Capabilities for information flow

Arnar Birgisson   Alejandro Russo   Andrei Sabelfeld
Chalmers University of Technology
{arnar.birgisson,russo,andrei}@chalmers.se

Abstract

This paper presents a capability-based mechanism for permissive yet secure enforcement of information-flow policies. Language capabilities have been studied widely, and several popular implementations, such as Caja and Joe-E, are available. By making the connection from capabilities to information flow, we enable smooth enforcement of information-flow policies using capability systems. The paper presents a transformation that given an arbitrary source program in a simple imperative language produces a secure program in a language with capabilities. We present formal guarantees of security and permissiveness and report on experiments to enforce information-flow policies for web applications using Caja.

1. Introduction

Secure composition Secure composition is a crucial challenge for modern computing systems. In the context of the web, straightforward integration of components (such as script inclusion in web pages) is a driving force for rich networked applications. A particularly thriving area is the area of web *mashups* [37], where standalone web services are mashed into integrated web portals. Often the integrated services combine code that operates on sensitive data (such as financial and health) with third-party code (such as advertisement and statistics). Blind integration implies security hazards of stealing and/or corrupting sensitive information across different components.

Integration vs. separation A key issue when building secure mashups is the delicate balance between integration and separation [40]. Integration drives rich features, while separation helps security. In an ideal world, we might like a mashup designer to focus on features while relying on a clean security policy that protect interactions among the different components from attacks. However, the state of the art in building secure mashups is dominated by separation-based approaches. For example, Yahoo! ADSafe [9], Facebook FBJS [17] and Google Caja [34] rely on conservative language subset techniques to allow integration of untrusted JavaScript code in mashups. In order not to force the programmer into programming on subsets directly, FBJS and Caja offer transformations that perform static analysis on JavaScript code and transform it into code within the respective subsets, rewriting sensitive parts and inserting dynamic checks.

Capability enforcement The Caja approach is particularly inspiring. Caja is based on the *object-capability model* [33, 35]. *Capabilities* [11, 26] are in essence unforgeable references for accessing critical resources. Language capabilities have been studied widely, and several popular implementations in addition to Caja, such as E [13, 33], Joe-E [32], Emily [46], and W7 [38] are available. However, end-to-end security policies have been out of reach for capability languages. By focusing on possessing references, capability languages fall short of such end-to-end guarantees as distinguishing between secure and insecure dependencies. Jaradin et al. remark when introducing the capability language SCOLL [22]:

> In capability systems, preventing data to flow is harder than preventing capabilities from being propagated, even if we only consider overt communication channels.

Information-flow policies This paper addresses closing the gap between intuitive policies and practical enforcement. Our goal is to build on the machinery of capability-based enforcement, but at the same time provide the mashup designer with a light and abstract way to specify security policies in source programs: in terms of sources and sinks of information and restrictions on *information flow* among them, ignoring implementation details internal to program execution.

We believe that a connection between information-flow policies and capability-based enforcement is particularly promising for securing web applications. Imagine a loan calculator web application that operates on secret data such as income but at the same time collects statistics about the use of popular features. A natural security policy is to stipulate that the income is a secret source, the statistics is a public sink and to require *noninterference* [8, 19]: that outputs to the public sink are independent of inputs from the secret source. Information-flow policies are particularly attractive for expressing mashup security policies. They allow expressing a range of decentralized policies from mutual distrust to controlled information release (or *declassification* [43]) of information among mashup components [28].

By making the connection from capabilities to information flow, we enable smooth enforcement of information-flow policies using capability systems. In the loan calculator scenario, the job of the transformation is to translate the policy to capabilities and apply capability-based enforcement. This enables flexibility for the programmer as the source code may manipulate data of different levels of sensitivity. At the same time, security is not compromised: the target of the transformation is a program in a capability language like Caja, which can be safely run within the browser.

Transformation This paper presents a capability-based mechanism for permissive yet secure enforcement of information-flow policies. The main contribution is a transformation that given an arbitrary source program in a simple imperative language produces a secure program in a language with capabilities. The key ingredients in the target language that allow us such a transformation are references and scopes. Unforgeable references allow for tight control over critical operations with side effects, such as writing to a location. We are able to track information flow by disabling refer-

PLAS '11 June 5, 2011, San Jose, California June 05, USA.
Copyright © 2011 ACM [to be supplied]...$10.00

ences to variables that are necessary to perform unsafe side effects in critical scopes (when the value to be written might depend on secret data).

Information flow in programs can be classified into two basic types: explicit flows, when secret data is passed to a public destination, or *implicit flows* [10], where information is leaked through branching on secret data and exposing different publicly-observable behavior in the branches.

For explicit flows, assigning an expression to a variable requires a capability to write to the variable. When the expression is secret, the transformation ensures that no capability to write the value of the expression to a public variable is passed. For implicit flows, whenever entering branches of a conditional with a secret guard or the body of a loop with a secret guard, again, the transformation ensures that no capability to write the value of the expression to a public variable is passed.

We have intentionally stripped down our language setting to the simplest possible, in order to pin down the essence of the problem: information flow can be tracked by controlling capabilities of code segments. Perhaps surprisingly, objects are not necessary in the target language: all we need is references and scopes.

Security and permissiveness We present formal guarantees of security and permissiveness for the transformation. We show that no matter what the source program is, the result of transformation satisfies the security condition of noninterference [8, 19]. Transformed programs might get stuck, e.g., as a result of not having a reference to perform an operation. However, we demonstrate that the transformation does not introduce abnormal termination at the price of permissiveness.

Our reference points for evaluating permissiveness are typical static and dynamic information-flow mechanisms from the information-flow literature. Classical static Denning-style enforcement [10, 47] rules out programs with explicit and implicit flows at compile time. It is known [42] to be less permissive than monitors that block explicit and implicit flows at runtime. For example, programs with insecure dead code might be rejected by the former but accepted by the latter.

It turns out that the permissiveness of the transformation is the same as that of dynamic monitoring: if the transformed program diverges, then so does the original program when monitored by a typical dynamic information-flow monitor [42], and vice versa. Further, we show a *transparency* result: if a transformed program terminates, then so does the original program and the two of them agree on the final result.

Thus, our transformation can be viewed as an *inlining* [16] of a dynamic information-flow monitor. The advantage of inlining security checks is that security can be enforced without modifying the runtime environment (e.g, a browser). In contrast to building a custom inlining transformation for information flow, we benefit from the infrastructure already in place for such capability enforcement systems as Caja.

Caja-based experiments Finally, we report on experiments to enforce information-flow policies for web applications using Caja. Using the loan calculator as a running example, we give a simple implementation in an imperative language, show how it translates to a capability program and illustrate its security and permissiveness.

Overview Section 2 presents a simple language with capabilities. Section 3 defines an information-flow security condition for this language. Section 4 describes a transformation from a simple imperative language to the language with capabilities. This section also contains a security and a permissiveness theorem. Section 5 describes our experiments with transforming JavaScript code to a capability safe subset. Section 6 discusses related work. Section 7 contains concluding remarks and future work.

$$
\begin{array}{llcl}
\text{Expressions} & e & ::= & a \mid x \mid e * e \mid \texttt{null} \\
\text{Commands} & c & ::= & \texttt{skip} \mid x := e \mid c\,;c \\
 & & \mid & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \\
 & & \mid & \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{call}\ p(\vec{x}) \\
\\
\text{Variables} & \mathit{Var} & = & \{x, y, p, \dots\} \\
\text{Values} & \mathit{Val} & = & \{a, b, \dots\} \\
\text{Operators} & * & \in & \{+, -, \dots\} \\
\text{Procedures} & \mathit{Proc} & = & \{\, \texttt{proc}\ (\vec{y})\ c\,, \dots\} \\
\\
\text{Memory} & \mu & : & \mathit{Loc} \hookrightarrow \mathit{Val} \\
\text{References} & \rho & : & \mathit{Var} \hookrightarrow \mathit{Loc} \cup \mathit{Proc}
\end{array}
$$

Figure 1. Notation and syntax

2. Language

Capabilities are a widely known method for protection at operating-system level [11, 26]. At the heart of capability-based security is that code *possessing* a capability, such as a reference to a file or system object, is allowed to access the resource. Capabilities can be thought of as tokens, tickets, or keys that give to the code the ability to access some resources. In a programming-language context, the object-capability model [33] provides means to isolate certain parts of the code by not giving them certain capabilities. Several programming languages implement this model (e.g., E[13, 33], Joe-E[32], Emily[46], W7[38] and Caja[34]) by eliminating language constructs that might leak authority by passing capabilities to code that is not intended to have them.

Syntax For the purpose of this paper, we consider a simple imperative language with capabilities (see Figure 1). Variables range over $x, y, p, \dots$, while values range over $a, b, \dots$. Expressions e consist of literals a, variables x, and composite expressions $e * e$ (where $*$ is a binary operation). The special literal `null` denotes a distinguished location that is never written to. We'll use the syntactic form `null` to refer to this location below when there is no risk of ambiguity. Commands, denoted by c, consist of standard imperative instructions: `skip`, assignments, sequential composition, conditionals, and loops. The language contains the additional command `call` $p(\vec{x})$ representing the call to a procedure p with a list of arguments $\vec{x}$. Only variables are allowed as arguments, since they will be passed by reference. Procedures are of the form `proc` $(\vec{y})\ c$, where $\vec{y}$ is the list of its formal arguments, and the command c is its body. We refer to commands that do not contain a `call` command as *simple commands*.

For simplicity, we choose the approach of *capability-based addressing* [26], where variable identifiers are references to objects in the system, which in our case are values stored in memory. In our approach, however, we impose one level of indirection between identifiers and values to easily model deletion of capabilities. More precisely, references refer to memory locations or procedures ($\rho : \mathit{Var} \hookrightarrow \mathit{Loc} \cup \mathit{Proc}$), and memory locations refer to values stored in memory ($\mu : \mathit{Loc} \hookrightarrow \mathit{Val}$). Capabilities are then modeled using references. For instance, a piece of code with an identifier x, denoting a non-null reference ($\rho(x) \neq \texttt{null}$), has the capability to read and write the value stored into that location. To revoke such a capability, it is enough to delete or overwrite the references to the location where the value is stored. Observe that by doing so, the value is still in memory but it is not accessible since the code has no references to it.

Semantics The semantics of our language are given as small-step operational semantics in Figure 2.

Configurations have the form $\langle c \mid \rho, \mu, S \rangle$, where c is a command, ρ and μ are mappings as described before, and S is a call

$$\text{SEQ}_1\ \frac{}{\langle \texttt{skip}; c \mid \rho, \mu, S\rangle \to \langle c \mid \rho, \mu, S\rangle} \qquad \text{SEQ}_2\ \frac{\langle c_1 \mid \rho, \mu, S\rangle \to \langle c_1' \mid \rho', \mu', S'\rangle}{\langle c_1; c_2 \mid \rho, \mu, S\rangle \to \langle c_1'; c_2 \mid \rho', \mu', S'\rangle}$$

$$\text{ASSIGN}\ \frac{v = eval(e, \rho, \mu) \qquad \rho(x) \neq \texttt{null}}{\langle x := e \mid \rho, \mu, S\rangle \to \langle \texttt{skip} \mid \rho, \mu[\rho(x) \mapsto v], S\rangle}$$

$$\text{IF}_1\ \frac{true(eval(e, \rho, \mu))}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \rho, \mu, S\rangle \to \langle c_1 \mid \rho, \mu, S\rangle} \qquad \text{IF}_2\ \frac{\neg true(eval(e, \rho, \mu))}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \rho, \mu, S\rangle \to \langle c_2 \mid \rho, \mu, S\rangle}$$

$$\text{WHILE}_1\ \frac{true(eval(e, \rho, \mu))}{\langle \texttt{while}\ e\ \texttt{do}\ c \mid \rho, \mu, S\rangle \to \langle c; \texttt{while}\ e\ \texttt{do}\ c \mid \rho, \mu, S\rangle} \qquad \text{WHILE}_2\ \frac{\neg true(eval(e, \rho, \mu))}{\langle \texttt{while}\ e\ \texttt{do}\ c \mid \rho, \mu, S\rangle \to \langle \texttt{skip} \mid \rho, \mu, S\rangle}$$

$$\text{CALL}\ \frac{\rho(p) = \texttt{proc}\ (\vec{y})\ c \qquad \rho' = \rho[y_i \mapsto \rho(x_i)], i \in \{1, \ldots, n\}}{\langle \texttt{call}\ p(\vec{x}) \mid \rho, \mu, S\rangle \to \langle c; \Lsh \mid \rho', \mu, \rho : S\rangle} \qquad \text{RETURN}\ \frac{}{\langle \Lsh \mid \rho, \mu, \rho' : S\rangle \to \langle \texttt{skip} \mid \rho', \mu, S\rangle}$$

Figure 2. Operational semantics

stack. If a transition leads to a configuration only with the command skip, then we say the execution *terminates*. We use the standard notation $\rho[x \mapsto l]$, where $l \in Loc$, to represent a mapping ρ' that is identical to ρ, except that $\rho'(x) = l$. We use the same notation for updates of μ. We assume the existence of a (partial) function $eval$, that assigns each expression its value, given a reference map and memory. The rule ASSIGN stores the result of evaluating an expression e into the memory location referenced by x ($\mu[\rho(x) \mapsto v]$). Predicate $true$ defines a truth value for each value in Val. The semantic rules for conditionals, loops, and sequential composition are standard [48], and thus we omit their description.

The rules for procedure calls deserve some explanation. In particular, procedure call arguments are invariably passed by reference, and procedures do not return values. The rule CALL is applied when a procedure is called. The rule pushes the current reference map ρ onto the call stack ($\rho : S$), binds the formal arguments to the memory locations referenced by the actual arguments ($\rho' = \rho[y_i \mapsto \rho(x_i)]$), and selects the procedure body for execution. We use the special command $\Lsh$, which is purely internal to the semantics, to mark the return point from procedure calls. This command invokes the rule RETURN, which restores the reference map of the caller by popping it from the stack.

3. Security

We specify security for programs via noninterference [8, 19]. We assume a security lattice $\mathcal{L}$ where security levels are ordered by a partial order $\sqsubseteq$, with the intention to only allow leaks from data at level ℓ_1 to data at level ℓ_2 when $\ell_1 \sqsubseteq \ell_2$. Each variable is associated with a security level. The notation $lev(e)$, where e is an expression, refers to the least upper bound of the levels of variables appearing in e.

Since security levels are assigned to variables, and variables store references, we must be careful to avoid aliasing problems, i.e., where two variables, with different security levels, refer to the same memory location. The next definition introduces the notion of *aliasing safety* in order to avoid such problems. It states that two variables referring to the same location must have the same security level.

Definition 1 (Aliasing safety)**.** *We say that a map of references ρ is* aliasing safe *if and only if for any pair of variables x, y, it holds that if $\rho(x) = \rho(y)$ then $lev(x) = lev(y)$.*

Since our language has no primitives to alter or define references, aliasing safety is trivially preserved by the program semantics in Figure 2.

Lemma 1 (Preservation of aliasing safety)**.** *For any command c, memory μ, stack S of aliasing safe mappings, and an aliasing safe mapping ρ, if $\langle c \mid \rho, \mu, S\rangle \to \langle c' \mid \rho', \mu', S'\rangle$, then ρ' is aliasing safe, and S' contains only aliasing safe mappings.*

The above might be avoided by assigning levels to locations instead, but this causes more problems later, when we model capabilities by assigning variables to the null location, as we will still need to refer the level of the variable.

We assume an attacker model, where the attacker can only observe values stored at locations under certain security level ℓ. The attacker's view of program memory is defined by a ℓ-equivalence relation w.r.t. ρ. More precisely, we have the following definition.

Definition 2 (ℓ-equivalence)**.** *Given a security level ℓ and an aliasing safe map of references ρ, we say that two memories μ, μ' are* ℓ-equivalent w.r.t. *ρ if and only if they agree on all locations at levels $\ell' \sqsubseteq \ell$,*

$$\forall x \in \mathrm{dom}(\rho) : lev(x) \sqsubseteq \ell \Rightarrow \mu(\rho(x)) = \mu'(\rho(x))$$

We write this as $\mu \approx_{\ell,\rho} \mu'$ and omit ρ when it is unambiguous from the context.

We now define our security condition of noninterference. Intuitively, if a program is run, under some references ρ and memory μ, and produces some results below or at security level ℓ, then the same program will produce the same results below or at level ℓ under a ℓ-equivalent memory μ' ($\mu \approx_{\ell,\rho} \mu'$).

Definition 3 (Termination-insensitive noninterference)**.** *Given a security level ℓ, and an aliasing safe reference map ρ, a command c satisfies* termination-insensitive noninterference *if and only if for any two memories μ_1 and μ_2 that are ℓ-equivalent w.r.t. ρ ($\mu_1 \approx_{l,\rho} \mu_2$), if*

$$\langle c \mid \rho, \mu_1, S\rangle \to^* \langle \texttt{skip} \mid \rho, \mu_1', S\rangle$$
$$\textit{and}\ \langle c \mid \rho, \mu_2, S\rangle \to^* \langle \texttt{skip} \mid \rho, \mu_2', S\rangle,$$

then $\mu_1' \approx_{l,\rho} \mu_2'$.

Note that the two terminating traces must agree on the reference map and the call stack. We later prove that, in our setting, this is always the case for any terminating command c (see Lemma 2 in Section 4).

$$\textsc{T-Skip}\ \frac{}{\vdash\ \texttt{skip} \leadsto \texttt{skip}} \qquad \textsc{T-Seq}\ \frac{P_1 \vdash c_1 \leadsto \hat{c}_1 \quad P_2 \vdash c_2 \leadsto \hat{c}_2}{P_1, P_2 \vdash c_1; c_2 \leadsto \hat{c}_1; \hat{c}_2}$$

$$\textsc{T-Assign}\ \frac{p \text{ fresh}}{\{p \mapsto \texttt{proc}\ (\vec{W})\ \bar{x} := e\} \vdash x := e \leadsto \texttt{call}\ p(\vec{W}_{\uparrow lev(e)})}$$

$$\textsc{T-Branch}\ \frac{P_1 \vdash c_1 \leadsto \hat{c}_1 \quad P_2 \vdash c_2 \leadsto \hat{c}_2 \quad p_1, p_2 \text{ fresh}}{\begin{array}{l} P_1, \{p_1 \mapsto \texttt{proc}\ (\vec{W})\ \hat{c}_1\}, \\ P_2, \{p_2 \mapsto \texttt{proc}\ (\vec{W})\ \hat{c}_2\} \end{array} \vdash \begin{array}{r} \texttt{if}\ e\ \texttt{then}\ c_1 \\ \texttt{else}\ c_2 \end{array} \leadsto \begin{array}{r} \texttt{if}\ e\ \texttt{then}\ \texttt{call}\ p_1(\vec{W}_{\uparrow lev(e)}) \\ \texttt{else}\ \texttt{call}\ p_2(\vec{W}_{\uparrow lev(e)}) \end{array}}$$

$$\textsc{T-Loop}\ \frac{P \vdash c \leadsto \hat{c} \quad p \text{ fresh}}{P, \{p \mapsto \texttt{proc}\ (\vec{W})\ \hat{c}\} \vdash \texttt{while}\ e\ \texttt{do}\ c \leadsto \texttt{while}\ e\ \texttt{do}\ \texttt{call}\ p(\vec{W}_{\uparrow lev(e)})}$$

Figure 3. Program transformation rules

4. Transformation

Figure 3 presents the transformation rules. They have the form $P \vdash c \leadsto \hat{c}$, where a simple command c (i.e., without procedure calls) is transformed into command $\hat{c}$ containing calls to procedures defined in P. More specifically, P is a mapping with elements of the form $p \mapsto \texttt{proc}\ (\vec{y})\ c$. Note that in a rule $P \vdash c \leadsto \hat{c}$, the set P is essentially an output of the rule. We have chosen to write it in the position of a context object on the left to improve the readability of our proofs. When extending a mapping P with a procedure named $p \notin \text{dom}(P)$, we write $P, \{p \mapsto \texttt{proc}\ (\vec{y})\ c\}$. We write P_1, P_2 to the union of mappings P_1 and P_2 provided that $\text{dom}(P_1) \cap \text{dom}(P2) = \emptyset$.

For each variable x present in c, the transformation introduces a fresh variable $\bar{x}$ with the same security level as x. We write $\vec{W}$ for the ordered list of all such variables. For instance, if a program has variables x, y, z, then $\vec{W} = \bar{x}$,$\bar{y}$,$\bar{z}$. Observe that such lists of variables can be used as arguments for procedures. We define the syntax $\vec{W}_{\uparrow \ell}$ for restricting a list of variables $\vec{W}$ based on their security levels. Intuitively, $\vec{W}_{\uparrow \ell}$ stores the `null` reference into those variables that have security level no higher or equal than ℓ. Formally:

Definition 4 (Restriction on $\vec{W}$)**.** *Given $\vec{W} = \bar{x}_1, \ldots, \bar{x}_n$, we define $\vec{W}_{\uparrow \ell}$ as the list of variables $y_1, \ldots, y_n$ such that*

$$y_i = \begin{cases} \bar{x}_i & \textit{if } \ell \sqsubseteq lev(x_i) \\ \texttt{null} & \textit{otherwise.} \end{cases}$$

The key aspects of the transformation rely on the following points about transformed programs: (i) Plain variables (e.g., x, y, z) are only used for reading values from memory, (ii) writing into a memory location indicated by variable x is only done through the variable $\bar{x}$. We refer to $\bar{x}$ as the *write reference* of x. (iii) Write references (e.g., $\bar{x}$, $\bar{y}$, $\bar{z}$) are bound through procedure parameters. In that manner, it is possible to remove the capability for a procedure invocation to write into a variable x by simply passing `null` as the argument for $\bar{x}$ when calling it. This is illustrated in Figure 4, where part (a) represents a reference map where x and $\bar{x}$ are mapped to the same location l, which maps to the value behind x. Part (b) shows a situation where $\bar{x}$ is a `null`-reference and thus the value behind x cannot be modified. In a setting of capability languages, x and $\bar{x}$ can be thought of as *data diodes* [33] that pass information only in one direction (by reading and writing, respectively).

The rule T-SKIP is trivial. When transforming assignments, rule T-ASSIGN uses capabilities to avoid explicit flows. An assignment to variable x is translated to a procedure call that performs the assignment through the write reference $\bar{x}$ ($p \mapsto \texttt{proc}\ (\vec{W})\ \bar{x} := e$).

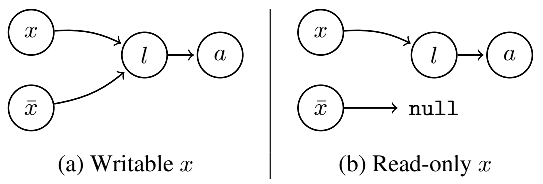


Figure 4. References as capabilities

Observe that the procedure takes the list of write references ($\vec{W}$) as arguments. When calling the procedure, the transformation makes sure that $\bar{x}$ is not `null` if the security level of x is greater than the security level of the expression. These restrictions is achieved by passing $\vec{W}_{\uparrow lev(e)}$ as argument to the procedure ($\texttt{call}\ p(\vec{W}_{\uparrow lev(e)})$). In other words, the body of procedure p does not have the capability to write to x if the security level of x is strictly below or incomparable with the security level of e. The rule T-SEQ obtains $\hat{c}_1$ and $\hat{c}_2$ by respectively transforming c_1 and c_2. Observe that the definition of procedures obtained when transforming each component are concatenated in the conclusion of the rule (P_1, P_2).

The rule T-BRANCH deserves some attention. This rule helps avoiding implicit flows. The transformed branches $\hat{c}_1$ and $\hat{c}_2$ are placed each in a procedure of its own so that we can control their capabilities. The transformed conditional statement calls the respective procedures, taking care of passing only write capabilities that are above or equal to the level of the conditional guard. Thus, neither branch is able to write to locations that are strictly below or incomparable with the guard level, which otherwise would allow for implicit leaks to happen. The rule T-LOOP is analogous to the rule T-BRANCH and thus we omit its explanation.

An execution of a program c on reference map ρ and memory μ is replaced by an execution of the transformed program $\hat{c}$ on $\hat{\rho}$ and μ. Here, $\hat{\rho}$ is obtained by extending ρ with references to procedures generated by the translation.

4.1 Soundness

In this section we prove that our translation provides sound enforcement noninterference. We start by proving a simple property of our language, namely that terminating executions of a command do not change the reference map or the stack. Formally:

Lemma 2. *Let c be a command, ρ a map of references, μ a memory and S a stack, such that*

$$\langle c \,|\, \rho, \mu, S \rangle \to^* \langle \texttt{skip} \,|\, \rho', \mu', S' \rangle.$$

Then $\rho' = \rho$ and $S' = S$.

As seen in the definition of noninterference, this property allows us to focus only on pairs of runs over different memories and not worry about the output reference maps.

The next lemma is essential for proving soundness. It states that if code only has the capabilities (i.e., non-null references) to write into memory locations at or above certain security level ℓ, then it respects such capabilities. In other words, executing such code does not alter memory locations at security level ℓ' where $\ell \not\sqsubseteq \ell'$. This corresponds to the *no write down* (or **-property*) in the Bell-LaPadula model [3].

Lemma 3. *Consider commands c and $\hat{c}$ and procedures P such that $P \vdash c \rightsquigarrow \hat{c}$, an aliasing safe map ρ for the variables appearing in c and memory μ. Choose a fresh name $p \notin \mathrm{dom}(P) \cup \mathrm{dom}(\rho)$. If*

$$\langle \texttt{call}\ p(\vec{W}_{\uparrow \ell}) \,|\, \hat{\rho}, \mu, S\rangle \rightarrow^* \langle \hat{c}' \,|\, \hat{\rho}', \mu', S'\rangle,$$

where $\hat{\rho} = \rho, P, \{p \mapsto \texttt{proc}\ (\vec{W})\ \hat{c}\}$, then

$$\forall x \in \mathrm{dom}(\rho) : \ell \not\sqsubseteq lev(x) \Rightarrow \mu(\rho(x)) = \mu'(\rho'(x)).$$

Now we can state our main security theorem, that any two terminating runs of a transformed program from ℓ-equivalent memories, must terminate in ℓ-equivalent memories as well. The proof of this and further formal statements are reported in the appendix.

Theorem 1 (Security of transformed programs)**.** *Consider a security level ℓ, commands c and $\hat{c}$ and procedures P such that $P \vdash c \rightsquigarrow \hat{c}$, an aliasing safe map ρ for the variables appearing in c, and a pair memories μ_1 and μ_2 with $\mu_1 \approx_{\ell,\rho} \mu_2$. It then holds that transformed program $\hat{c}$ satisfies noninterference. Formally: If*

$$\langle \hat{c} \,|\, \hat{\rho}, \mu_1, S\rangle \rightarrow^* \langle \texttt{skip} \,|\, \rho'_1, \mu'_1, S'_1\rangle$$
$$\textit{and } \langle \hat{c} \,|\, \hat{\rho}, \mu_2, S\rangle \rightarrow^* \langle \texttt{skip} \,|\, \rho'_1, \mu'_2, S'_2\rangle$$

where $\hat{\rho} = \rho, P$; then $\mu'_1 \approx_{\ell,\rho} \mu'_2$.

4.2 Permissiveness and transparency

Now that we have showed the soundness of the transformation, we turn our focus to the price we pay for securing arbitrary programs: how much semantic modification with respect to the original program is caused by the transformation?

As foreshadowed in Section 1, our reference points for evaluating permissiveness are typical static and dynamic information-flow mechanisms from the information-flow literature. Classical static Denning-style enforcement [10, 47] rules out programs with explicit and implicit flows at compile time. It is known [42] to be less permissive than monitors that block explicit and implicit flows at runtime. For example, programs with insecure dead code might be rejected by the former but accepted by the latter.

It turns out that the permissiveness of the transformation is the same as that of dynamic monitoring. In the rest of this section, we show permissiveness and transparency results. First, if the transformed program diverges, then so does the original program when monitored by a typical dynamic information-flow monitor [42], and vice versa. Second, we show a transparency result: if a transformed program terminates, then so does the original program and the two of them agree on the final result.

Noninterference by a runtime monitor Our approach accepts the same set of secure programs as a simple runtime monitor. To illustrate that, we select the monitor from [42] for simple commands, i.e., the language in Figure 1 without procedures. The semantics for the monitor present transitions of the form $st \xrightarrow{\alpha} st$, where st denotes a stack of security levels, and α ranges over events triggered by commands. The monitor uses the information in the event to determine if the execution can proceed. Intuitively, every time that a command triggers an event α, the monitor allows execution to proceed if it is also able to perform the labeled transition α. We extend the semantics given in Figure 2 to trigger events when performing small-step transitions. Figure 5 shows the events triggered by each semantic rule. For instance, rule ASSIGN triggers the event $a(x, e)$, i.e., the conclusion of the rule becomes:

Rule	Event	Interpretation
SEQ$_1$	nop	No operation
SEQ$_2$	α	Propagates event α from premise
ASSIGN	$a(x,e)$	Assignment
IF$_1$, IF$_2$	$b(e)$	Entering branch
WHILE$_1$	$b(e)$	Entering loop iteration
WHILE$_2$	nop	Skipping loop
END	f	Exiting branch

Figure 5. Events

$$\text{NOP}\ \frac{}{st \xrightarrow{nop} st}$$

$$\text{FLOW}\ \frac{lev(e) \sqsubseteq lev(x) \qquad lev(st) \sqsubseteq lev(x)}{st \xrightarrow{a(x,e)} st}$$

$$\text{BRANCH}\ \frac{}{st \xrightarrow{b(e)} lev(e) : st} \qquad \text{END}\ \frac{}{\ell : st \xrightarrow{f} st}$$

Figure 6. Monitor semantics

$$\langle x := e \,|\, \rho, \mu, S\rangle \xrightarrow{a(x,e)} \langle \texttt{skip} \,|\, \rho, \mu[\rho(x) \mapsto v], S\rangle$$

Entering and leaving branches of conditionals and bodies of loops are also recorded by the semantics. For example, the conclusion of rule IF$_1$ becomes:

$$\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \,|\, \rho, \mu, S\rangle \xrightarrow{b(e)} \langle c_1; \texttt{end} \,|\, \rho, \mu, S\rangle$$

where $b(e)$ signals branching on expression e. Note the instrumentation of the command in the resulting configuration. The special command `end` is inserted in order to record when the computation exists the branch or loop body. This command is appended in the result of the transition rules IF$_1$,IF$_2$, and WHILE$_1$. The semantic rule for executing *end* triggers event f to indicate to the monitor that a branch has just finished executing. Formally:

$$\text{END}\ \frac{}{\langle \texttt{end} \,|\, \rho, \mu, S\rangle \xrightarrow{f} \langle \texttt{skip} \,|\, \rho, \mu, S\rangle}$$

A monitored execution of a program is then driven by a single transition rule:

$$\frac{\langle c \,|\, \rho, \mu, \epsilon\rangle \xrightarrow{\alpha} \langle c' \,|\, \rho', \mu', \epsilon\rangle \qquad st \xrightarrow{\alpha} st'}{\langle c \,|\, \rho, \mu; st\rangle \rightarrow \langle c' \,|\, \rho', \mu'; st'\rangle} \qquad (\star)$$

This formalizes the idea that every triggered event α must be synchronized with the monitor. The monitor might disallow execution by stopping it (whenever it is unable to perform an α transition). The symbol ϵ denotes the empty call stack. Observe that we do not need a call stack since we only consider monitored executions of programs without procedure calls.

As mentioned earlier, the state of the monitor consists on a stack of security levels. The stack st, which initially is empty, keeps track of the dynamic security context [18][24]: the security levels of expressions appearing in the guards of branching commands (i.e., conditionals and loops) governing the current control flow. Intuitively, the security stack plays a similar role as *program counters* in security type systems [47].

The semantics for the monitor is described in Figures 6. The notation $lev(st)$ works the same as for expressions, i.e., it represents the least upper bound of all levels appearing in st. Security levels are pushed on the stack by events $b(e)$, which are emitted when entering a branch or a loop iteration (rule BRANCH), and popped from the stack by events f, emitted when leaving the branch or loop iteration (rule END).

The rule FLOW ensures that programs possibly leaking information become stuck. The first premise of this rule ($lev(e) \sqsubseteq lev(x)$) prevents explicit flows, while the second one ($lev(st) \sqsubseteq lev(x)$) prevents implicit flows. The soundness of this monitor is proved in the full version of [42].

Permissiveness and transparency w.r.t. monitoring We will show that the transformation in Figure 3 is exactly as permissive as the monitor in Figure 6. In other words, any terminating monitored execution of a given program has an equivalent terminating execution on its transformed version, and any program that gets stuck in the monitor does so in the transformed version as well. To prove such a result, we need a series of lemmas. First we define a relationship between a reference map and a monitor stack. This relationship establishes that a reference map has exactly the write references of variables that a monitor would allow updates to, given a stack of security levels.

Definition 5. *Given a security stack st and a reference map ρ, $st \sim_0 \rho$ if and only if for all $\bar{x}$ in $\mathrm{dom}(\rho)$ it holds that*

$$\rho(\bar{x}) = \begin{cases} \rho(x) & \text{if } lev(st) \sqsubseteq lev(x) \\ \texttt{null} & \text{otherwise.} \end{cases}$$

We extend the definition of $\sim_0$ to give a relationship between a monitor stack and all reference maps on a program stack.

Definition 6. *Given a call stack $S = \rho_0 : \cdots : \rho_n$ and a security stack $st = \ell_1 : \cdots : \ell_n$, where $n \geq 0$, then $st \sim S$ if and only if*

$$\forall i = 0, \ldots, n-1 \; : \; (\ell_{i+1} : \cdots : \ell_n) \sim_0 \rho_i$$

and $\rho_n(\bar{x}) = \rho_n(x)$ for all x.

The intention of this definition is to provide a relation between a monitor and program stacks, so that when a reference map is popped of the program stack, it will be related by $\sim_0$ to the monitor stack if one element after popped of it as well.

The next theorem states two things simultaneously, that our transformation based enforcement is at least as permissive as enforcement through a run-time monitor, and that the enforcement is *transparent*. Transparency means that programs that the enforcement admits maintain their correct semantics, i.e., that the resulting memory after termination is the same for monitored runs and runs of the transformed program. For this theorem we need to add the transformation rule $\vdash end \rightsquigarrow \urcorner$ for technical reasons.

Theorem 2 (Correspondence with monitor)**.** *Consider a command c and its transformed version $\hat{c}$, such that $P \vdash c \rightsquigarrow \hat{c}$. Let ρ be an aliasing safe map for the variables of c, and $\hat{\rho}$ be the union of ρ and P with additional write references for every variable. Assume that st and S are a monitor stack and a call stack, respectively, such that $st \sim \hat{\rho} : S$. For any (possibly partial) execution of a monitored program*

$$\langle c \,|\, \rho, \mu; st \rangle \rightarrow^* \langle c' \,|\, \rho, \mu'; st' \rangle,$$

there is a corresponding execution of the transformed one

$$\langle \hat{c} \,|\, \hat{\rho}, \mu, S \rangle \rightarrow^* \langle \hat{c}' \,|\, \hat{\rho}', \hat{\mu}', S' \rangle,$$

such that the following conditions hold.

i) $P' \vdash c' \rightsquigarrow \hat{c}'$ *with* $P' \subseteq P$,
ii) $st' \sim \hat{\rho}' : S'$, *and*
iii) $\mu' = \hat{\mu}'$.

We note that the important special case of the theorem is when considering terminating traces, i.e., ones where $c' = \hat{c}' = \texttt{skip}$. The next theorem provides similar correspondence, but now for configurations that are blocked by the monitor.

Theorem 3 (Correspondence on blocked runs)**.** *Consider a command c, different from* skip*, and its transformed version $\hat{c}$, such that $P \vdash c \rightsquigarrow \hat{c}$. Let ρ be an aliasing safe map for the variables of c, and $\hat{\rho}$ be the union of ρ and P with additional write references for every variable. Assume st and S are monitor and program stacks, respectively, and together with $\hat{\rho}$ are such that $st \sim \hat{\rho} : S$. If the monitored execution from $\langle c \,|\, \rho, \mu; st \rangle$ is blocked by the monitor, then the execution of $\langle \hat{c} \,|\, \hat{\rho}, \mu, S \rangle$ gets stuck.*

Together, Theorems 2 and 3 give the equivalence of the two enforcement methods.

Corollary 1. *Consider a command c and its transformed version $\hat{c}$, such that $P \vdash c \rightsquigarrow \hat{c}$. Let ρ be an aliasing safe map for the variables of c, and $\hat{\rho}$ be the union of ρ and P with additional write references for every variable. Then it holds that*

$$\langle c \,|\, \rho, \mu; \epsilon \rangle \rightarrow^* \langle \texttt{skip} \,|\, \rho, \mu'; st' \rangle$$
$$\textit{iff} \;\; \langle \hat{c} \,|\, \hat{\rho}, \mu, \epsilon \rangle \rightarrow^* \langle \texttt{skip} \,|\, \hat{\rho}', \mu', \epsilon \rangle,$$

Transparency w.r.t. the original program While the above corollary states transparency with respect to the monitor, most useful is the transparency with respect to the original program. Because the only possibility of affecting the semantics by the monitor is by blocking execution, it is straightforward to show (e.g., [5]) that the monitored execution, when it terminates, does not alter the program semantics. It then follows directly that if a transformed program terminates, it must preserve the semantics of the original program as well.

Corollary 2 (Transparency)**.** *Let c, $\hat{c}$ and P be as before. Assume ρ is an aliasing safe map and $\hat{\rho} = \rho, P$. Let μ be a memory. If the transformed program $\hat{c}$ terminates,*

$$\langle \hat{c} \,|\, \hat{\rho}, \mu, S \rangle \rightarrow^* \langle \texttt{skip} \,|\, \hat{\rho}, \mu', S \rangle,$$

then so does the original,

$$\langle c \,|\, \rho, \mu, \epsilon \rangle \rightarrow^* \langle \texttt{skip} \,|\, \rho, \mu'', \epsilon \rangle,$$

and their results agree $\mu' = \mu''$.

5. Caja-based experiments

We have manually experimented with translating a simple imperative subset of JavaScript to a capability safe subset. The latter is a subset called *Core Cajita* and is defined and proven to be capability safe in [27]. Cajita itself is a subset of JavaScript which forms the basis of Caja, an isolation system for web mashups [34].

In Core Cajita, a program consists of a set of modules, represented by a top-level function. The body of a module must adhere to rules restricting which constructs are used. These rules ensure that a module cannot gain capabilities (references to objects) other than those given to it by its caller. Furthermore, all property access and modification must be done through the Cajita runtime functions `getPub` and `setPub`, which act as a reference monitor. When invoking a module, the caller can *freeze* objects before passing their reference to another module, in which case the callee module has read-only access to that object enforced by the `getPub`/`setPub` functions.

The notational noise of introducing `getPub` and `setPub` may be avoided by rewriting to full Caja. However, since only Core Cajita has been proven capability safe, where direct property access is disallowed, we have chosen it as our target language. Another future candidate could be Secure EcmaScript (SES) [15].

```
if (h && !l) {
  h = false;
} else {
  l = false;
}
```

```
function p1(H,L) {
  p11(H,L);
}
function p2(H,L) {
  p21(H,L);
}

if (getPub(H, 'h')
    && !getPub(L, 'l')) {
  p1(H, deepSnapshot(L));
} else {
  p2(H, deepSnapshot(L));
}
```

Figure 7. Translating a conditional

The translation works very much like the formal translation described in Figure 3, where modules correspond to procedures. We partition variables into two levels, *high* and *low*, and store them as properties of two objects, `H` and `L`. When calling modules which represent code running inside conditionals and loops with high guards, or one that assigns the value of a high expression, we pass a reference to `H` unmodified, but only a *snapshot* (a frozen copy) of `L`. This differs a little from the formal translation, where we send `null`-references. There however, the procedure arguments are only write references, while here they are objects used for both reading and writing. Sending a frozen object makes the low variables read-only, which is exactly the intention with sending a `null` write reference in the formal translation.

We should note that both Caja and EcmaScript 5 provide only *shallow* snapshotting and freezing of objects, meaning that properties on nested objects are still writable. In our translation, we assume the existence of a `deepSnapshot` function, which constructs a deep snapshot recursively. Writing such a function is non-trivial, e.g. since variables captured by lexical scoping cannot be extracted from function values. Creating deep snapshots thus requires some restrictions on what objects can be stored in the variables `H` and `L`, which we do not discuss here. There are other subtle issues here, for example that high properties of low objects will become read-only in high contexts, meaning that the enforcement is not fully transparent compared with a monitor. Solutions exist, but are out of scope for the current discussion. Since the experiments with Cajita are meant as proof-of-concept, we leave those loose ends for future work.

In the following examples, assume that `h` and `l` are high and low variables, respectively. To translate an assignment such as `l = l + h;` we choose a fresh name `p` and output the following module,

```
function p(H,L) {
  var tmp = getPub(L, 'l') + getPub(H, 'h');
  cajita.setPub(L, 'l', tmp);
}
```

and rewrite the assignment as

```
p(H, deepSnapshot(L));
```

since the expression being evaluated is high. For expressions that only contain low variables, we simply pass `L` unmodified.

Rewriting conditional blocks similarly follows the formal rewriting. For example, an `if`-statement translates to two Cajita modules, as shown in Figure 7. `p11` and `p21` referenced in the modules are the modules generated by translating the two assignments in the if branches.

We observe that we can make a couple of optimizations without breaking the enforcement of noninterference. First, since we know at the time of transformation if a direct assignment results in an explicit leak, we do not need to create a separate module (i.e., function) to compute the expression value. Instead, if the level of the expression is higher than the level of the variable assigned, we can simply insert a statement that raises an error. In this case, we only create modules for blocks of conditional statements and loops.

Second, we can avoid unnecessary copying of the `L` object when it is already frozen, by replacing `deepSnapshot(L)` in the above with

```
(isFrozen(L) ? L : deepSnapshot(L))
```

where `isFrozen` comes from the Cajita runtime.

ECMAScript 5 [14] has native support for freezing objects. However, capability safety is not provided in general, so a translation to Caja/Cajita is still needed to ensure code cannot access the `L` object unless it is explicitly passed to it. We leave as future work how the above approach can be extended to use only ECMAScript 5 features without compromising the guarantees that the capability model provides.

Figure 8 shows a side-by-side comparison of an (optimized) transformation. The program on the left is a simple loan calculator, taking several inputs (prefixed with `in_`). Of the inputs, the principal amount and a boolean stating if the customer already has a loan with the current bank, are considered confidential information. The outputs of the calculation is a monthly payment, shown to the user, and statistics intended to be collected by the web-application (prefixed by `stats_`). The statistics are considered public outputs, and must be independent of the confidential inputs.

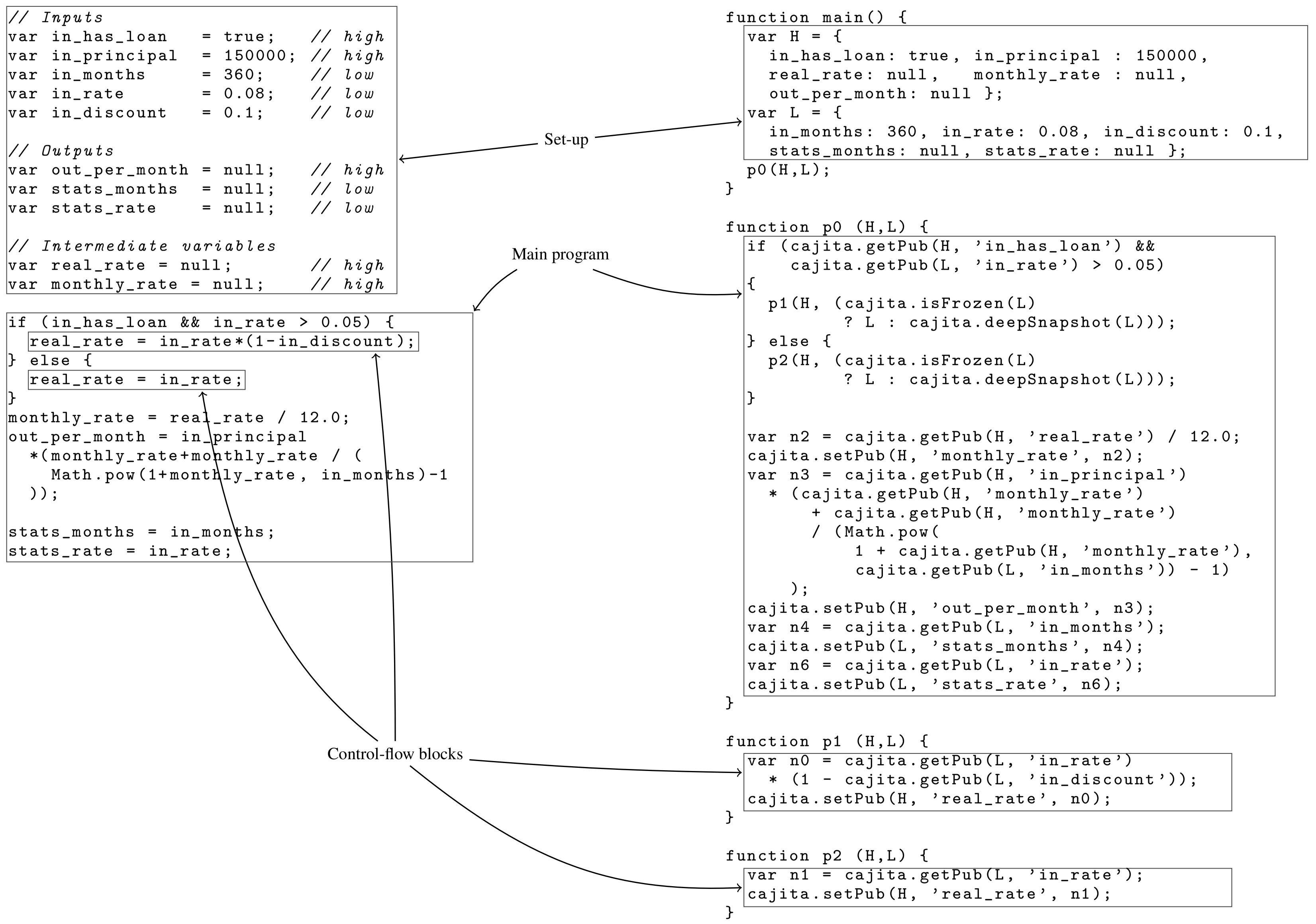
This is enforced by the transformed program. For example, if one were to update the statistic output variables inside one of the conditional branches, the rewritten program would fail as the procedures `p1` and `p2` are passed read-only snapshots of the `L` store. If the original program contained an explicit assignment, say assigning `real_rate` to `stats_rate`, this assignment would not be included in the transformed program, and instead replaced by an error statement.

The translation of the set-up code is simply a specification of a policy. Here we have chosen to simply indicate the level of variables with a comment, but one could imagine other ways of policy specification.

In a program rewritten with the optimized rewriting rules, extra method calls are only generated for control blocks and snapshotting is only performed when necessary. Therefore we expect the performance overhead to be reasonable for code where enforcing an information flow policy is critical. Indeed, our initial experiments show that a rewritten program runs as fast or only marginally slower than a program without enforcement, although experiments with a more involved case study are required to state definitive results.

6. Related work

Information flow vs. access control McLean [30, 31] scrutinizes the relation between information-flow and access-control policies. He observes that the former are expressed in terms of dependencies between inputs and outputs whereas the latter are expressed in terms of privileges of principals to perform critical operations. Schneider [44] and Hamlen et al. [20] discuss noninterference as a typical example of a policy that is not a safety property, in contrast to access control. Our paper demonstrates that dependency policies can be securely approximated by reference-based access control: we let segments of code play the role of principals and enforce secure information flow by restricting operations with publicly visible side-effects for appropriate code segments. Such an enforcement does not demand new classes of expressive power: the expressive power of our technique is exactly the same as that of online reference monitors.



```
// Inputs
var in_has_loan   = true;   // high
var in_principal  = 150000; // high
var in_months     = 360;    // low
var in_rate       = 0.08;   // low
var in_discount   = 0.1;    // low

// Outputs
var out_per_month = null;   // high
var stats_months  = null;   // low
var stats_rate    = null;   // low

// Intermediate variables
var real_rate = null;       // high
var monthly_rate = null;    // high
```

```
if (in_has_loan && in_rate > 0.05) {
  real_rate = in_rate*(1-in_discount);
} else {
  real_rate = in_rate;
}
monthly_rate = real_rate / 12.0;
out_per_month = in_principal
  *(monthly_rate+monthly_rate / (
    Math.pow(1+monthly_rate, in_months)-1
  ));

stats_months = in_months;
stats_rate = in_rate;
```

```
function main() {
  var H = {
    in_has_loan: true, in_principal : 150000,
    real_rate: null,   monthly_rate : null,
    out_per_month: null };
  var L = {
    in_months: 360, in_rate: 0.08, in_discount: 0.1,
    stats_months: null, stats_rate: null };
  p0(H,L);
}

function p0 (H,L) {
  if (cajita.getPub(H, 'in_has_loan') &&
      cajita.getPub(L, 'in_rate') > 0.05)
  {
    p1(H, (cajita.isFrozen(L)
           ? L : cajita.deepSnapshot(L)));
  } else {
    p2(H, (cajita.isFrozen(L)
           ? L : cajita.deepSnapshot(L)));
  }

  var n2 = cajita.getPub(H, 'real_rate') / 12.0;
  cajita.setPub(H, 'monthly_rate', n2);
  var n3 = cajita.getPub(H, 'in_principal')
    * (cajita.getPub(H, 'monthly_rate')
        + cajita.getPub(H, 'monthly_rate')
        / (Math.pow(
            1 + cajita.getPub(H, 'monthly_rate'),
            cajita.getPub(L, 'in_months')) - 1)
      );
  cajita.setPub(H, 'out_per_month', n3);
  var n4 = cajita.getPub(L, 'in_months');
  cajita.setPub(L, 'stats_months', n4);
  var n6 = cajita.getPub(L, 'in_rate');
  cajita.setPub(L, 'stats_rate', n6);
}

function p1 (H,L) {
  var n0 = cajita.getPub(L, 'in_rate')
    * (1 - cajita.getPub(L, 'in_discount'));
  cajita.setPub(H, 'real_rate', n0);
}

function p2 (H,L) {
  var n1 = cajita.getPub(L, 'in_rate');
  cajita.setPub(H, 'real_rate', n1);
}
```

Figure 8. Original program (left) and its transformed version (right)

The work by Leroy et al. [25] and Bierman et al. [4] have the ability to perform checks when assignments are performed in order to guarantee some safety properties. In a similar manner, we can say that our transformation carefully removes some capabilities when assignments are performed in order to preserve security. However, different from those work, our transformation also relies on removing capabilities when branches are executed.

Jia and Zdancewic encodes non-interference using logic-based access control in the pure programming language AURA [23]. The main idea of the encoding is to use principals as security levels and allow access to secret information only by providing specific proofs. More specifically, a secret value of type t is encoded as a value of type **pf** $(H$ **says** $Reveal) \rightarrow t$. In that manner, the secret value of type t can be only accessed by providing a proof that principal H, representing the security level for confidential data, allows that (i.e., **pf** $(H$ **says** $Reveal)$). The proof of **pf** $(H$ **says** $Reveal)$ can be thought as the reading capability for secret values. Different from that work, we consider computations with side-effects and writing capabilities.

Capabilities As mentioned earlier, capabilities [11] originate from the area of operating systems. Capabilities are deployed by the Cambridge CAP computer, the Hydra System, StarOS, IBM System/38, the Intel iAPX423, and Amoeba [26]. In this context, capability is an unforgeable reference to a file or a system object. Access to this file or system object is only possible when possessing the capability.

Language-based capabilities have been explored in the context of the *object-capability model* [33, 35]. The object-capability model argues for objects to play the role of both subjects and objects in traditional access-control models. Capabilities guard resources at a fine-grained, language-based level. An object might not be allowed to access a particular field of another object but at the same time might be allowed to call a particular method. The object-capability model is at core of languages E [13, 33], Joe-E [32], Emily [46], W7 [38], and Caja [34].

Miller et al. [35] provide insights on what is within the reach for capabilities. They discuss examples when, using data diodes, capabilities can help guarantee the mandatory access-control discipline *no write down* in the Bell-LaPadula model. In a similar vein, Spiessens and Van Roy [45] spell out the examples from [35] in a more abstract setting.

With the goal to enforce information confinement, Jaradin et al. [21] extend an Oz-based capability language with *membranes*, execution contexts associated with a set of token values. It would be interesting to see what semantic properties can be guaranteed by membranes.

Maffeis et al. [27] appear to be the first to investigate semantic guarantees offered by pure capabilities. In particular, they formalize the principles from the object-capability literature "only connectivity begets connectivity" and "no authority amplification" using programming-language semantics. They prove that these principles are guaranteed for a class of object-capability languages, including a Caja-based subset of JavaScript.

Information flow Information-flow control is a lively area of research [41]. A number of alternatives are possible for enforcing information-flow policies. Popular choices include static analysis [10, 36, 47], where leaks are prevented at compile time and dynamic analysis [18, 24, 42], where leaks are blocked at runtime. However, significant program analysis machinery needs to be built in order to track information flow whenever the language is not trivial.

Our approach can be viewed as a form of an inlining transformation, where the enforcement mechanism is embedded in the code itself. Inlining transformations for information flow receive increasing attention.

Chudnov and Naumann [7] present an inlining approach to monitoring information flow. They inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [39]. Flow sensitivity allows security levels of variables to change over time. The soundness of the inlined monitor is ensured by bisimulation of the inlined monitor and the original monitor.

Magazinius et al. [29] investigate on-the-fly inlining of a dynamic information-flow monitor. Dynamic code evaluation is handled by transforming the code on the fly. The monitor is based on *no sensitive upgrade* by Austin and Flanagan [1] that allows a form of flow sensitivity.

Our target programs incorporate dynamic enforcement that is flow-insensitive, and thus not comparable in permissiveness to the above two approaches [39]. As we discussed earlier, its permissiveness is the same as for dynamic monitoring [42]. However, we can easily imagine extensions of the transformation to correspond to dynamic disciplines such as no sensitive upgrade.

Devriese and Piessens [12] consider enforcing noninterference by running multiple runs of the program, one for each security level. Secret inputs are replaced with dummy values in a public run. A secret run may use values computed by public runs but not vice versa. Whenever the original program satisfies noninterference, the semantics of the multi-run is unchanged. An advantage of this approach is its language-independence: it is sufficient to focus on channeling inputs and outputs to the copies of the right security level. For permissiveness, not modifying the semantics of programs that satisfy noninterference is a benefit. However, from an error-reporting point of view, it might be a disadvantage that insecurities result in silent behavior modification instead of alarms. In this light, our approach is less permissive, but instead whenever a run of a transformed program terminates normally, it is guaranteed that this run is both secure and unmodified with respect to the original program.

7. Conclusions

We have showed that capability languages are suitable for enforcing information-flow policies. We grant capabilities to write to a variable only if the write operation is free of explicit and implicit flows. This opens up a connection between clean information-flow policies and practical enforcement based on capabilities.

Our work is a first step in the area, intended to invite further efforts of the community. As mentioned earlier, we have intentionally stripped down our language setting to the simplest possible, in order to pin down the essence of the problem: information flow can be tracked by controlling capabilities of code segments.

Although the present formalization is based on controlling capabilities to write operations, we note that controlling read operations provides further alternatives. When dealing with explicit flows, an alternative to restricting write operations based on the security level of the right-hand side of an assignment, is to instead restrict read capabilities. Indeed, explicit flows can be prevented by not allowing to read from variables whose security levels are not lower or equal to the level of the variable being assigned. This corresponds to the *no read up* (or *simple security*) in the Bell-LaPadula model [3]. It is however less natural to control implicit flows by read capabilities: when branching on an expression and thus, in effect, performing a read of the expression, it is too early in the execution to decide whether the read is allowed because its security consequences depend on the side effects (writes) in the branches. This motivates focusing on write operations for implicit flows (explicit flows are tracked via write operations for uniformity).

We expect our approach to naturally scale to more complex languages. The core of the transformation is extracting control-flow regions and handing out to these regions appropriate write capabilities, depending on the guards at branching points. Given a control-flow analyzer for a language, it can be used to build the control-flow graph of a given program. From the control-flow graph, we extract the "interesting" segments of the program: assignments with sensitive variables on the right-hand side, as well as segments whose reachability depends on sensitive data: the branches of conditionals with sensitive conditions and bodies of loops with sensitive guards. The rest of the job of the transformation is to create procedures for these segments and pass to them capabilities to write only to sensitive variables.

Performance is obviously a concern. As hinted earlier, the transformation can be optimized to avoid creation of unnecessary procedures. For example, when the guard of a conditional is public, there is no need to create a procedure for the branches. Further, we see benefits in decoupling the underlying static analysis from the actual transformation. In this scenario, the programmer writes code in a capability language, and our static analysis checks for information flows. Placing more burden on the programmer, this avoids breaking the program up into unnecessarily small units.

Support for declassification is vital because many programs need to release information as a part of legitimate functionality. Capabilities are a natural fit for expressing declassification policies. Suppose a declassification policy allows releasing the average salary of a company's employees. This is modeled by creating a procedure that calculates the average and stores into a public variable. Such a declassification-based procedure is exempt from transformation: it is passed to the result of transformation unchanged. The resulting code can use this procedure in public computation.

We do not anticipate obstacles in making our approach *flow-sensitive*, with the possibility of security levels of variables to change over time. Disciplines like *no sensitive upgrade* [1] and *permissive upgrade* [2] help steer clear of security pitfalls with dynamic flow-sensitive information-flow enforcement [39].

We remark that results dual to the ones in this paper are feasible: information-flow enforcement can be used for enforcing capability properties. To prevent implicit flows, the program counter mechanism for information-flow control ensures that no variables that are below or incomparable with a given security level are updated in sensitive context. By manipulating the program counter appropriately, we can tune information-flow enforcement to enforce capability properties.

Future work is focused on language extensions and declassification support, as outlined above. We also plan to advance our experiments to the area of securing web mashups. With lattice-based decentralized policies [28] on the policy side and Caja on the enforcement side, we plan further case studies with security-critical mashups.

Our approach is hybrid. At transformation time, it is guaranteed that the control-flow regions are carefully recorded and assignments are rewritten to respect explicit and implicit flows. At runtime, we rely on the capability sets to preserve secure information flow. Since the transformation is custom-made and the capabilities are standard, we aim at pushing the line to rely more on capabilities and less on the transformation. For example, when processing a conditional that branches on secret, we no longer need to transform the branches because they will not receive capabilities to update public data. Further, there are aspect-oriented alternatives to rewrite the assignments.

Acknowledgments

Thanks are due to Úlfar Erlingsson, Mark Miller, and the Google Caja team for inspiring discussions. This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR. Arnar Birgisson is a recipient of the Google Europe Fellowship in Computer Security, and this research is supported in part by this Google Fellowship.

References

[1] 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.

[2] 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.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[4] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *In Proc. International Conference of Functional Programming*, pages 99–110. ACM Press, 2003.

[5] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying Facets of Information Integrity. In *Information Systems Security: 6th International Conference, ICISS 2010*, volume 6503 of *LNCS*, pages 48–65. Springer-Verlag, 2010.

[6] A. Birgisson, A. Russo, and A. Sabelfeld. Capabilities for information flow. Technical report, Chalmers University of Technology, Apr. 2011. Located at `http://www.cse.chalmers.se/~russo/flowcaps-tr.pdf`.

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

[8] 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.

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

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

[11] J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3):143–155, Mar. 1966.

[12] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[13] The E language. `http://erights.org/elang/`.

[14] ECMA International. Standard ECMA-262, 5th edition, 2009.

[15] Secure ecmascript. `http://wiki.ecmascript.org/doku.php?id=ses:ses`, 2009.

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

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

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

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

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

[21] Y. Jaradin, F. Spiessens, and P. V. Roy. Capability confinement by membranes. Technical Report RR2005-03, Universit catholique de Louvain, 2005.

[22] Y. Jaradin, F. Spiessens, and P. V. Roy. SCOLL: A language for safe capability based collaboration. Technical report, Universit catholique de Louvain, 2005.

[23] L. Jia and S. Zdancewic. Encoding information flow in aura. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09. ACM, 2009.

[24] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Sci-*

ence Conference (ASIAN'06), volume 4435 of *LNCS*. Springer-Verlag, 2006.

[25] X. Leroy and F. Rouaix. Security properties of typed applets. In *In Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403. ACM Press, 1999.

[26] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proceedings of IEEE Security and Privacy'10*. IEEE, 2010.

[28] 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.

[29] 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.

[30] J. McLean. Security models and information flow. In *Proc. IEEE Symp. on Security and Privacy*, pages 180–187, May 1990.

[31] J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, Jan. 1990.

[32] A. Mettler and D. Wagner. The Joe-E language specification (draft). Technical report, U.C. Berkeley, 2006.

[33] M. Miller. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

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

[35] M. Miller, K. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University, 2003.

[36] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[37] Programmable web. `http://programmableweb.com`.

[38] J. A. Rees. A security kernel based on the lambda-calculus. Technical report, Massachusetts Institute of Technology, 1996.

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

[40] P. D. Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joose. Security of web mashups: a survey. In *Nordic Conference in Secure IT Systems*, LNCS, 2010.

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

[42] 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.

[43] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, Jan. 2009.

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

[45] F. Spiessens and P. Van Roy. A practical formal model for safety analysis in capability-based systems. In *Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 248–278. Springer-Verlag, 2005.

[46] M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *C5*, pages 163–169, 2007.

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

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

Appendix: Selected proofs

We include the proof of security. For the proofs of correspondence with dynamic monitors, the interested reader is referred to a technical report [6] containing full proofs of all theorems.

Proof of Lemma 3. We prove this by induction on the original program c. (i.e. where $\hat{c}$ is the translation of c). First let $\hat{\rho}, \hat{\mu}$ and S. The case for $c = \texttt{skip}$ is trivial.

Case $c = x := e$: Then, $\hat{c} = \texttt{call}\, p'(\vec{W}_{\uparrow \ell(e)})$ where $p' \mapsto \texttt{proc}\ (\vec{W}')\ \bar{x} := e$. The first steps of the trace are

$$\begin{aligned}\langle \texttt{call}\, p(\vec{W}_{\uparrow\ell}) \,|\, \hat{\rho}, \mu, S\rangle &\rightarrow \langle \texttt{call}\, p'(\vec{W}_{\uparrow lev(e)}; \Lsh \,|\, \rho_1, \mu, \hat{\rho} :: S\rangle \\ &\rightarrow \langle x := e; \Lsh; \Lsh \,|\, \rho_2, \mu, \rho_1 :: \hat{\rho} :: S\rangle.\end{aligned}$$

Here ρ_1 is identical to $\hat{\rho}$ except that $\rho_1(\hat{y})$ equals the `null`-location for all y with $\ell \not\sqsubseteq lev(y)$. Likewise, ρ_2 is identical to ρ_1 but with some write references set to `null`. At this point, if $\ell \sqsubseteq lev(x)$ the execution continues by updating the location referred to by $\bar{x}$ (and thus x), but then the theorem is vacuously true. In case of the opposite, execution becomes stuck since $\bar{x}$ is a `null`-reference.

The three remaining cases, for sequential composition, *if* and *while* statements, follow easily by induction. □

Proof of Theorem 1. We proceed by induction on the syntax of the original program c. The case $c = \texttt{skip}$ is trivial.

Case $c = x := e$: It is easy to see that the execution of $\hat{c}$ at most modifies locations referenced by x (through $\hat{x}$). Thus, if $lev(x) \not\sqsubseteq \ell$, there is nothing more to prove. If $lev(x) \sqsubseteq \ell$, then we observe that $\hat{c} = \texttt{call}\, p(\vec{W}_{\uparrow lev(e)})$. If $lev(e) \sqsubseteq \ell$, then its evaluation yields the same value over $\hat{\mu}_1$ and $\hat{\mu}_2$ so the theorem holds. If $lev(e) \not\sqsubseteq \ell$, then Lemma 3 gives that evaluation of $\hat{c}$ does not alter locations that are not above or equal to $lev(e)$, which in particular includes all levels below (and equal to) ℓ, and the theorem holds.

Case $c = c_1; c_2$: Follows directly by induction.

Case $c = \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$: In $\hat{c}$, the two branches are represented by procedures p_1, p_2, whose bodies are transformed versions of c_1, c_2. The bodies thus satisfy noninterference by induction. In that case, the only way for the resulting memories to differ on low values after executing $\hat{c}$ from $\hat{\mu}_1$ and $\hat{\mu}_2$, is if in each execution a separate branch is taken. If $lev(e) \sqsubseteq \ell$, then the evaluation of e is the same in both executions, so the same branch will be executed. If however $lev(e) \not\sqsubseteq \ell$, then we note that each branch consists of a call, passing $\vec{W}_{\uparrow lev(e)}$ as write capabilities. In this case, Lemma 3 again gives that neither branch will modify locations of levels not above $lev(e)$. This includes the level ℓ and all levels below it.

Case $c = \texttt{while}\ e\ \texttt{do}\ c$: This case is in principle the same as the previous case, since we do not consider non-terminating executions. At the start of each iteration there is a branching point, in which both executions agree if the level of e is below ℓ. Otherwise, Lemma 3 is again used to show that the body does not alter low parts of the memories. □

Appendix: JavaScript example

In this appendix, Figure 10 includes the full (non-optimized) rewriting of the loan calculator example from Section 5.

The rewritten programs refer to a Cajita runtime. To test the programs in an ECMAScript 5 compatible interpreter, such as Node.js (`http://nodejs.org/`), one may use the mock runtime in Figure 9, which simulates the necessary API by using ECMAScript 5 features. Note that this mock runtime only serves the purpose of our examples, and is not general. In particular, the deep snapshot function only works within one interpreter (frame) on JSON objects, i.e. objects that are primitive values or non-prototypical instances of `Object` or `Array`.

```
var cajita = (function () {

  function getPub(obj, prop) {
    return obj[prop];
  }

  function setPub(obj, prop, val) {
    obj[prop] = val;
  }

  function deepSnapshot(obj) {
      if (!(obj instanceof Object)) return obj;
      var o = {};
        for (prop in obj) {
            o[prop] = deepSnapshot(obj[prop]);
        }
        return Object.freeze(o);
    }

    return {
      getPub: getPub,
      setPub: setPub,
      deepSnapshot: deepSnapshot
    };
})();
```

Figure 9. A mock Cajita runtime that uses ECMAScript 5 features to freeze objects

```
function main() {
  var H = {
    // Inputs
    in_has_loan   : true,
    in_principal  : 150000,

    // Intermediaries
    real_rate     : null,
    monthly_rate  : null,

    // Outputs
    out_per_month : null
  };
  var L = {
    // Inputs
    in_months     : 360,
    in_rate       : 0.08,
    in_discount   : 0.1,

    // Outputs
    stats_months  : null,
    stats_rate    : null
  };

  p0(H,L);
  return {H: H, L: L};
}

function p0 (H,L) {

  if (cajita.getPub(H, 'in_has_loan') &&
      cajita.getPub(L, 'in_rate') > 0.05)
  {
    p1(H, cajita.deepSnapshot(L));
  } else {
    p2(H, cajita.deepSnapshot(L));
  }

  p0_1(H,cajita.deepSnapshot(L));
  p0_2(H,cajita.deepSnapshot(L));

  p0_3(H,L);

  p0_4(H,L);
}

function p1 (H,L) {
  p1_1(H,L);
}

function p2 (H,L) {
  p2_1(H,L);
}

function p1_1 (H,L) {
  var n0 = cajita.getPub(L, 'in_rate')
    * (1.0 - cajita.getPub(L, 'in_discount'));
  cajita.setPub(H, 'real_rate', n0);
}

function p2_1 (H,L) {
  var n1 = cajita.getPub(L, 'in_rate');
  cajita.setPub(H, 'real_rate', n1);
}

function p0_1 (H,L) {
  var n2 = cajita.getPub(H, 'real_rate') / 12.0;
  cajita.setPub(H, 'monthly_rate', n2);
}

function p0_2(H,L) {
  var n3 = cajita.getPub(H, 'in_principal')
    * (cajita.getPub(H, 'monthly_rate')
        + cajita.getPub(H, 'monthly_rate')
        / (Math.pow(
            1 + cajita.getPub(H, 'monthly_rate'),
            cajita.getPub(L, 'in_months')) - 1)
      );
  cajita.setPub(H, 'out_per_month', n3);
}

function p0_3(H,L) {
  var n4 = cajita.getPub(L, 'in_months');
  cajita.setPub(L, 'stats_months', n4);
}

function p0_4(H,L) {
  var n5 = cajita.getPub(L, 'in_rate');
  cajita.setPub(L, 'stats_rate', n5);
}

// Run e.g. in Node.js
var rv = main();
console.log(
    "Pmt per month: " + rv.H.out_per_month
  + " (rate " + 100*rv.H.real_rate + "%)");
console.log(
    "Statistics: " + rv.L.stats_months
  + " months at " + 100*rv.L.stats_rate + "%");
```

Figure 10. Fully (i.e. non-optimized) rewritten program from Section 5. Compare with Figure 8.