gamma'} m_2'$.*

We sketch three lemmas that lead to the proof of noninterference. We start by showing that the transformation only introduces shadow and local variables.

Lemma 1. *Given a string e and a variable x, if $x \notin FV(e)$, then $x \notin FV(trans(e))$.*

Similarly to Γ-equality, we define indistinguishability by a set of variables. Two memories are indistinguishable by a set of variables V if and only if the memories agree on the variables appearing in V. Formally, $m_1 =_V m_2 \stackrel{\text{def}}{=} \forall x \in V \cdot m_1(x) = m_2(x)$. Given a memory m, we define $L(m)$ to be the set of variables whose shadow variables are set to L. Formally, $L(m) = \{\ x \mid x \in m, x' \in m,\ x' = L\}$. In the following lemmas, let function environment Σ contain the definitions of $vars$, lev, and $trans$ as described in the previous section. The next lemma shows that there are no changes in the content and set of public variables when pc is set to H.

Lemma 2. *Given a memory m and a string s representing a command c such that $m(pc) = H$ and $\langle \texttt{eval}(trans(s)) \mid m, \Sigma\rangle \longrightarrow^* \langle stop \mid m', \Sigma'\rangle$, we have $L(m) = L(m')$ and $m =_{L(m)} m'$.*

The next lemma shows that neither the set of shadow variables set to L nor the contents of public variables depend on secrets. More specifically, the lemma establishes that two terminating runs of a transformed command c, under memories that agree on public data, always produce the same public results and set of shadow variables assigned to L.

Lemma 3. *Given memories m_1, m_2 and a string s representing a command c, if $L(m_1) = L(m_2)$, $m_1 =_{L(m_1)} m_2$, and we have that $\langle \texttt{eval}(trans(s)) \mid m_1, \Sigma_i\rangle \longrightarrow^* \langle stop \mid m_1', \Sigma_1'\rangle$, and $\langle \texttt{eval}(trans(e)) \mid m_2, \Sigma_i\rangle \longrightarrow^* \langle stop \mid m_2', \Sigma_2'\rangle$, then it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.*

To prove this lemma, we apply Lemma 2 when the program hits a branching instruction with secrets on its guard. The lemmas lead to a theorem that guarantees the soundness of the inlining, i.e., that transformed code satisfies noninterference. Formally:

Theorem 1 (Soundness). *For an environment Γ and a program P, we have $\Gamma \vdash P \rightsquigarrow P' \implies \models \{\Gamma\}\, P'\, \{\Gamma'\}$, where Γ' can be extracted from the shadow variables of any run that succeeds to terminate, i.e., the above is true for any Γ' such that if $\langle P \mid m, \cdot\rangle \longrightarrow^* \langle stop \mid m', \Sigma\rangle$ then $\Gamma'(x) = L$ for all variables from $L(m')$ and $\Gamma'(x) = H$, otherwise.*

The theorem above is proved by evaluating the program P' until reaching function $trans$ and then applying Lemma 3.

4 Experiments

With JavaScript as our target language, we have manually translated code according to the transformation rules described in Section 2. In a fully-fledged implementation, the transformation function can be implemented either as a set of regular expressions that parse the supplied string and inline the monitor code or using a full JavaScript parser. Although the parsing by the transformation function may not be generally equivalent to the parsing by the browser, this does not affect the security of the resulting program.

```
var h = true;
var l = false;
if (h) {
  l = true;
}
```

Listing 1.1. Insecure code

```
var l = true;
var h = false;
if (l) {
  h = true;
}
```

Listing 1.2. Secure code

```
var h = true;
var l = false;
if (h) {
  eval('l=true');
}
```

Listing 1.3. Insecure code with `eval`

```
var l = true;
var h = false;
if (l) {
  eval('h=true');
}
```

Listing 1.4. Secure code with `eval`

```
var h = true;
var l = false;
let (pc = pc || shadow['h']) {
  if (h) {
    if (!pc || shadow['l']) {
      shadow['l'] = false || pc;
      l = true;
    }
    else
      throw new Error;
  }
}
```

Listing 1.5. Listing 1.1 transformed

```
var h = true;
var l = false;
let (pc = pc || shadow['h']) {
  if (h) {
    let (pc = pc || false) {
      eval(trans('l=true'));
    }
  }
  else {
    throw new Error;
  }
}
```

Listing 1.6. Listing 1.3 transformed

Manual inlining The design of the monitor affects its performance in comparison to the unmonitored code. Our analysis of the performance of the monitor shows that using the **`let`** statement (which is readily available in, e.g., Firefox) has minimal impact on the performance.

Consider the sample programs in Listings 1.1–1.4. Listing 1.1 is an example of an implicit flow that is insecure: whether a low variable is assigned depends on the value of a high variable in the guard. Listing 1.2 is a dual example that is secure. Listings 1.3 and 1.4 are versions of the same program with an `eval`. For simplicity, the code includes the initialization of variables (both high, h, and low, l,) with constants. Listings 1.5 and 1.6 display the results of transformations for Listings 1.1 and 1.3 (with some obvious optimizations). Tables 1 and 2 present the average performance of our sample programs as well as their respective transformations. The performance is measured as the number of milliseconds to execute the specified number of iterations of a loop that contains a given piece of code. Our experiments were performed on a Dell Precision M2400 PC running Firefox version 3.5.7 on the Windows XP Professional SP3 operating system. We have not included other browsers in our performance test since Firefox is the only browser yet to support the let **`let`**-statement. The next section discuss alternatives to using let and their impact on performance. As can be seen from these results, the inlined monitor either entirely removes (when an insecurity is suspected) or adds an overhead of about 2–3 times the execution time of the untransformed code. The source code for these performance tests is available via [24].

The experiment with the manual inlining shows that the overhead is not unreasonable but it has to be taken seriously for the transformation to scale. Thus, a fully-fledged implementation needs to critically rely on optimizations. We briefly discuss possibilities for optimizations in Section 6.

Alternatives to **`let`**. We briefly discuss alternatives to the **`let`**-based implementation. A naive way of implementing this equivalent structure in JavaScript would be to surround the code block with a **`with`** statement. The **`with`** statement appends an object to the scope chain, much like the **`let`** statement, so that its properties are directly accessi-

Iterations	Listing 1.1	Listing 1.5 (Listing 1.1 transformed)	Listing 1.2	Listing 1.2 transformed
10^6	11	0	11	29
10^7	107	0	99	176
10^8	379	0	336	890
10^9	2696	0	2677	8194

Table 1. Browser performance comparison for simple code

Iterations	Listing 1.3	Listing 1.6 (Listing 1.3 transformed)	Listing 1.4	Listing 1.4 transformed
10^3	37	0	38	58
10^4	172	0	196	262
10^5	1179	0	1219	1898
10^6	13567	0	13675	18654

Table 2. Browser performance comparison for code with `eval`

ble as variables, effectively masking existing variables with the same name until the end of the block. For example, in `var x = { pc: true }; with(x){ pc }`, the `pc` inside the **`with`** block refers to `x.pc`. This implementation would however be disastrous for the monitor because the dynamic creation of new objects and manipulation of the scope chain gave high resource demands, making it more than 1000 times slower than the original code. A more efficient alternative to to **`let`** can be implemented by defining the `pc` as an array. When entering a new block of code, the current index `i` of the `pc` is incremented and `pc[i]` is set to the new value, e.g., `pc[++i] = pc[i -1] || shadow['h'].`

Secure inlining In a fully-fledged implementation, a secure monitor requires a method of storing and accessing the shadow variables in a manner which prevents accidental or deliberate access from the code being monitored and ensure their integrity.

By creating a separate name space for shadow variables, inaccessible to the monitored code, we can prevent them from being accessed or overwritten. In JavaScript, this can be achieved by creating an object with a name unique to the monitored code and defining the shadow variables as properties of this object with names reflecting the variable names found in the code. Reuse of names makes conversion between variables in the code and their shadow counterparts simple and efficient. However, the transformation must ensure that the aforementioned object is not accessed within the code being monitored.

The ability of code to affect the monitor is crucial for the monitor to be secure. JavaScript, however, provides multiple ways of affecting its runtime environment. Even if the code is parsed to remove all direct references to the monitor state variables, like `pc`, indirect access as in `x = 'pc'; this[x]` provides another alternative. Not only is the integrity of the auxiliary variables important, but also the integrity of the transformation function. Monitored code can attempt to replace the transformation function with, e.g., the identity function, i.e., `this['trans'] = function(s){ return s }`. We envisage a combination of our monitor with safe language subset and reference monitoring technology [27, 7, 11, 22, 21] to prevent operations that compromise the integrity of the monitor.

Scaling up Although these results are based on a subset of JavaScript, they scale to a more significant subset. We expect the handling of objects to be straightforward, as fields can be treated similarly to variables. Compared to static approaches, there is no need to restrict aliasing since the actual alias are available at runtime. In order to prevent implicit flows through exceptions, the transformation can be extended to extract control flow information from `try`/`catch` statements and use it for controlling side effects. In order to address interaction between JavaScript and the Document Object Model, we rely on previous results on tracking information flow in dynamic tree structures [32] and on monitoring timeout primitives [30].

5 Related Work

Language-based information-flow security encompasses a large body of work, see an overview [33]. We briefly discuss inlining, followed by a consideration of most related work: on formalizations of purely dynamic and hybrid monitors for information flow.

Inlining Inlined reference monitoring [10] is a mainstream technique for enforcing safety properties. A prominent example in the context of the web is BrowserShield [29] that instruments scripts with checks for known vulnerabilities. The focus of this paper is on inlining for information-flow security. Information flow is not a safety property [26], but can be approximated by safety properties (e.g., [4, 34, 2]), just like it is approximated in this paper (see the remark at the end of Section 1).

Most recently, and independently of this work, Chudnov and Naumann [5] have investigated an inlining approach to monitoring information flow. They inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [31]. The soundness of the inlined monitor is ensured by bisimulation of the inlined monitor and the original monitor.

Dynamic information-flow enforcement Fenton [12] discusses purely dynamic monitoring for information flow but does not prove noninterference-like statements. Volpano [43] considers a purely dynamic monitor to prevent explicit flows. Implicit flows are allowed, and so the monitor does not enforce noninterference. In a flow-insensitive setting, Sabelfeld and Russo [34] show that a purely dynamic information-flow monitor is more permissive than a Denning-style static information-flow analysis, while both the monitor and the static analysis guarantee termination-insensitive noninterference.

Askarov and Sabelfeld [1] investigate dynamic tracking of policies for information release, or *declassification*, for a language with dynamic code evaluation and communication primitives. Russo and Sabelfeld [30] show how to secure programs with timeout instructions using execution monitoring. Russo et al. [32] investigate monitoring information flow in dynamic tree structures.

Austin and Flanagan [2, 3] suggest a purely dynamic monitor for information flow with a limited form of flow sensitivity. They discuss two disciplines: *no sensitive-upgrade*, where the execution gets stuck on an attempt to assign to a public variable in secret context, and *permissive-upgrade*, where on an attempt to assign to a public variable in secret context, the public variable is marked as one that cannot be branched on later in the execution. Our inlining transformation draws on the *no sensitive-upgrade* discipline extended with the treatment of dynamic code evaluation.

Hybrid information-flow enforcement Mechanisms by Venkatakrishnan et al. [41], Le Guernic et al. [18, 17], and Shroff et al. [37] combine dynamic and static checks. The mechanisms by Le Guernic et al. for sequential [18] and concurrent [17] programs are flow-sensitive.

Russo and Sabelfeld [31] show formal underpinnings of the tradeoff between dynamism and permissiveness of flow-sensitive monitors. They also present a general framework for hybrid monitors that is parametric in the monitor's enforcement actions (blocking, outputting default values, and suppressing events). The monitor by Le Guernic et al. [18] can be seen as an instance of this framework.

Ligatti et al. [20] present a general framework for security policies that can be enforced by monitoring and modifying programs at runtime. They introduce *edit automata* that enable monitors to stop, suppress, and modify the behavior of programs.

Tracking information flow in web applications is becoming increasingly important (e.g., a server-side mechanism by Huang et al. [15] and a client-side mechanism for JavaScript by Vogt et al. [42], although, like a number of related approaches, they do not discuss soundness). Dynamism of web applications puts higher demands on the permissiveness of the security mechanism: hence the importance of dynamic analysis.

6 Conclusions

To the best of our knowledge, the paper is the first to consider on-the-fly inlining for information-flow monitors. On-the-fly inlining is a distinguished feature of our approach: the security checks are injected as the computation goes along. Despite the highly dynamic nature the problem, we manage to avoid the caveats that are inherent with dynamic enforcement of information-flow security. We show that the result of the inlining is secure. We are encouraged by our preliminary experimental results that show that the transformation is light on both performance overhead and on the difficulty of implementation.

Future work is centered along the practical considerations and experiments reported in Section 4. As the experiments suggest, optimizing the transformation is crucial for its scalability. The relevant optimizations are both JavaScript- and security-specific optimizations. For an example of the latter, `let` injection is unnecessary when the guard of a conditional is low. Our larger research program pursues putting into practice modular information-flow enforcement for languages with dynamic code evaluation [1], timeout [30], tree manipulation [32], and communication primitives [1]. A particularly attractive application scenario with nontrivial information sharing is web mashups [23].

Acknowledgments Thanks are due to David Naumann for interesting comments. This work was funded by the Swedish research agencies SSF and VR.

References

1. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

2. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
3. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.
4. G. Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'08)*, LNCS, pages 20–34. Springer-Verlag, Mar. 2009.
5. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
6. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
7. D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.
8. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
9. B. Eich. Flowsafe: Information flow security for the browser. `https://wiki.mozilla.org/FlowSafe`, Oct. 2009.
10. U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004.
11. Facebook. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2009.
12. J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.
13. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.
14. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM TOPLAS*, 28(1):175–205, 2006.
15. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. International Conference on World Wide Web*, pages 40–52, May 2004.
16. D. Kozen. Language-based security. In *Proc. Mathematical Foundations of Computer Science*, volume 1672 of *LNCS*, pages 284–298. Springer-Verlag, Sept. 1999.
17. G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, July 2007.
18. G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.
19. X. Leroy. Java bytecode verification: algorithms and formalizations. *J. Automated Reasoning*, 30(3–4):235–269, 2003.
20. J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.
21. S. Maffeis, J. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proc. of ESORICS'09*. LNCS, 2009.
22. S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.
23. J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Apr. 2010.
24. J. Magazinius, A. Russo, and A. Sabelfeld. Inlined security monitor performance test. `http://www.cse.chalmers.se/~d02pulse/inlining/`, 2010.

25. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 193–205, 2008.
26. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.
27. M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.
28. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.
29. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007.
30. A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
31. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
32. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, Sept. 2009.
33. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
34. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.
35. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
36. F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101. Springer-Verlag, 2000.
37. P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.
38. V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.
39. P. H. I. Systems. Sparkada examinar. Software release. `http://www.praxis-his.com/sparkada/`.
40. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proc. Symp. on Static Analysis*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, Sept. 2005.
41. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. International Conference on Information and Communications Security*, pages 332–351. Springer-Verlag, Dec. 2006.
42. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.
43. D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.
44. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
45. D. S. Wallach, A. W. Appel, and E. W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.
46. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.