# On-the-fly Inlining of Dynamic Security Monitors

Jonas Magazinius Alejandro Russo Andrei Sabelfeld

*Dept. of Computer Science and Engineering, Chalmers University of Technology*
*412 96 Gothenburg, Sweden, Fax: +46 31 772 3663*
*Email: andrei@chalmers.se*

**Abstract**

How do we guarantee that a piece of code, possibly originating from third party, does not jeopardize the security of the underlying application? 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 experimental results based on both manual and automatic code rewriting.

*Keywords:* information flow, language-based security, non-interference, reference monitors, inlining

## 1. Introduction

*The problem.* How do we guarantee that a piece of code, possibly originating from third party, does not jeopardize the security of the underlying application? For example, it is common to include third-party code in web applications for producing usage statistics and for providing context-sensitive advertisement. The integration of the third-party code is tight in the sense that in order to function properly, the code cannot be isolated from the main application. This brings us to the challenge of tracking information flow in the web application: how do we ensure that the code preserves confidentiality and integrity of sensitive data?

In the context of web applications, code integration often takes place on the client side, often by including JavaScript code. This provides much power and flexibility because the integrated code is in possession of user credentials. At the same time it brings a security challenge of ensuring that the power is not abused. One alternative is to modify the browser to analyze the code it runs. However, in a scenario when it cannot be guaranteed the users have modified browser versions, a viable alternative is to apply code rewriting to *inline* security checks in the code itself. Depending on the setting, this kind of inlining can be performed by the server, proxy, or the client (or their combination). A intriguing challenge for secure inlining is dynamic code evaluation: how do we ensure that rewritten code escapes security checks by dynamically evaluating a specially crafted string that steals sensitive data? This brings us to the question we answer in this paper: how to tightly and yet securely integrate third-party code in a language with dynamic code evaluation?

*Background.* Analyzing code for security is the subject of the area of *language-based security*. Language-based approach to security gains increasing popularity [17, 40, 49, 37, 20, 29, 8, 12] because it provides natural means for specifying and enforcing application and language-level security policies. Popular highlights include Java stack inspection [49], to enforce stack-based access control, Java bytecode verification [20], to verify bytecode type safety, and web language-based mechanisms such as Caja [29], ADsafe [8], and FBJS [12], to enforce sandboxing and separation by program transformation and language subsets.

Language-based information-flow security [37] 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 [37], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [31, 42, 43].

While the above tools are mostly based on static analysis, considerable progress has been also made on understanding monitoring for secure information flow [13, 47, 45, 19, 18, 41, 27, 38, 3, 2]. Mozilla's ongoing project FlowSafe [10] aims at empowering Firefox with runtime information-flow tracking,

where dynamic information-flow reference monitoring [3, 4] 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* [11] are realized by modifying the underlying application with inlined security checks. Note that there is no widely accepted formal definition for inlined monitors because it highly depends on the architecture of the underlying application and security policy to enforce. 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.

*Contribution.* 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* [9] via control-flow constructs. The shadow variables record information flow by propagating security levels from the righthand side to the lefthand side of an assignment, taking into account both the security level of the expression assigned and the control-flow security context of the assignment.

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. Every time it is called, the library replaces any dynamic code evaluation primitive by its secure version, implying that before any string gets to run, it will be first rewritten to respect the propagation of the security levels of data through shadow variables.

Our approach stays clear of the pitfalls with purely 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 [35]. However, we draw on recent results on dynamic information-flow monitoring [38, 3] that show that security can be enforced purely dynamically. 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* [7, 14, 48, 37]: 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.

Our work includes a discussion of practical considerations and encouraging experimental results. Our experiments with manual transformation give an indication of a reasonable performance overhead. Pushing the experiments further, we fully automate the transformation for a simple subset of JavaScript without dynamic code evaluation. Based on user-defined functions in the Opera browser, we show how the transformation can be deployed in a realistic browser setting.

Note that it is known that noninterference is not a safety property [28, 44]. *Precise* characterizations of what can be enforced by monitoring have been studied in the literature (e.g., [39, 15]), 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.

This paper is modified and extended with respect to its conference version [26]. Compared to it, we have streamlined the transformation and its presentation, included the semantics and proofs, and advanced further our experiments from manual to automatic transformation.

The paper is organized as follows. Section 2 presents the code transformation that inlines security checks to provide information-flow security. Section 3 shows the security guarantees provided by the transformation. Section 4 evaluates our approach in practice. Section 5 discussed related work. Section 6 draws the conclusions for this work.

## 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 sometimes be relabeled during program execution. Our source-to-source transformation injects purely dynamic security checks.

*Language.* Figure 1 presents a simple imperative language enriched with functions, local variables, and dynamic code eval-

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

Figure 1: Language

uation. 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 ($FV(e) \subseteq \{x\}$, where $FV(e)$ denotes the free variables that occur in $e$). Expressions $e$ consist of strings $s$, security levels $\ell$, 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 [50]. Command $\mathtt{let}\ x = e\ \mathtt{in}\ c$ binds the value of $e$ to the local variable $x$, which is only visible in $c$. Command $\mathtt{eval}(e)$ takes a string $e$, which represents a program, and runs it.

*Semantics.* Figure 2 displays the semantics of the language. A program $P$, memory $m$, and *function environment* $\Sigma$ 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 $\Sigma$ consists of a list of definitions of the form $\mathbf{def}\ f(x) = e$. We assume expressions evaluate according to rules of the form $\langle e \mid m, \Sigma\rangle \Downarrow v$, where expression $e$ in memory $m$ and function environment $\Sigma$ evaluates to value $v$. The semantics of expressions is total and call-by-value. Big-step semantic rules for commands have the form $\langle P \mid m, \Sigma\rangle \Downarrow m'$, which indicates that program $P$ in memory $m$ and function environment $\Sigma$ evaluates to (or terminates in) memory $m'$. While most of the rules are standard [50], we pay particular attention to the rule [Eval] for dynamic code evaluation. Dynamic code evaluation takes place 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 ($parse(s) = c$). For simplicity, we assume that executions of programs get stuck when failing to parse. (In a realistic programming language, failing to parse results in a runtime error.)

*Inlining transformation.* At the core of the monitor is a combination of the *no sensitive upgrade* discipline by Austin and Flanagan [3] and a treatment of dynamic code evaluation from a flow-insensitive monitor by Askarov and Sabelfeld [2].

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. Since the transformation operates on strings that represent programs, we consider programs and strings as interchangeable terms. String constants are enclosed by double-quote characters (e.g., "`text`"). Operator $+\!\!+$ concatenates strings (e.g., "`conc`"$+\!\!+$"`atenation`" results in "`concatenation`"). Given the source code $src$ (as a string) and a mapping $\Gamma$ (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 $\Gamma$, the source code $src$ is transformed into the target code $trg$. Since functions are side-effect free, the inlining of function declarations is straightforward: they are simply propagated to the result of the transformation.

In order for the transformation to work, variables $x'$ (for any global variable $x$), as variable $pc$ 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 defer this discussion until Section 4.

The top-level 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 variables appearing in 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 $\Gamma$. 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 $x \in Fresh(c)$, 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 $\mathtt{eval}(trans(src))$, which has the effect of on-the-fly application of function $trans$ to the string $src$ at runtime.

*On-the-fly inlining.* To motivate the transformation rules, we briefly clarify the key dangers that need to be addressed. First, we track explicit flows of the form $x := e$, where the data involved in $e$ might leak into $x$. We ensure that the security level of $x$ is at least as high as the least upper bound of security levels of data involved in $e$. Second, we track implicit flows of the form $\mathtt{if}\ e\ \mathtt{then}\ x := 1\ \mathtt{else}\ x := 0$, where the data involved in $e$ might leak into $x$ through the control-flow structure of the program. It might be tempting to upgrade the security level of $x$ to the upper bound of security levels of data involved in $e$, but this upgrade is not always secure.

To illustrate the case of insecure upgrade, consider the example in Listing 1 (e.g., [35]). For readability, we omit the else branches which we assume contain `skip`. Variables $t$ and $l$ are low-level variables, and $h$ is a high-level variable. Depending

$$\textsc{Def}\ \frac{\langle c \mid m, \cup_i\{f_i \mapsto \textbf{def}\ f_i(x) = e_i\}\rangle \Downarrow m'}{\langle \textbf{def}\ f_1(x) = e_1; \ldots; \textbf{def}\ f_n(x) = e_n; c \mid m, \emptyset\rangle \Downarrow m'} \qquad \textsc{Skip}\ \langle \texttt{skip} \mid m, \Sigma\rangle \Downarrow m \qquad \textsc{Assign}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v}{\langle x := e \mid m, \Sigma\rangle \Downarrow m[x \mapsto v]}$$

$$\textsc{Seq}\ \frac{\langle c_1 \mid m, \Sigma\rangle \Downarrow m' \qquad \langle c_2 \mid m', \Sigma\rangle \Downarrow m''}{\langle c_1; c_2 \mid m, \Sigma\rangle \Downarrow m''} \qquad \textsc{If1}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v \qquad v \neq 0 \qquad \langle c_1 \mid m, \Sigma\rangle \Downarrow m'}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid m, \Sigma\rangle \Downarrow m'}$$

$$\textsc{If2}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow 0 \qquad \langle c_2 \mid m, \Sigma\rangle \Downarrow m'}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid m, \Sigma\rangle \Downarrow m'} \qquad \textsc{While1}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v \qquad v \neq 0 \qquad \langle c; \texttt{while}\ e\ \texttt{do}\ c \mid m, \Sigma\rangle \Downarrow m'}{\langle \texttt{while}\ e\ \texttt{do}\ c \mid m, \Sigma\rangle \Downarrow m'}$$

$$\textsc{While2}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow 0}{\langle \texttt{while}\ e\ \texttt{do}\ c \mid m, \Sigma\rangle \Downarrow m} \qquad \textsc{Let}\ \frac{\langle e \mid m, \Sigma\rangle \Downarrow v \qquad m(x) = v' \qquad \langle c \mid m[x \mapsto v], \Sigma\rangle \Downarrow m'}{\langle \texttt{let}\ x = e\ \texttt{in}\ c \mid m, \Sigma\rangle \Downarrow m'[x \mapsto v']}$$

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

Figure 2: Semantics

on the value of $h$ the branch of the first conditional will either be taken or not. If $h$ is 1, the then branch is taken and the level of $t$ will be upgraded to high and its value set to 1. Thus, when branching on $t$, the then branch will not be taken, the value of $l$ will remain 1, and its level will remain low. If the value of $h$ is 0, then the level of $t$ will remain low. Thus, $l$ will be set to 0. Further, since the level of $t$ is low, so is the level of $l$. To sum up, the value of high variable $h$ is leaked into the low variable $l$. This example illustrates the fundamental tension between dynamism and permissiveness of information-flow monitors [35].

```
t:=0; l:=1;
if h then t:=1;
if !t then l:=0
```

Listing 1: Example of information-flow leak

At the heart of on-the-fly inlining is the transformation function $\mathit{trans}(\cdot)$, displayed in Figure 3. We describe the definition of function $\mathit{trans}(y)$ by cases on string $y$. 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. Following Austin and Flanagan [3], we 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 $H$. 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 ($pc = H$). When $pc \not\sqsubseteq x'$, the transformation forces the program to diverge ($\mathit{loop}$) in order to preserve confidentiality. We define $\mathit{loop}$ as simply `while` 1 `do skip`. This is the only case in the monitor where the execution of the program might be blocked due to a possible insecurity. We remark that if our language did not feature dynamic code evaluation, then it would be entirely possible to avoid blocking altogether: by forcing an upgrade of all variables that might be updated in high context (see, e.g., Chudnov and Naumann's inlining [6] for Russo and Sabelfeld's hybrid monitors [35]). However, in the presence of dynamic code evaluation, it is hard to statically approximate the set of updated variables.

The security level of $x$ is updated to the join of $pc$ and the security level of variables appearing in $e$, as computed by function $\mathit{lev}(\dot{)}$. Function $\mathit{lev}(s)$ returns the least upper bound of the security levels of variables encountered in the string $s$. The formal specification of $\mathit{lev}(s)$ is given as $\sqcup_{x' \in FV(s)} x'$. Observe that directly calling $\mathit{lev}(e)$ does not necessarily returns the confidentiality level of $e$ because the argument passed to $\mathit{lev}$ is the result of evaluating $e$, which is a constant string. To illustrate this point, consider $w = "\mathit{text}"$, $w' = H$, and $e = w$. In this case, calling $\mathit{lev}(e)$ evaluates to $\mathit{lev}("\mathit{text}")$, which is defined to be $L$ since "$\mathit{text}$" does not involve any variable. Clearly, setting $\mathit{lev}("\mathit{text}") = L$ is not acceptable since the string is formed from a secret variable. Instead, the transformation uses function $\mathit{vars}$ to create a string that involves all the variables appearing in an expression $e$ ($\mathit{vars}("e")$). Observe that such string is not created at runtime, but when inlining commands. Function $\mathit{vars}$ returns a string with the shadow variables of the variables appearing in the argument string. For instance, assuming that $e = "\mathit{text}" +\!\!+\ w +\!\!+\ y$, we have that $\mathit{vars}("e") = "w' +\!\!+\ y'"$. Shadow variable $x'$ is then properly updated to $pc \sqcup \mathit{lev}(ex)$, where $ex = \mathit{vars}("e")$.

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 the branches are transformed. In order to track implicit flows, the value of $pc$ is locally raised to the old $pc$ joined with the secu-

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

Figure 3: Inlining transformation

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

Figure 4: Top-level transformation

rity level of the expression in the guard. This manner to manipulate the $pc$, similar to security type systems [48], 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 ($trans(\text{"}c\text{"})$). We note that the rule for `let` offers a form of secure upgrade. It is perfectly security to create new high variables in high context, as long as these variables are local to this context. The second conditional in Listing 1 is a crucial part of the attack. The attack fails if the scope of $t$ is restricted to the first conditional.

The inlining of dynamic code evaluation is the most interesting feature of the transformation. Similarly to conditionals, the inlining of $\texttt{eval}(e)$ locally raises the $pc$: the execution depends on the security level of $e$ and the current value of $pc$ ($pc := pc \sqcup lev(ex)$). In the transformed code, the transformation wires itself before executing calls to `eval` ($\texttt{eval}(trans(e))$). 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* [7, 14, 48, 37]. 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 $\Gamma$-*equal* (written $m_1 =_\Gamma m_2$) if they agree on the variables whole level is $L$ according to $\Gamma$ ($m_1 =_\Gamma m_2 \stackrel{\text{def}}{=} \forall x \in Vars.\, \Gamma(x) = L \Longrightarrow m_1(x) = m_2(x)$). The formal statement of noninterference is as follows.

**Definition 1.** For initial and final security environments $\Gamma$ and $\Gamma'$, respectively, a program $P$ satisfies *noninterference* (written $\models \{\Gamma\}\, c\, \{\Gamma'\}$) if whenever $m_1 =_\Gamma m_2$, $\langle P \mid m_1, \emptyset\rangle \Downarrow m'_1$, and $\langle P \mid m_2, \emptyset\rangle \Downarrow m'_2$, then $m'_1 =_{\Gamma'} m'_2$.

We state two lemmas that lead to the proof of noninterference (found in the appendix).

Similarly to $\Gamma$-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 $\Sigma$ 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 1.** *Given a memory $m$ and a string $s$ representing a command such that $m(pc) = H$ and $\langle \texttt{eval}(trans(s)) \mid m, \Sigma\rangle \Downarrow m'$, we have $m'(pc) = H$, $L(m) = L(m')$, 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 2.** *Given memories $m_1$ and $m_2$ and a string $s$ representing some command, whenever it holds that $L(m_1) = L(m_2)$, $m_1 =_{L(m_1)} m_2$, $\langle \texttt{eval}(trans(s)) \mid m_1, \Sigma_i\rangle \Downarrow m'_1$, and $\langle \texttt{eval}(trans(e)) \mid m_2, \Sigma_i\rangle \Downarrow m'_2$, 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 1 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 $\Gamma$ and a program $P$, assume $\Gamma \vdash P \rightsquigarrow P'$. Extract any environment $\Gamma'$ from the levels of shadow variables in a successfully terminating memory: if $\langle P' \mid m, \emptyset \rangle \Downarrow m'$ then $\Gamma'(x) = L$ for all variables from $L(m')$ and $\Gamma'(x) = H$, otherwise. Then, $\models \{\Gamma\}\, P'\, \{\Gamma'\}$.*

The theorem above is proved by evaluating the program $P'$ until reaching function $trans$ and then applying Lemma 2.

## 4. Experiments

The experiments in this section study the applicability of the technique in general and investigate indicative overhead in the browser setting. With JavaScript as our target language, we initially performed manual transformation of code according to the transformation rules described in Section 2. We have since then pushed the experiments further by performing automatic inlining for a small subset of JavaScript. Although the language does not feature dynamic code evaluation, we foresee no difficulties in accommodating it. In a fully-fledged implementation, the transformation function would cover the complete JavaScript language. Our experiments are an encouraging first step, but a fully-fledged implementation still remains future work.

*Manual inlining.* The design of the monitor affects its performance in comparison to the unmonitored code. To test the overhead introduced by the inlined monitor, we manually transformed some code samples and compared the execution time of the transformed code that of to the original. In transforming, we have favored generality over performance in order to compare among different browsers. For certain browsers performance could be enhanced by taking advantage of browser-specific features. Our experiments were performed on a Dell Precision M2400 PC running the latest version to date of the browsers Firefox, Chrome, Safari, Opera and InternetExplorer on the Windows XP Professional SP3 operating system.

Consider the sample programs in Listings 2–5. Listing 2 is an example of an implicit flow that is insecure; the low variable is incremented while smaller than the high variable. Listing 3 is a dual example that is secure; the high variable is instead decremented while greater than the low variable. Listings 4 and 5 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 6 and 7 display the result of the manual transformation of Listings 2 and 4 (with some obvious optimizations). We encode the security levels $L$ and $H$ as the JavaScript values **`false`** and **`true`**, respectively. In this encoding, $\sqcup$ is represented by `||` (logic OR), and $x \sqsubseteq y$ is given by `!x || y`, where `!` is the logic negation.

Tables 1–5 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 a specified number of iterations of the given piece of code. Because of the slowdown introduced by eval, the code with eval was executed 100000 iterations, while the code without eval was executed 10000000 iterations. As can be seen from these results, for the secure code samples, the inlined monitor adds an overhead ranging from 20 % to 1700 % over the untransformed code, depending on browser. For the transformed insecure code samples the execution time is *0 ms*, as the monitor stops the execution when it encounters an information-flow leak. The source code for these performance tests is available via [25].

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.

*Automatic inlining.* Automatic inlining can be done in a few ways. One could as an example perform inlining on the server side or in a proxy. We have taken an approach that is entirely on the client side. The Opera browser [32] allows the user to include privileged JavaScript called "User JavaScript". User JavaScript can be specified to be included in any page and is executed before any code on that page. It also has access to a number of functions and events not accessible ordinary JavaScript, among these the event "BeforeScript". Before any script is parsed and executed, Opera will fire the "BeforeScript" event. Any handler defined for this event can rewrite the script source code before returning it to the parser of the browser. By defining the transformation function as a handler for that event we can inline the monitor whenever a new script is loaded. Technically this means that regardless of how the application is composed on the server side, we can intercept and inline the monitor before it is interpreted by the browser.

We have implemented the transformation function through the code rewriting feature of the ANTLR [1] language recognition tool. This allows us to generate a parser from ANTLR-grammar, which rewrites the code and inlines the monitor. The generated parser is 7650 LOC of JavaScript, not counting additional 165 LOC for the user defined JavaScript and 6139 LOC in the runtime library. As a performance consideration, we are certain that this code can be dramatically reduced in size using JavaScript compression tools. All sources are available on demand.

At load time, we enumerate the Internet origins of the scripts in a page and for each origin load initial security levels for variables, which we assume are declared in the respective pages. The initial security levels are assigned to shadow variables containing the run-time level of their respective corresponding variable. The set of shadow variables corresponds to the $\Gamma$ of Section 3. For our purposes, it is sufficient to consider a flat security lattice with bottom, top, and incomparable security levels corresponding to origins in-between. If more than one origin claims ownership of a variable, the variable will be treated as top secret. More interesting lattices that allow collaborative information release are treated elsewhere [24]. At parse time, we rewrite the source inlining the monitor according to the transformation rules. At run-time, the monitor validates the inlined

| Program | Source | Transformed | Overhead |
|---|---|---|---|
| Insecure | 1266 ms | 0 ms | -100 % |
| Secure | 160 ms | 322 ms | 101 % |
| Insecure eval | 2265 ms | 0 ms | -100 % |
| Secure eval | 2088 ms | 2497 ms | 19 % |

Table 1: Performance overhead in the Firefox browser

| Program | Source | Transformed | Overhead |
|---|---|---|---|
| Insecure | 3378 ms | 0 ms | -100 % |
| Secure | 173 ms | 413 ms | 138 % |
| Insecure eval | 872 ms | 0 ms | -100 % |
| Secure eval | 581 ms | 963 ms | 65 % |

Table 2: Performance overhead in the Chrome browser

| Program | Source | Transformed | Overhead |
|---|---|---|---|
| Insecure | 2247 ms | 0 ms | -100 % |
| Secure | 115 ms | 1989 ms | 1629 % |
| Insecure eval | 695 ms | 0 ms | -100 % |
| Secure eval | 378 ms | 815 ms | 115 % |

Table 3: Performance overhead in the Safari browser

| Program | Source | Transformed | Overhead |
|---|---|---|---|
| Insecure | 1648 ms | 0 ms | -100 % |
| Secure | 309 ms | 405 ms | 31 % |
| Insecure eval | 646 ms | 0 ms | -100 % |
| Secure eval | 500 ms | 582 ms | 16 % |

Table 4: Performance overhead in the Opera browser

| Program | Source | Transformed | Overhead |
|---|---|---|---|
| Insecure | 2375 ms | 0 ms | -100 % |
| Secure | 1828 ms | 4878 ms | 166 % |
| Insecure eval | 8484 ms | 0 ms | -100 % |
| Secure eval | 8266 ms | 17031 ms | 106 % |

Table 5: Performance overhead in the Internet Explorer browser

```
var l = 0;
while(l < h) {
  l = l + 1;
}
```

Listing 2: Insecure code

```
var l = 0;
while(h > l) {
  h = h - 1;
}
```

Listing 3: Secure code

```
var l = 0;
while(l < h) {
  eval('l=l+1');
}
```

Listing 4: Insecure code with `eval`

```
var l = 0;
while(h > l) {
  eval('h=h-1');
}
```

Listing 5: Secure code with `eval`

```
var l = 0;
pc[++i]=pc[i-1]||s['l']||s['h'];
while(l < h) {
  if (!pc || s['l']) {
    s['l'] = pc || s['l'];
    l = l + 1;
  } else throw new Error;
}
i--;
```

Listing 6: Listing 2 transformed

```
var l = 0;
pc[++i]=pc[i-1]||s['l']||s['h'];
while(l < h) {
  pc[++i] = pc[i-1] || false;
  eval(trans('l=l+1'));
  i--;
}
i--;
```

Listing 7: Listing 4 transformed

checks.

As a design choice, an infrequently used character, such as "_", can be removed from the set of allowed characters for identifiers in the source language. This would prevent valid code, according to the parser, from referring to variables using this character. By naming all shadow variables "$\_name\_$", all shadow variables would become inaccessible to the monitored code while the the browser still can parse it. The shadow variables storing the initial value of a variable is similarly named "$\_\_name$", prepended with two underscores. There is also a small set of special shadow variables used by the monitor itself. These special shadow variables are prepended by one underscore. The naming convention is exemplified in Listing 8.

```
var x;   // User variable
var _x_; // Level of x
var _pc; // Program counter
```

Listing 8: Shadow variable naming convention

In order to track implicit information flows, the level of the program counter is stored in the special variable `_pc`. The `_pc` works like a stack and holds the level of the program counter, reflecting the current execution context. The initial execution context is $\bot$.

Before executing the assignment to a variable the `_pc` is checked to be less than or equal to the level of the variable. If it is not, the program gets stuck in an infinite loop, preventing sensitive upgrade. When determining the new level of the variable, the current level of the execution context (the `_pc`) needs to be taken into account. The new level is therefore determined by taking the join of the level of `_pc` and the level of the expression. Listing 9 gives an example of an assignment before and after transformation. The additions in the transformed code are runtime-checks and information-flow tracking through the shadow variables.

```
x = y + z;
```

```
while(!_pc.leq(_x_));
_x_ = _pc.join(_y_).join(_z_), x = y + z;
```

Listing 9: Assignment transformation rule

Each variable declaration in the source language results in the declaration of three variables in the target language; the variable itself and a shadow variable for holding the level of the variable. At the time of assignment the variable does not yet have a security level, and it might not be assigned any value at all (e.g., "`var x;`"). To deal with this, the owner of a variable has the possibility of providing an initial level for all the variables the owner consider to be security critical. If one is not provided, the initial level is treated as $\bot$. Listing 10 provides an example of how variable declarations are transformed.

```
var x = y;
```

```
while(!_pc.leq(_init['x']));
var _x_ = _pc.join(_init['x']).join(_y_), x, __x = x = y;
```

Listing 10: Declaration transformation rule

The transformation of sequential composition is the sequential composition of the respective transformed parts. To prevent information from flowing implicitly from a high context to a low variable, the monitor tracks the level of the context in every branch. When a branch is encountered, the current level of the `_pc` is stored. Next the `_pc` is updated with the join of its current level and the level of the expression that is branched upon. Each of the two alternative code paths is then transformed and after the two branches join again, the level of the `_pc` before the branch is restored. In the implementation management of the `_pc` is done through the helper methods `branch()` and `join_point`, exemplified in Listing 11.

```
if (x) {
    x = y;
}
else {
    y = z;
}
```

```
_pc.branch(_x_);
if (x) {
    while(!_pc.leq(_x_));
    _x_ = _pc.join(_y_), x = y;
}
else {
    while(!_pc.leq(_y_));
    _y_ = _pc.join(_z_), y = z;
}
_pc.join_point();
```

Listing 11: Branch transformation rule

Since iteration is a form of repeated branching on the same expression, the transformation in Listing 12 is quite similar to the branching case. The current level of `_pc` is stored before iterating, a new level is computed as the join of the current level

and the level of the expression and the old level is restored after iteration finishes. Naturally the body of the iteration is transformed as well.

```
while(x < 10) {
    x = x + 1;
}
```

```
_pc.branch(_x_);
while(x < 10) {
    while(!_pc.leq(_x_));
    _x_ = _pc.join(_x_), x = x + 1;
}
_pc.join_point();
```

Listing 12: Iteration transformation rule

*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. This is achieved in the transformation function by statically checking that the variables used in the code does not violate the name space restrictions. Thereby, the monitored code cannot reference monitor specific variables.

*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 [36] and on monitoring timeout primitives [34].

The ability of code to affect the monitor is crucial for the monitor to be secure. The JavaScript subset used in this article is restricted to static variable references only. Full JavaScript, however, provides multiple ways of dynamically 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 [29, 8, 12, 23, 22] to prevent operations that compromise the integrity of the monitor.

The actions of the monitor is directly reflected in the security state (the shadow variables and program counter). By logging the security state throughout a given execution trace, auditability can be achieved. While auditability is important to a developer, the information can leak data to an attacker, e.g. if a certain execution path was taken or not.

## 5. Related Work

Language-based information-flow security encompasses a large body of work, see an overview [37]. 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 [11] is a mainstream technique for enforcing safety properties. A prominent example in the context of the web is BrowserShield [33] that instruments scripts with checks for known vulnerabilities. However, this work is not directly related because it does not target secure information flow. The focus of this paper is on inlining for information-flow security. Recall that information flow is not a safety property [28], but can be approximated by safety properties (e.g., [5, 38, 3]), just like it is approximated in this paper (see the remark at the end of Section 1).

This paper draws on a conference version [26]. As mentioned in Section 1, the main additions are the streamlined transformation, the semantics and proofs, as well as the experimental part which now includes automatic transformation.

Venkatakrishnan et al. [45] present an inlining of a hybrid monitor for a language with procedures and show that it enforces noninterference.

Recently, and independently of this work, Chudnov and Naumann [6] have investigated an inlining approach to monitoring information flow. They inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [35]. The soundness of the inlined monitor is ensured by bisimulation of the inlined monitor and the original monitor.

The advantage of hybrid monitors over purely dynamic ones is additional permissiveness. Recall the discussion on the delicacies of handling flow sensitivity from Section 2, illustrated in Listing 1. Having access to the entire code, hybrid monitors are able to infer that both variables $t$ and $l$ are dependent on variable $h$. Not having access to the branch not taken, purely dynamic monitors are forced to sacrifice permissiveness when analyzing conditional statements [35]. As remarked in Section 2, if not for dynamic code evaluation, it is possible to avoid blocking the execution altogether.

*Dynamic information-flow enforcement.* Fenton [13] discusses purely dynamic monitoring for information flow but does not prove noninterference-like statements. Volpano [47] 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 [38]

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.

A series of work explores flow-insensitive monitoring. Askarov and Sabelfeld [2] investigate dynamic tracking of policies for information release, or *declassification*, for a language with dynamic code evaluation and communication primitives. Russo and Sabelfeld [34] show how to secure programs with timeout instructions using execution monitoring. Russo et al. [36] investigate monitoring information flow in dynamic tree structures.

Austin and Flanagan [3, 4] 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.* Information-flow mechanisms by Le Guernic et al. [19, 18] and Shroff et al. [41] combine dynamic and static checks. The mechanisms Le Guernic et al. for sequential [19] and concurrent [18] support flow sensitivity.

Russo and Sabelfeld [35] 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. [19] can be seen as an instance of this framework.

Recently, Moore and Chong [30] have proposed two optimizations to the hybrid monitors by Russo and Sabelfeld [35]: selective tracking of variable security levels and smooth porting memory abstractions for languages with dynamically allocated memory.

Ligatti et al. [21] 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. [16] and a client-side mechanism for JavaScript by Vogt et al. [46], 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, we can directly proceed to transforming the branches 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 [2], timeout [34], tree manipulation [36], and communication primitives [2]. A particularly attractive application scenario with nontrivial information sharing is web mashups [24].

*Acknowledgments.* Thanks are due to David Naumann for interesting comments. This work was funded, in part, by the European Community under the WebSand project and, in part, by the Swedish research agencies SSF and VR.

## References

[1] ANTLR Parser Generator. http://www.antlr.org/.

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

[3] 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.

[4] 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.

[5] 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.

[6] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[7] 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.

[8] D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.

[9] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[10] B. Eich. Flowsafe: Information flow security for the browser. `https://wiki.mozilla.org/FlowSafe`, Oct. 2009.

[11] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004.

[12] Facebook. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2009.

[13] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[14] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[15] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM TOPLAS*, 28(1):175–205, 2006.

[16] 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.

[17] D. Kozen. Language-based security. In *Proc. Mathematical Foundations of Computer Science*, volume 1672 of *LNCS*, pages 284–298. Springer-Verlag, Sept. 1999.

[18] G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, July 2007.

[19] 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.

[20] X. Leroy. Java bytecode verification: algorithms and formalizations. *J. Automated Reasoning*, 30(3–4):235–269, 2003.

[21] 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.

[22] S. Maffeis, J. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proc. of ESORICS'09*. LNCS, 2009.

[23] 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.

[24] 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.

[25] J. Magazinius, A. Russo, and A. Sabelfeld. Inlined security monitor performance test. `http://www.cse.chalmers.se/~d02pulse/inlining/`, 2010.

[26] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proceedings of the IFIP International Information Security Conference (SEC)*, Sept. 2010.

[27] 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.

[28] 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.

[29] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.

[30] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proc. IEEE Computer Security Foundations Symposium*, June 2011.

[31] 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.

[32] Opera, User JavaScript. http://www.opera.com/docs/userjs/.

[33] 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.

[34] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[35] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[36] 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.

[37] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[38] 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.

[39] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[40] 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.

[41] 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.

[42] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.

[43] P. H. I. Systems. Sparkada examinar. Software release. `http://www.praxis-his.com/sparkada/`.

[44] 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.

[45] 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.

[46] 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.

[47] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.

[48] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[49] 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.

[50] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

**Appendix (proofs)**

**Definition 2.** Semantic equivalence on configurations is defined as
$\langle c_1 \mid m_1, \Sigma_1\rangle \approx \langle c_2 \mid m_2, \Sigma_2\rangle$ whenever $\langle c_1 \mid m_1, \Sigma_1\rangle \Downarrow m' \Leftrightarrow \langle c_2 \mid m_2, \Sigma_2\rangle \Downarrow m'$.

**Definition 3.** We define semantic equivalence on commands as $c_1 \approx c_2$ whenever $\forall m, \Sigma. \langle c_1 \mid m, \Sigma\rangle \approx \langle c_2 \mid m, \Sigma\rangle$.

**Lemma 1.** *Given a memory $m$ and a string $s$ representing a command such that $m(pc) = H$ and $\langle \texttt{eval}(trans(s)) \mid m, \Sigma\rangle \Downarrow m'$, we have $m'(pc) = H$, $L(m) = L(m')$ and $m =_{L(m)} m'$.*

**Proof**. The proof is by induction on the structure of the command represented by $s$. We have the following cases on $s$, evaluating $\texttt{eval}(trans(s))$ by applying the semantic rule for `eval()` in Figure 2.

$s =$ "`skip`" **-** It holds trivially since $trans(\text{"}\texttt{skip}\text{"}) =$ "`skip`" and `skip` does not modify the memory.

$s =$ "$x := e$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}); x := e \texttt{ else } loop\text{"}) \mid m,\Sigma\rangle \approx$

$\langle \texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \texttt{ else } loop \mid m,\Sigma\rangle \approx$

(it must hold that $pc \sqsubseteq x'$ because the original configuration terminates)

$\langle x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \mid m,\Sigma\rangle \approx$

($pc$ is $H$; because $pc \sqsubseteq x'$ we have $m(x') = H$)

$\langle x := e \mid m,\Sigma\rangle \Downarrow m[x \mapsto v]$

where $parse(\text{"}\texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}); x := e \texttt{ else } loop\text{"}) = \texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \texttt{ else } loop$ and $\langle e \mid m, \Sigma\rangle \Downarrow v$.

Since $m(x') = m'(x') = H$, then $x \notin L(m)$ and $x \notin L(m')$. Hence, $L(m) = L(m')$. Clearly, $m =_{L(m)} m[x \mapsto v] = m'$.

$s =$ "$c_1; c_2$" **-** It is straightforward to show for any commands $c_1$ and $c_2$ that

$\texttt{eval}(\text{"}c_1;c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"} + \text{";"} + \text{"}c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"}); \texttt{eval}(\text{"}c_2\text{"}) \approx c_1; c_2$

Thus, we have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_1\text{"}) + \text{";"} + trans(\text{"}c_2\text{"})) \mid m,\Sigma\rangle \approx$

$\langle \texttt{eval}(trans(\text{"}c_1\text{"})); \texttt{eval}(trans(\text{"}c_2\text{"})) \mid m,\Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_2\text{"})) \mid m'',\Sigma\rangle \Downarrow m'$

where $\langle \texttt{eval}(trans(\text{"}c_1\text{"})) \mid m, \Sigma\rangle \Downarrow m''$.

By induction hypothesis for $c_1$, it holds that $m''(pc) = H$, $L(m) = L(m'')$ and $m =_{L(m)} m''$. By induction hypothesis for $c_2$, it holds that $m'(pc) = H$, $L(m'') = L(m')$ and $m'' =_{L(m)} m'$ and therefore by transitivity $L(m) = L(m')$ and $m =_{L(m)} m'$.

$s =$ "`if` $e$ `then` $c_1$ `else` $c_2$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx \langle \texttt{eval}(\text{"}\texttt{let } pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}) \texttt{ in "} +$

$\quad \text{"}\texttt{if } e \texttt{ then "} + trans(\text{"}c_1\text{"}) + \text{"} \texttt{ else "} + trans(\text{"}c_2\text{"}) + \text{";"}) \mid m,\Sigma\rangle \approx$

$\langle \texttt{let } pc = pc \sqcup lev(FV(\text{"}e\text{"})) \texttt{ in if } e \texttt{ then eval}(trans(\text{"}c_1\text{"}))$

$\quad \texttt{else eval}(trans(\text{"}c_2\text{"})) \mid m,\Sigma\rangle \approx$

($pc$ remains $H$)

$\langle \texttt{if } e \texttt{ then eval}(trans(\text{"}c_1\text{"})) \texttt{ else eval}(trans(\text{"}c_2\text{"})) \mid m,\Sigma\rangle \approx$

(assume $\langle e \mid m,\Sigma\rangle \Downarrow v$, where $v \neq 0$; if the value of $e$ is 0 then the proof proceeds

with $c_2$)

$\langle \texttt{eval}(trans(\text{"}c_1\text{"})) \mid m,\Sigma\rangle \Downarrow m'$

where $parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in}\ \text{"} + \text{"}\texttt{if}\ e\ \texttt{then}\ \text{"} + trans(\text{"}c_1\text{"}) + \text{"}\ \texttt{else}\ \text{"} + trans(\text{"}c_2\text{"}) + \text{";"}) = parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'\text{"}) = \texttt{let}\ pc = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'$ and $c_i' \approx \texttt{eval}(trans(\text{"}c_i\text{"}))$.

By induction hypothesis $L(m) = L(m')$, $m =_{L(m)} m'$ and $m(pc) = H$.

$s = \text{"}\texttt{while}\ e\ \texttt{do}\ c\text{"}$ **-** We have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ pc{=}pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in while}\ e\ \texttt{do}\ \text{"} + trans(\text{"}c\text{"})) \mid m,\Sigma\rangle \approx$

$\langle \texttt{let}\ pc{=}pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in while}\ e\ \texttt{do eval}(trans(\text{"}c\text{"})) \mid m,\Sigma\rangle \approx$

($pc$ remains $H$)

$\langle \texttt{while}\ e\ \texttt{do eval}(trans(\text{"}c\text{"})) \mid m,\Sigma\rangle \approx$

(because the original configuration terminates, the loop iterates over $c$

a finite number of times)

$\langle \texttt{eval}(trans(\text{"}c\text{"}));\ldots;\texttt{eval}(trans(\text{"}c\text{"})) \mid m,\Sigma\rangle \Downarrow m'$

where $parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in while}\ e\ \texttt{do}\ \text{"} + trans(\text{"}c\text{"})) = parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in while}\ e\ \texttt{do}\ c'\text{"}) = \texttt{let}\ pc = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in while}\ e\ \texttt{do}\ c'$ and $c' \approx \texttt{eval}(trans(\text{"}c\text{"}))$.

This proceeds similarly to the case to sequential composition. By repetitive application of the induction hypothesis and the transitivity of equality and $\Gamma$-equality, we have $L(m) = L(m')$, $m =_{L(m)} m'$ and $m(pc) = H$.

$s = \text{"}\texttt{let}\ x = e\ \texttt{in}\ c\text{"}$ **-** We have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ x'{=}pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in let}\ x{=}e\ \texttt{in}\ \text{"} + trans(\text{"}c\text{"})) \mid m,\Sigma\rangle \approx$

$\langle \texttt{let}\ x'{=}pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in let}\ x{=}e\ \texttt{in eval}(trans(\text{"}c\text{"})) \mid m,\Sigma\rangle \Downarrow m' \iff$

(since $pc$ is $H$ then $x'$ is set to $H$)

$\langle \texttt{let}\ x{=}e\ \texttt{in eval}(trans(\text{"}c\text{"})) \mid m[x' \mapsto H],\Sigma\rangle \Downarrow m''\ \&\ m'{=}m''[x' \mapsto m(x')] \iff$

(assume $\langle e \mid m[x' \mapsto H],\Sigma\rangle \Downarrow v$)

$\langle \texttt{eval}(trans(\text{"}c\text{"})) \mid m[x' \mapsto H, x \mapsto v],\Sigma\rangle \Downarrow m'''\ \&\ m'{=}m'''[x' \mapsto m(x'), x \mapsto m(x)]$

where $parse(\text{"}\texttt{let}\ x' = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in let}\ x = e\ \texttt{in}\ \text{"} + trans(\text{"}c\text{"})) =$
$parse(\text{"}\texttt{let}\ x' = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in let}\ x = e\ \texttt{in}\ c'\text{"}) = \texttt{let}\ x' = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in let}\ x = e\ \texttt{in}\ c'$ and $c' \approx \texttt{eval}(trans(\text{"}c\text{"}))$.

By induction hypothesis $m'''(pc) = H$, and therefore $m'(pc) = H$. By induction hypothesis, it also holds that $L(m[x' \mapsto H, x \mapsto v]) = L(m''')$. Then,

$$
\begin{array}{rcll}
L(m[x' \mapsto H, x \mapsto v]) & = & L(m''') & \text{(by setting } x \text{ to } m(x) \text{ on both sides)} \\
L(m[x' \mapsto H]) & = & L(m'''[x \mapsto m(x)]) & \text{(by setting } x' \text{ to } m(x') \text{ on both sides)} \\
L(m) & = & L(m'''[x \mapsto m(x), x' \mapsto m(x')]) & \\
L(m) & = & L(m') &
\end{array}
$$

In the same manner it holds that $m[x' \mapsto H, x \mapsto v] =_{L(m[x' \mapsto H, x \mapsto v])} m'''$ and thereby $m =_{L(m)} m'$.

$s = \text{"}\texttt{eval}(e)\text{"}$ **-** We have

$\langle \texttt{eval}(trans(s)) \mid m,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ pc{=}pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in eval}(trans(e))\text{"}) \mid m,\Sigma\rangle \approx$

$\langle \texttt{let}\ pc{=}pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in eval}(trans(e)) \mid m,\Sigma\rangle \approx$

($pc$ remains $H$)

$\langle \texttt{eval}(trans(e)) \mid m,\Sigma\rangle \approx \langle \texttt{eval}(trans(s')) \mid m,\Sigma\rangle \Downarrow m'$

where $parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in eval}(trans(e))\text{"})$
$= \texttt{let}\ pc = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in eval}(trans(e))$ and $\langle e \mid m, \Sigma\rangle \Downarrow s'$.

By induction hypothesis it holds that $m(pc) = H$, $L(m) = L(m')$ and $m =_{L(m)} m'$.

□

**Lemma 2.** *Given memories $m_1$ and $m_2$ and a string $s$ representing some command, whenever it holds that $L(m_1) = L(m_2)$, $m_1 =_{L(m_1)} m_2$, $\langle \texttt{eval}(trans(s)) \mid m_1, \Sigma_i\rangle \Downarrow m_1'$, and $\langle \texttt{eval}(trans(s)) \mid m_2, \Sigma_i\rangle \Downarrow m_2'$, then it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.*

**Proof**. The proof is by induction on the structure of the command represented by $s$. We have the following cases on $s$, evaluating $\texttt{eval}(trans(s))$ for memories $m_1$ and $m_2$.

$s =$ "`skip`" **-** It holds trivially since $trans($"`skip`"$) =$ "`skip`" and `skip` does not modify the memory.

$s =$ "$x := e$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"});x := e \texttt{ else } loop\text{"}) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(FV(\text{"}e\text{"}));x := e \texttt{ else } loop \mid m_i,\Sigma\rangle \approx$

(it must hold that $pc \sqsubseteq x'$ because the original configuration terminates)

$\langle x' := pc \sqcup lev(FV(\text{"}e\text{"}));x := e \mid m_i,\Sigma\rangle \approx$

(since $L(m_1)=L(m_2)$ then $pc \sqcup lev(FV(\text{"}e\text{"}))=\ell$ is the same in either memory)

$\langle x := e \mid m_i[x' \mapsto \ell],\Sigma\rangle \Downarrow$

$m_i[x' \mapsto \ell, x \mapsto v] = m_i'$

where $parse(\text{"}\texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}); x := e \texttt{ else } loop\text{"}) = \texttt{if } pc \sqsubseteq x' \texttt{ then } x' := pc \sqcup lev(FV(\text{"}e\text{"})); x := e \texttt{ else } loop$ and $\langle e \mid m_i[x' \mapsto \ell], \Sigma\rangle \Downarrow v$.

Trivially $L(m_1') = L(m_2')$ since $m_i[x' \mapsto \ell]$ holds for both memories. For $m_1' =_{L(m_1')} m_2'$ we have two cases to consider; if $\ell$ is $H$ then $x \notin L(m_1')$ and therefore $m_1' =_{L(m_1')} m_2'$ holds. If $\ell$ is $L$ then $e$ does not refer to any high variables and since $m_1 =_{L(m_1)} m_2$ then $m_1(e) = m_2(e)$ and hence the value of $x$ will be the same, $m_1' =_{L(m_1')} m_2'$.

$s =$ "$c_1; c_2$" **-** Recall that for any commands $c_1$ and $c_2$ we have

$\texttt{eval}(\text{"}c_1;c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"} + \text{";"} + \text{"}c_2\text{"}) \approx \texttt{eval}(\text{"}c_1\text{"});\texttt{eval}(\text{"}c_2\text{"}) \approx c_1;c_2$

then we have

$\langle \texttt{eval}(trans(s)) \mid m_i,\Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_1\text{"}) + \text{";"} + trans(\text{"}c_2\text{"})) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{eval}(trans(\text{"}c_1\text{"}));\texttt{eval}(trans(\text{"}c_2\text{"})) \mid m_i,\Sigma\rangle \approx \langle \texttt{eval}(trans(\text{"}c_2\text{"})) \mid m_i'',\Sigma\rangle \Downarrow m_i'$

where $\langle \texttt{eval}(trans(\text{"}c_1\text{"})) \mid m_i, \Sigma\rangle \Downarrow m_i''$.

By induction hypothesis for $c_1$, it holds that $L(m_i) = L(m_i'')$ and $m_i =_{L(m_i)} m_i''$. By induction hypothesis for $c_2$, it holds that $L(m_i'') = L(m_i')$ and $m_i'' =_{L(m_i)} m_i'$ and therefore by transitivity $L(m_i) = L(m_i')$ and $m_i =_{L(m_i)} m_i'$. Because $L(m_1) = L(m_2)$ and $m_1 =_{L(m_1)} m_2$ then also $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

$s =$ "`if` $e$ `then` $c_1$ `else` $c_2$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let } pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}) \texttt{ in } \text{"} +$

$\quad \text{"}\texttt{if } e \texttt{ then } \text{"} + trans(\text{"}c_1\text{"}) + \text{" } \texttt{else } \text{"} + trans(\text{"}c_2\text{"}) + \text{";"}) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{let } pc = pc \sqcup lev(FV(\text{"}e\text{"})) \texttt{ in if } e \texttt{ then eval}(trans(\text{"}c_1\text{"}))$

$\quad \texttt{else eval}(trans(\text{"}c_2\text{"})) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{if } e \texttt{ then eval}(trans(\text{"}c_1\text{"})) \texttt{ else eval}(trans(\text{"}c_2\text{"})) \mid m_i[pc \mapsto \ell],\Sigma\rangle \approx$

$\langle \texttt{eval}(trans(c_i)) \mid m_i[pc \mapsto \ell],\Sigma\rangle \Downarrow m_i'$

where $parse(\text{"}\texttt{let } pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"}) \texttt{ in } \text{"} + \text{"}\texttt{if } e \texttt{ then } \text{"} + trans(\text{"}c_1\text{"}) + \text{" } \texttt{else } \text{"} + trans(\text{"}c_2\text{"}) + \text{";"}) = parse(\text{"}\texttt{let } pc = pc \sqcup lev(FV(\text{"}e\text{"})) \texttt{ in if } e \texttt{ then } c_1' \texttt{ else } c_2'\text{"}) = \texttt{let } pc = pc \sqcup lev(FV(\text{"}e\text{"})) \texttt{ in if } e \texttt{ then } c_1' \texttt{ else } c_2'$, $c_i' \approx \texttt{eval}(trans(\text{"}c_i\text{"}))$, $\langle pc \sqcup lev(FV(\text{"}e\text{"})) \mid m_i, \Sigma\rangle \Downarrow \ell$ and $\langle e \mid m_i[pc \mapsto \ell], \Sigma\rangle \Downarrow v$.

We have two cases depending on the value of $pc$. If $\ell$ is $H$, then regardless of which branch is taken, we can apply Lemma 1 to
$\langle \texttt{eval}(trans(c_i)) \mid m_i[pc \mapsto H], \Sigma\rangle$ and thereby it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$. If $\ell$ is $L$ then $v$ is the same in both memories and the same branch is taken and by induction hypothesis it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

$s =$ "`while` $e$ `do` $c$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ pc{=}pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in while}\ e\ \texttt{do}\ \text{"} + trans(\text{"}c\text{"})) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{let}\ pc{=}pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in while}\ e\ \texttt{do eval}(trans(\text{"}c\text{"})) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{while}\ e\ \texttt{do eval}(trans(\text{"}c\text{"})) \mid m_i[pc{\mapsto}\ell],\Sigma\rangle \approx$

(because the original configuration terminates, the loop iterates over $c$

a finite number of times)

$\langle \texttt{eval}(trans(\text{"}c\text{"}));\ldots;\texttt{eval}(trans(\text{"}c\text{"})) \mid m_i,\Sigma\rangle \Downarrow m_i'$

where $\langle pc \sqcup lev(FV(\text{"}e\text{"})) \mid m_i, \Sigma\rangle \Downarrow \ell$ and $parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in while}\ e\ \texttt{do}\ \text{"} + trans(\text{"}c\text{"})) = parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in while}\ e\ \texttt{do}\ c'\text{"}) = \texttt{let}\ pc = pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in while}\ e\ \texttt{do}\ c'$ and $c' \approx \texttt{eval}(trans(\text{"}c\text{"}))$.

We have two cases depending on the value of $pc$. If $\ell$ is $H$, then we can apply Lemma 1 to $\langle \texttt{eval}(trans(\text{"}c\text{"})) \mid m_i[pc \mapsto H], \Sigma\rangle$ and thereby it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

If $\ell$ is $L$, then the proof proceeds similarly to sequential composition. Note that $v$ is the same in both memories and by repetitive application of induction hypothesis and transitivity of equality and $\Gamma$-equality, we have $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

$s =$ "`let` $x = e$ `in` $c$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ x'{=}pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in let}\ x{=}e\ \texttt{in}\ \text{"} + trans(\text{"}c\text{"})) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{let}\ x'{=}pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in let}\ x{=}e\ \texttt{in eval}(trans(\text{"}c\text{"})) \mid m_i,\Sigma\rangle \Downarrow m_i' \iff$

$\langle \texttt{let}\ x{=}e\ \texttt{in eval}(trans(\text{"}c\text{"})) \mid m_i[x'{\mapsto}\ell],\Sigma\rangle \Downarrow m_i''$ & $m_i'{=}m_i''[x'{\mapsto}m_i(x')] \iff$

(assume $\langle e \mid m_i[x'{\mapsto}\ell],\Sigma\rangle \Downarrow v$)

$\langle \texttt{eval}(trans(\text{"}c\text{"})) \mid m_i[x'{\mapsto}\ell, x{\mapsto}v],\Sigma\rangle \Downarrow m_i'''$ & $m_i'{=}m_i'''[x'{\mapsto}m_i(x'), x{\mapsto}m_i(x)]$

where $parse(\text{"}\texttt{let}\ x' = pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in let}\ x = e\ \texttt{in}\ \text{"} + trans(\text{"}c\text{"})) =$
$parse(\text{"}\texttt{let}\ x' = pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in let}\ x = e\ \texttt{in}\ c'\text{"}) = \texttt{let}\ x' = pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in let}\ x = e\ \texttt{in}\ c'$ and $c' \approx \texttt{eval}(trans(\text{"}c\text{"}))$.

We have two cases depending on the value of $pc$. If $\ell$ is $H$, then we apply Lemma 1 to $\langle \texttt{eval}(trans(\text{"}c\text{"})) \mid m_i[x' \mapsto \ell, x \mapsto v], \Sigma\rangle$ and by that it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

If $\ell$ is $L$, then by induction hypothesis, it holds that $L(m_1''') = L(m_2''')$. Then,

$$\begin{aligned}
L(m_1''') &= L(m_2''') \quad \text{(by setting } x \text{ to } m_i(x) \text{ on both sides)} \\
L(m_1'''[x{\mapsto}m_1(x)]) &= L(m_2'''[x{\mapsto}m_2(x)]) \text{(setting } x'{=}m_i(x') \text{ on both sides)} \\
L(m_1'''[x'{\mapsto}m_1(x'), x{\mapsto}m_1(x)]) &= L(m_2'''[x'{\mapsto}m_2(x'), x{\mapsto}m_2(x)]) \\
L(m_1') &= L(m_2')
\end{aligned}$$

In the same manner it holds that $m_1''' =_{L(m_1''')} m_2'''$ and thereby $m_1' =_{L(m_1')} m_2'$.

$s =$ "$\texttt{eval}(e)$" **-** We have

$\langle \texttt{eval}(trans(s)) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{eval}(\text{"}\texttt{let}\ pc{=}pc \sqcup lev(\text{"} + vars(\text{"}e\text{"}) + \text{"})\ \texttt{in eval}(trans(e))\text{"}) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{let}\ pc{=}pc \sqcup lev(FV(\text{"}e\text{"}))\ \texttt{in eval}(trans(e)) \mid m_i,\Sigma\rangle \approx$

$\langle \texttt{eval}(trans(e)) \mid m_i[pc{\mapsto}\ell],\Sigma\rangle \approx \langle \texttt{eval}(trans(s')) \mid m_i[pc{\mapsto}\ell],\Sigma\rangle \Downarrow m_i'$

where $parse(\text{"}\texttt{let}\ pc = pc \sqcup lev(\text{"} +\!\!+\ vars(\text{"}e\text{"})\ +\!\!+\text{"})\ \texttt{in}\ \texttt{eval}(trans(e))\text{"})$
$= \texttt{let}\ pc = pc \sqcup lev(\mathit{FV}(\text{"}e\text{"}))\ \texttt{in}\ \texttt{eval}(trans(e))$, $\langle e \mid m_i[pc \mapsto \ell], \Sigma\rangle \Downarrow s'$ and $\langle pc \sqcup lev(\mathit{FV}(\text{"}e\text{"})) \mid m_i, \Sigma\rangle \Downarrow \ell$.

We have two cases depending on the value of $pc$. If $\ell$ is $H$ we can apply Lemma 1 to $\langle \texttt{eval}(trans(s')) \mid m_i[pc \mapsto H], \Sigma\rangle$ and thereby it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$. If $\ell$ is $L$ then $v$ is the same in both memories, so the same code is evaluated and by induction hypothesis it holds that $L(m_1') = L(m_2')$ and $m_1' =_{L(m_1')} m_2'$.

□