Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing

Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld

Chalmers University of Technology, 412 96 Gothenburg, Sweden

Abstract. Tracking information flow in dynamic languages remains an open challenge. It might seem natural to address the challenge by runtime monitoring. However, there are well-known fundamental limits of dynamic flow-sensitive tracking of information flow, where paths *not* taken in a given execution contribute to information leaks. This paper shows how to overcome the permissiveness limit for dynamic analysis by a novel use of testing. We start with a program supervised by an information-flow monitor. The security of the execution is guaranteed by the monitor. Testing boosts the permissiveness of the monitor by discovering paths where the monitor raises security exceptions. Upon discovering a security error, the program is modified by injecting an annotation that prevents the same security exception on the next run of the program. The elegance of the approach is that it is sound no matter how much coverage is provided by the testing. Further, we show that when the mechanism has discovered the necessary annotations, then we have an accuracy guarantee: the results of monitoring a program are at least as accurate as flow-sensitive static analysis. We illustrate our approach for a simple imperative language with records and exceptions. Our experiments with the QuickCheck tool indicate that random testing accurately discovers annotations for a collection of scenarios with rich information flows.

1 Introduction

In a dynamically loaded *web mashup* that involves sensitive information from several parties, how do we prevent information leakage? A web mashup consolidates independent web services, potentially by mutually distrusting providers, into an integrated web service. For example, a web mashup to display the location of secret objects (say vehicles collecting cash from ATMs) might make use of a map service (such as Google Maps) for enhanced visualization. The map service code needs access to the secrets in order to display them. At the same time, the map service needs access to its servers to load new map components on demand. How do we ensure that the map service does not leak secrets back to its servers?

The state of the art in web mashup security [24] leaves the question open. A range of approaches from separation to full integration has been suggested, tailored to web mashup scenarios such as online ads, where access-control policies

S. Foresti, M. Yung, and F. Martinelli (Eds.): ESORICS 2012, LNCS 7459, pp. 55–72, 2012.

are sufficient. However, the problem of tracking information in mashups after access has been granted remains largely unsolved. Of particular challenge is handling the dynamic nature of programming languages like JavaScript that manipulate information in web mashups.

With the above scenario as our long-term motivation, the goal of this paper is a practical mechanism for tracking information flow in dynamic languages.

It might seem natural to address dynamic languages with dynamic analysis. Dynamic enforcement of secure information flow can be done similarly to dynamic type checking: values are decorated with labels representing the security of each value, and for each operation the labels are checked at runtime. The data labels may change over time, which means that the analysis is *flow-sensitive*.

Flow-sensitive enforcement is one where an assignment such as $x := e$ propagates the security level of the expression e to the variable x. On the other hand, a flow-insensitive system assigns security levels that do not change. Such a system disallows the assignment if the level of the expression is not at least as restrictive as the level of the variable. Flow-sensitivity allows an information-flow policy to be specified in terms of sources, where information enters the system, and sinks, where information exits the system, rather than on syntactic variables inside the program. This frees the programmer from explicitly managing security levels of local variables. Since variables can be reused for different purposes, flow-sensitivity also has the potential of accepting more programs that are secure.

However, there are well-known fundamental limits of dynamic flow-sensitive tracking of information flow [9,29,5,22]. Flow-sensitivity introduces a channel for leaking information though the labels themselves, which is possible to exploit even though labels may not be observable in the language.

Consider the program in Figure 1, assuming *secret* to hold 0 or 1 initially. The program copies *secret* into *public*. However, a purely dynamic monitor faces challenges to detect this flow. Indeed, when *secret* is 1, then *public* is never accessed after branching on *secret*. When *secret* is 0, then the assignment of 0 to *public* takes place inside of a conditional that branches on a variable *temp* that has not been touched since its initialization. In both cases, the problem is the branches that are *not* executed, which are missed by purely dynamic analysis. In general, it is not possible to have sound dynamic flow-sensitive information-flow enforcement that is strictly more permissive than flow-sensitive static analysis [22].

```
public = 1; temp = 0;
if (secret) temp = 1;
if (!temp) public = 0;
```

Fig. 1. Flow-sensitivity attack

This implies that a purely dynamic information-flow monitor must be either unsound (i.e., there are *false negatives*) or imprecise (i.e., there are *false positives* that are accepted by static analysis). In this design space, the *no-sensitive-upgrade* [32,2] discipline shows how to achieve soundness. This discipline states that the original label of a variable under assignment must be taken into account, and if it is not at least as restrictive as the level of the control-flow context, upgrading its level is disallowed. With this discipline, the program above is

stopped if it reaches the assignment to *temp* because it attempts an upgrade in the *secret* control-flow context.

Hence, no-sensitive-upgrade provides soundness at the price of permissiveness. Of particular concern is that for programs like one in Figure 1, the permissiveness of monitoring is worse than that of static analysis. Indeed, flow-sensitive static analysis [12] is able to detect the flow in the program above, and ensure that both *temp* and *public* become secret after the conditional. On the other hand, secure programs with flow-sensitive manipulation of dynamic data structures are out of reach for static analysis, implying that many interesting programs are rejected due to the crude approximation by static analysis. This means that neither static nor dynamic analysis as is provide a satisfactory solution to the problem of false positives.

This paper shows how to achieve the best of the two worlds without resorting to full-scale static analysis. We overcome the permissiveness limit for dynamic analysis by a novel use of testing. We show that testing boosts the permissiveness of dynamic information-flow enforcement by discovering places in code for automatic injection of upgrade annotations. Upon discovering a security error, the program is modified by injecting an annotation that prevents the same security exception on the next run of the program. Further, we show that when the mechanism has discovered the necessary annotations, then we have an accuracy guarantee: the results of monitoring a program are at least as accurate as flow-sensitive static analysis. The process leads to a program that is never blocked by the monitor because sensitives upgrades have been "tested away". Importantly, eradicating sensitive upgrades is not at the price of unnecessarily pushing up security levels for data: we show that the levels are never pushed above what is demanded by the static approach. This allows us significant reduction of false positives while in total absence of false negatives.

The elegance of the approach is that it is sound no matter how much coverage is provided by the testing. In contrast to fuzzing or vulnerability and penetration testing, it is not the original program that is tested but its monitored counterpart. This guarantees security, thanks to the soundness of the monitor. As discussed above, we gain permissiveness in the sense that the monitor stops less programs and accuracy in the sense that the results of monitoring a program are at least as accurate as flow-sensitive static analysis.

We illustrate our approach for a simple imperative language with references and exceptions. Our experiments with the random testing tool QuickCheck [7] indicate that random testing accurately discovers annotations for a collection of scenarios with rich information flows. We are able to further enhance the permissiveness by *delayed upgrades*, which records the reference to be upgraded but does not perform the actual upgrade until just before entering sensitive context.

We envision that our method can be applied most productively during the software development and testing phase, when our approach can help discovering upgrade annotations before the code is shipped.

2 Background

The dynamic features of languages like JavaScript offer on their own a compelling argument for dynamic information-flow enforcement. In addition, independent of language features, functionality provided by the execution environment may pose challenges for static analyses. Consider, for instance, the API provided by the DOM [11] in combination with Google maps. When creating a new Google map we need to pass the part of the page where the map should be drawn. Typically, this is done by assigning an id to the element and fetching it with getElementById as illustrated below.

```
<script type="text/javascript" label="google">
  new google.maps.Map(document.getElementById("map_canvas"));
  ...
<div id="map_canvas" label="google"></div>
<form>
```

From an information-flow perspective we want to enforce that the Google code is only allowed send back the parts of the page labeled 'google'. This entails that the analysis must treat getElementById differently depending on which element is fetched (something which cannot be statically decided in general — in particular since the page may be dynamically changing). For a dynamic analysis this poses no problem, since the elements are tagged with their labels.

Speaking more generally, dynamic analysis has the ability to handle data with dynamic structure, e.g., heaps, with high precision. Consider the following example, where l_1 and l_2 are aliases.

```
l1 = new {}; l1.f = 1; l2 = l1; l2.f = h;
```

A flow-sensitive static analysis must take the alias into account and update the type of both l_1, and l_2. In general aliasing is not decidable, and the program would be rejected by static analyses like Jif [19]. Dynamic analyses do not have this problem, since the label of f is stored with the value of f.

However, as shown in the introduction, dynamic enforcement of secure information flow has fundamental limits for flow-sensitivity under secret control. *Secret control* or *secret context* refers to the commands inside conditionals and loops with guards that contain secrets. Promising steps in the direction of overcoming these limits are *privatization operations* [3] or *upgrade* [10] commands that enable the upgrade of labels before entering secret contexts. This work makes use of upgrade commands for the security levels of values (`upg`), the structure of heap objects (`upgs`) and exceptions (`upge`), all explained below.

Values. Consider the following example, where the public variable l is assigned to under secret control. This causes the monitor to block on a sensitive upgrade.

```
if (h) l = 1;
```

By inserting an upgrade that upgrades the label of l before the execution of the conditional we make sure that execution is not stopped.

```
l = upg(l,secret); if (h) l = 1;
```

Structure. If structured data, like records, is changed under secret control the structure of the data may encode secrets. In general, the security labels associated with the different parts of structured data might not be enough to model the security level of the structure. The reason for this is that not only the presence of certain data may encode secrets but also the absence, and it's not necessarily the case that the security level of the absence of data can be read from the security level of the presence of other data. In such cases, if the absence is visible to the program, the security model of the structured data must be extended to model absence.

Using records as an example, consider for instance the following program, where the field f is added to o depending on the secret h.

```
o = new {}; if (h) o.f = 1;
```

Following the general explanation above, after execution, the presence or absence of f encodes the value of h. In the case h is true the field f will be present, and the fact that its presence is secret is recorded in the security label of the value of f. However, in the case h is false the field f will not be present and its absence encodes information about h. Since the absence of fields is visible via record projection, as is explained in Section 3, records are equipped with a structure label. The structure label of records can be understood as an upper bound of the context in which the record may have been modified, or in the terms of absent fields as the upper bound on the security level of the non-existence of the absent fields.

Returning to the example above, o has public structure, which causes the execution of the secret conditional of the example to be stopped — adding a field would require upgrade of the structure label under secret control. In order to allow for the addition, the structure of the record can be upgraded before the secret context.

```
o = new {}; upgs(o,secret); if (h) o.f = 1;
```

Exceptions. Exceptions pose a significant challenge for secure information flow due to the non-local transfer of control. In the example below the value of h is copied into l.

```
try { if (h) throw; l = 0; } catch { l = 1; }
```

The standard static solution to this is to type commands following a potential exception in a secret context as under secret control [20,18]. Since the majority of commands in languages like JavaScript can cause exceptions and due to the possibility of non-local transfer of control this can cause a significant amount of code to be typed under secret control. Following [10] we adopt a more permissive discipline and introduce a special exception label that tracks the level at which exceptions are allowed to be thrown. Initially, the exception label is public, which allows the body of the `try` above to execute in public context (in the case $h = 1$ the monitor will stop with a security violation). To allow for exceptions in secret contexts the language provides an upgrade, which can be inserted before the secret context as follows.

```
try { l = upg(l, secret); upge(secret);
      if (h) throw; l = 0; }
catch { l = 1; }
```

This upgrade causes the subsequent commands of the try, and of the handler to be considered to be a secret context. After the try, the exception label is once again lowered.

Manual upgrade annotations open the possibility of improving the permissiveness of the monitored program. However, they come at a price of placing the heavy annotation burden on the programmer. It forces the programmer to be aware of the monitor the programs will run under. As this is undesirable, and sometimes impossible (e.g., with legacy code), our goal is to fully relieve the programmer from the annotation burden. With the background set, we proceed to describe a method that applies testing for automatically discovering and injecting upgrade instructions to boost the permissiveness of the monitor.

3 Monitor and Rewriting

This section introduces the language — a simple JavaScript-inspired language with records and exceptions, its monitor semantics, which is essentially a distilled version of the monitor of [10], and establishes the soundness of the monitor.

3.1 Syntax and Semantics

Figure 2 shows the syntax of expressions and commands, as well as supporting structures. Values v consist of strings, numbers, a special value `undefined`, together with the pointers. Records are maps from values to values, and the heap μ is a partial map from pointers to records. A reference is a pair of a pointer and a value, referring to a particular field in a record.

Values stored in records, as well as the components of a reference, are decorated with a security label σ; the structure label of records is written after the semicolon inside the curly braces.

As is common [8,31] we assume that the labels form a predefined lattice, and do not consider the case where labels are not know a priori or where the structure of the lattice can be modified dynamically. Without loss of generality, we will use a simple two-level lattice described by $\mathsf{public} \sqsubseteq \mathsf{secret}$, where $\sqsubseteq$ denotes the lattice order. Let $\sqcup$ and $\sqcap$ denote least upper bound, and greatest lower bound, and let $\bot$ and $\top$ denote public and secret labels.

Expressions consist of literals for the primitive values, variables, projections of records, and pure binary operators. Lefthand sides make up a subset of expressions that can be assigned to, and will evaluate to references. Righthand sides of assignments can be expressions or record allocations, optionally annotated with an explicit upgrade of the value or structure label.

The commands are standard, apart from the `upge` command, which upgrades the current exception label. Variables represent string-keyed fields in a distinguished record $\mu(0)$. This record, referred to as the *global record*, is in line with how variables are handled in JavaScript and simplifies the semantics.

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & n \mid \text{`}s\text{'} \mid x \mid e[e] \mid \texttt{undefined} \mid e * e \\
\text{Pointers} & p & \in & \mathbb{N}_0 \\
\text{Values} & v, w & ::= & n \mid \text{`}s\text{'} \mid \texttt{undefined} \mid p \\
\text{References} & \rho & ::= & (p^\sigma, v^\sigma) \\
\text{Lefthand sides} & l & ::= & x \mid l[e] \\
\text{Righthand sides} & r & ::= & e \mid \texttt{new}\ \{\} \mid \texttt{upg}(r, \sigma) \mid \texttt{upgs}(r, \sigma) \\
\text{Records} & o & ::= & \{v \mapsto v^\sigma, \ldots, v \mapsto v^\sigma; \sigma\} \\
\text{Heap} & \mu & : & \text{Pointers} \hookrightarrow \text{Records} \\
\text{Commands} & c & ::= & \texttt{skip} \mid l := r \mid c\,;c \mid \texttt{upge}(\sigma) \\
 & & \mid & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{try}\ c\ \texttt{catch}\ c \mid \texttt{throw}
\end{array}
$$

Fig. 2. Notation and syntax

Evaluation and dereferencing is detailed in Figure 3. We write $[\![\cdot]\!]_\mu$ for the evaluation of expressions and lefthand sides in a heap μ. This evaluation returns either a labeled value or a reference and is free of side-effects. Dereferencing a reference further resolves it to a value by looking it up in the heap. The dereferenced value has a label that takes into account also the security labels of the expressions used to build the reference itself. This ensures that values that are reached via secret pointers have a secret label. Dereferencing is written $(\cdot)^*_\mu$, and for convenience we define it for values as identity and write $[\![e]\!]^*_\mu$ instead of $([\![e]\!]_\mu)^*_\mu$.

Dereferencing a non-existing field succeeds with the `undefined` value. This means that the existence of fields can be probed by record projection.

Note that evaluation and dereferencing are not total functions. In particular, the expression $e_1[e_2]$ is not valid if $[\![e_1]\!]^*$ is not a pointer value. In our formalization such cases cause the evaluation to get stuck, while in practice they might result in throwing reference exceptions.

Let E denote *environments*, consisting of pairs of a heap and an exception label. We define the semantics of the language and the monitor as a big step relation $\rightarrow$. An initial configuration $\langle\, c \mid pc, E \,\rangle$ consists of a command c, a security label pc, and an environment E. The label pc represents the level of the current control context, and is updated by conditional branches and iteration.

The relation $\rightarrow$ relates an initial configuration to an execution result, if one exists. A terminating execution of a command c in an environment E may result in one of the following: (i) In the case of successful termination, the term $\mathsf{Ok}\ E'$, where E' is the resulting environment. (ii) In the case of an uncaught exception, the term $\mathsf{Throw}\ E'$, where E' is then environment in which the exception was thrown. (iii) A security stop $\mathsf{Stop}(t, \sigma)$, where t is either a reference to a field (p, w), the term $\mathsf{struct}(p)$ representing the structure label of a record, or $\mathsf{exception}$ representing the runtime exception level.

A stop indicates that the program has reached a point, where the corresponding entity requires at least the security level σ for the monitor to be sound. For instance, attempting to write to a public field under secret control results in a $\mathsf{Stop}((p, w), \top)$, where (p, w) identifies the field, and $\top$ signifies that the field must be at least $\top$ for the write to be accepted. Similarly, attempting to add a field to a record with public structure under secret control results in $\mathsf{Stop}(\mathsf{struct}(p), \top)$, where p identifies the record that must be have secret structure for the addition to

$$[\![n]\!]_\mu = n^\bot \qquad [\![s]\!]_\mu = s^\bot \qquad [\![\texttt{undefined}]\!]_\mu = \texttt{undefined}^\bot \qquad [\![x]\!]_\mu = (0^\bot, x'^\bot)$$
$$[\![e_1 * e_2]\!] = (v_1 * v_2)^{\sigma_1 \sqcup \sigma_2} \quad \text{where } v_1^{\sigma_1} = [\![e_1]\!]^* \text{ and } v_2^{\sigma_2} = [\![e_2]\!]^*$$
$$[\![e_1[e_2]]\!]_\mu = (p^{\sigma_1}, v^{\sigma_2}) \quad \text{where } p^{\sigma_1} = [\![e_1]\!]^*_\mu, \text{ and } v^{\sigma_2} = [\![e_2]\!]^*_\mu$$

$$(v^\sigma)^*_\mu = v^\sigma \qquad (p^{\sigma_p}, w^{\sigma_w})^*_\mu = \begin{cases} v^{\sigma_p \sqcup \sigma_w \sqcup \sigma_v} & \text{if } \mu(p) = \{\ldots, w \mapsto v^{\sigma_v}, \ldots; \sigma_s\} \\ \texttt{undefined}^{\sigma_p \sqcup \sigma_w \sqcup \sigma_s} & \text{otherwise and } \mu(p) = \{\ldots; \sigma_s\} \end{cases}$$

Fig. 3. Evaluation and dereferencing

be accepted. Finally, attempting to throw an exception under secret control with a public exception level results is $\mathsf{Stop}(\mathsf{exception}, \top)$, indicating that the exception level must be $\top$ for the exception to be accepted. From such a stop and its corresponding execution tree, we determine a location in the source program where an explicit upgrade command needs to be inserted to avoid that particular stop. This process is described in Section 4.

Unlike expressions, the evaluation of righthand sides can have side-effects, and we use the same relation notation $\rightarrow$ as for commands for the evaluation of righthand sides. Evaluation of a configuration $\langle\, r \mid pc, E \,\rangle$ can result in a labeled value and an updated environment $\mathsf{Ok}(v^\sigma, E')$, or a security stop $\mathsf{Stop}(t, \sigma)$, which carries the same meaning as above. Evaluation of a righthand side can never throw an exception.

For space reasons, the full set of inference rules defining the semantics can be found in the full version of this paper [4]. To give the reader an insight into the monitor, we exemplify with the rule for successful execution of the internal #put operator, which handles record updates.

$$\text{PUT}\ \frac{\begin{array}{c}[\![l]\!]_\mu = (p^{\sigma_p}, w^{\sigma_w}) \qquad \mu(p) = \{\ldots, w \mapsto v_0^{\sigma_0}, \ldots; \sigma_s\} \\ (pc \sqcup \epsilon \sqcup \sigma_p) \sqcap \sigma_w \sqsubseteq \sigma_s \qquad pc \sqcup \epsilon \sqcup \sigma_p \sqcup \sigma_w \sqsubseteq \sigma_0 \\ o' = \mu(p)[w \mapsto v^{\sigma_v \sqcup pc \sqcup \epsilon \sqcup \sigma_p \sqcup \sigma_w}; \sigma_s \sqcup \sigma_w]\end{array}}{\langle\, \#\texttt{put}(l, v^{\sigma_v}) \mid pc, \mu, \epsilon \,\rangle \rightarrow \mathsf{Ok}(\mu[p \mapsto o'], \epsilon)}$$

$\#\texttt{put}(l, v)$ is an internal command that performs the writing part of assignments, writing a value v to a field represented by the lefthand side l. The evaluation is split into three cases: one succeeding and two stopping. To allow the update we require that σ_0, the previous label of the value, is above the control context, as well as above the combined labels of the reference from l. In addition, since writing with a secret key can affect the structure, the key's security label σ_w must be added to the structure label of the record. For this reason we demand that if σ_w is secret then either $pc \sqcup \epsilon \sqcup \sigma_p$ is public, or the structure label of the record σ_s is secret. These conditions ensure that the label of the value is independent of secrets. When they are satisfied, the record is updated with the new labeled value, its label raised to include the control context and the reference labels. The cases where the conditions are not satisfied correspond to the two stopping cases: one demanding the upgrade of the value of the field, and one demanding the upgrade of the structure label.

3.2 Soundness

As is common [31], we use *termination-insensitive noninterference* (TINI) as our semantic security condition. TINI offers the possibility of liberal enforcement well suited for dynamic monitors, while only allowing low-bandwidth leaks. Like other typical semantic security conditions TINI is undecidable.

Noninterference can be stated as the preservation of a family of low-equivalence relations under execution. For languages with heaps, the family is indexed over a *bijection* on low-reachable pointers ensuring that the low-reachable parts of low-equivalent heaps are isomorphic. Low-equivalence guarantees that low-reachable public values are equal — for the secret parts no demands are made. In addition it guarantees that the labeling is independent of secrets. For space reasons, the low-equivalence relation $\sim$ can be found in the full version of this paper [4].

TINI states that successful execution in low-equivalent environments results in low-equivalent environments. Let $\mathcal{C}$ denote any non-Stop configuration.

Theorem 1 (TINI). *For any program c, β, and two heaps μ_1 and μ_2 such that $\mu_1 \sim_\beta \mu_2$, we have that if $\langle\, c \mid \bot, \mu_i, \bot \,\rangle \to \mathcal{C}_i$ for $i = 1, 2$ then there is a β' such that $\mathcal{C}_1 \sim_{\beta'} \mathcal{C}_2$.*

This means that the resulting (low-reachable) public parts of the heap are independent of secrets; whatever choice of secret values in the initial heaps, the produced results are equal in their public values. The proof of this and further theorems are contained in the full version of the paper [4].

4 Rewriting

To improve the permissiveness of the dynamic monitor, executions resulting in stops (found by, e.g., testing) are used to patch the program with explicit upgrades to prevent the stop from occurring again.

A heap is called *initial* if it contains no records other than the global record, itself containing only primitive (non-pointer) values. Let μ_0 range over initial heaps. Given a derivation tree of an execution $\langle\, c \mid \bot, \mu_0, \bot \,\rangle \to \mathsf{Stop}(t, \sigma)$, the different cases for t dictate how the program needs to be rewritten in order to prevent that particular stop.

case $t = (p, w)$**:** This stop indicates that the program attempted to assign to the field w of the record at heap location (with pointer) p, which would have resulted in upgrading its existing security level in secret context, or over a secret reference. In order to make this run succeed, the field must be explicitly upgraded.

In the case that $p = 0$, i.e., the upgrade refers to a variable in the program. The execution tree is used to see where the program entered the secret context in order to insert an upgrade command just before that point. In this case w is a string value with the name of the variable, which is converted to an identifier x and the command $x := \mathtt{upg}(x, \sigma)$ is inserted before the secret context.

```
if (h) l = 1;  ⌒  l = upg(l,secret); if (h) l = 1;
```

If $p \neq 0$ however, the reference is to a field in a record other than the global record, and may be built from a lefthand side containing arbitrary expressions. Building an upgrade command that refers to the same field at a different place in the program requires complex tracking of heap mutations. Instead of inserting an upgrade command before an enclosing conditional, the execution tree is used to find an assignment to the field in a public context and over public pointers. Such an assignment exists, because the record is not in the initial heap, and the stop indicates that the field of the record must already exist with a public label. This implies that the field was added in a public context over a public pointer. The assignment is converted to an upgrade, by wrapping its righthand side with `upg` and the label σ. This ensures that the field is labeled as secret from that point in the program.

```
o = new {}; o.f = 0; if (h) o.f = 1;  ⌒
                        o = new {}; o.f = upg(0, secret); if (h) o.f = 1;
```

case $t = \mathsf{struct}(p)$**:** This stop indicates that an upgrade of a record's structure label was needed in secret context, or over a secret pointer. Similar to the second case above, the structure of the record must be upgraded in a public context over a public pointer. The execution tree is used to find the last assignment satisfying these properties where p was the value of the assignment's righthand side, and wrap that righthand side with `upgs` and the appropriate label. Such an assignment must always exist, as the only record existing in the initial heap is the global record which always has secret structure. Hence p must point to an allocated record, and it must have been allocated in a public context—otherwise the structure of the record would already be public.

```
o = new {}; if (h) o[h] = 1;  ⌒  o = upgs(new {}); if (h) o[h] = 1;
```

case $t = \mathsf{exception}$**:** This stop is generated when the program attempts to throw an exception in a context where the exception label ϵ is not above the *pc*. To make this execution succeed, the exception label must be upgraded whether the secret branch is entered or not. The execution tree is used to determine the syntactic `if` or `while` command in which we enter secret control, and the program is patched by inserting an $\mathsf{upge}(\sigma)$ before this command.

```
if (h) throw;  ⌒  upge(secret); if (h) throw;
```

In what follows, we will refer to one step of the above process as a rewriting relation on programs. If $\langle\, c \mid \bot, \mu, \bot \,\rangle \to \mathsf{Stop}(t, \sigma)$, then we say $c \curvearrowright_\mu c'$ where the program c' is obtained by applying the above rules on the proof of the stopped execution. If $\langle\, c \mid \bot, \mu, \bot \,\rangle \to \mathcal{C}$ let $c \curvearrowright_\mu c$.

The process above describes how to rewrite the program to make one failing run succeed. Of course, there may be other failing runs so this process is iterated. Let $\mathcal{S}$ be a set of initial heaps and let $\curvearrowright_\mathcal{S}$ be a relation on programs such that $c \curvearrowright_S c'$ iff there exists a heap $\mu \in \mathcal{S}$ such that $c \curvearrowright_\mu c'$ and $c \neq c'$.

Theorem 2 (Termination). *For any set $\mathcal{S}$ of initial heaps, any sequence*

$$c_0 \curvearrowright_\mathcal{S} c_1 \curvearrowright_\mathcal{S} c_2 \curvearrowright_\mathcal{S} \cdots$$

terminates, i.e., there is an n such that $c_n \curvearrowright_\mu c_n$ for all $\mu \in \mathcal{S}$.

The theorem is straightforward, considering that the number of possible upgrade commands, as well as the number of locations they may be inserted are bounded given the rewriting procedure above in a finite lattice of security levels.

For a given set $\mathcal{S}$ of initial heaps rewriting will produce a program that the monitor will not stop when run in any of the heaps in S. A program is *non-stopping* if the monitor does not stop execution for any initial environment. Under the assumption that all values, including strings, have finite domains (which is the case in all practical settings, due to hardware limitations) rewriting can be used to find non-stopping programs.

Theorem 3. *Let $\mathcal{T}$ be the set of all initial heaps. The result of rewriting based on $\mathcal{T}$ is non-stopping, i.e., for $c \curvearrowright^*_{\mathcal{T}} c'$, it holds that c' is non-stopping.*

5 Accuracy

Consider the security labeling of the execution environment under execution. We say that a labeling is more accurate than another if is at least as permissive, and it is not more secret. In this section we establish that upgrade injection does not result in a security labeling that is less accurate than that of a standard flow-sensitive static type system; the contrary, however, is possible.

To show accuracy we adapt a standard flow-sensitive information-flow type system [18,19] to the language in Figure 2 and establish its soundness. For space reasons the development of the type system and its soundness can be found in the full version of this paper [4].

The type language consists of two different types: *primitive* types, and *record* types. Primitive types are security labels or security labeled record type *names*. The use of names to make recursive record types inductive is common practice, and their meaning in terms of record types is given by a map ρ from record type names C to record types. Finally, record types are maps from values to primitive types. Let $\Gamma, \Delta ::= (C, \epsilon)$ denote environment types and exception environment types respectively, where C is the type of the global record, and ϵ is the exception level.

The type judgments for commands are of the form $pc, \Gamma_1 \vdash_\Delta c \Rightarrow \Gamma_2$. The judgment is read: the command c is well-typed in security context pc, environment type Γ_1 and exception environment type Δ yielding environment type Γ_2. The intuition is that if c is run in environments that correspond to Γ_1 the result will correspond to either Γ_2 or Δ depending on whether the execution was successful or resulted in an exception.

With this we can formulate the accuracy result: rewriting executions of well-typed programs results in well-typed programs w.r.t. the same entry and exit environment types.

Theorem 4 (Accuracy). *For any program c_1 and initial heap μ such that $\bot, C, \bot \vdash_\Delta c_1 \Rightarrow \Gamma$ and $\delta \vdash \mu : C, \bot$ we have that if $c_1 \curvearrowright_\mu c_2$ then $\bot, C, \bot \vdash_\Delta c_2 \Rightarrow \Gamma$.*

The result implies that the rewriting process produces programs that will be at least as accurate as a standard static type system. In many cases the upgrade injection together with dynamic monitoring will be more accurate. This is due to the possibility of flow-sensitive heap entities, the presence of dead code, or the possibility of value dependent labels as an example in the next section will show.

6 Implementation

We have implemented the monitor from Section 3 in Haskell. The implementation uses QuickCheck [7] to generate random initial heaps and perform the iterative process of finding stopping executions and automatically injecting upgrade commands into the input program.

When the monitor encounters the situation that an upgrade is needed but the control-flow context, the exception label or the reference used does not allow it, it stops the execution and conveys this information back to the test runner. The test runner uses this, together with an execution trace collected during the run, to determine a syntactic location in the original program where an upgrade command is inserted.

QuickCheck uses *generators* to perform random testing of Haskell code, by generating test cases and checking if user-supplied *properties* hold for it. Our implementation allows for descriptions of generators of initial heaps, where both existence, value and labeling of initial variables can be randomized. The monitor is then tested against the property that running a given program does not result in a security stop. When QuickCheck finds a stopping case, the test harness rewrites the program and restarts the testing process.

Our experiments have shown that performing this iterative process yields a rewritten program where enough upgrades have been inserted so that no initial heap results in a stopped execution. Below we present some of the more illustrative experiments which run using an initial heap description that labels `h` as secret boolean (i.e., a number with values 0 or 1) and `l` as public.

Experiment 1: Consider the example of Section 1; the implementation discovers the stopped runs where `t` and `l` are upgraded in secret context, and inserts the needed upgrades immediately before each conditional. The resulting program is shown in Figure 4, where the two upgrades have been inserted where secret context may be entered.

```
l = 1; t = 0;
t = upg(t,secret);
if (h)  t = 1;
l = upg(l,secret);
if (!t) l = 0;
```

Fig. 4. Consistent labeling

Experiment 2: As described in previous sections, the existence of a field may encode secret information. For this reason the monitor tracks the security level of the structure of a record. Thus the program `o = new {}; if (h) o[0] = 1;` is stopped by the monitor, and the rewriter turns this stop into the program shown in Figure 5. Adding the `upgs` makes adding a field in secret context safe, since any later projections of non-existing fields will be labeled as secrets.

Experiment 3: When writing to a field, it is not sufficient to consider only the control context to determine if its value or the structure of the containing record. The choice of field and record, which is written to, may depend on secret information in the lefthand side used to refer to it. This is reflected in the security labels of the reference built from the lefthand side, and is taken into account when updating the record. Consider the following program.

```
o = upgs(new {}, secret);
if (h) o[0] = 1;
```

Fig. 5. Secret structure

```
o = new {}; o[0] = 0; o[1] = 0; o[h] = 1;
```

First a new record is allocated and initialized to contain two zero-valued fields with keys 0 and 1 resp. Thereafter, one of the fields is modified depending on the secret value `h`. The assignment would label the modified field as secret, but this would constitute an upgrade which itself depends on the value of `h`. Thus, the implementation stops the assignment. Since h is a secret number with values 0 or 1 both fields (but not the structure) will be upgraded, resulting in the following program.

```
o = new {}; o[0] = upg(0, secret); o[1] = upg(0, secret); o[h] = 1;
```

Experiment 4: The rewriter is also able to inject upgrades of the exception label. Recall the program from Section 2, which attempts to leak h through the use of exceptions. The implementation detects this and inserts an upgrade of the exception label before entering secret context. This alone is not enough to make the program run, since this upgrade now makes the assignments to l be under secret control (recall that the exception label is considered part of the control context). Thus, another iteration of rewriting is required to upgrade the variable l itself as well. The result is shown in Figure 6.

```
try {
  l = upg(l, secret);
  upge(secret);
  if (h) throw;
  l = 0;
} catch { l = 1; }
```

Fig. 6. Throw under secret control

Experiment 5: When a variable needs to be upgraded, the upgrade is inserted at the closest point in the program, where the context is strictly lower than the target level. For lattices with more than two levels there is a risk that this upgrade will trigger another stop, since the label of the value of the variable may be lower than the label of the context at this point. This is intentional; instead of moving the upgrade up, the stop is allowed to trigger another rewrite in the next iteration. This results in a stepwise upgrade of the variable with the possibility of a more accurate labeling.

Consider the left program of Figure 7, in which the variables `pub`, `cls` and `sec` have corresponding security labels from a lattice with public $\sqsubseteq$ classifed $\sqsubseteq$ secret. Here, the last assignment requires `x` to be upgraded to secret. If this is done at the assignment `x = 0`, then the runs where `cls` is true will unnecessarily force `cls` to be upgraded as well. However, `x` cannot be directly upgraded from public to secret in the else branch, because that upgrade would be under classified

```
x = 0;
if (cls) {
    if (x) cls = x;
} else {
    if (sec) x = sec;
}
```

```
x = 0;
x = upg(x, classified);
if (cls) {
    if (x) cls = x;
} else {
    x = upg(x, secret);
    if (sec) x = sec;
}
```

Fig. 7. Cascading upgrades

control. This in turn creates an upgrade of x to classified before entering the outer if-command. The resulting program is shown on the right in Figure 7.

It is worth noting that this example improves on the precision of a static type system. As seen from an observer at the classified level, it is a safely visible decision which branch is taken in the outer conditional, but that decision depends on the value. Standard type-systems for information flow are not value-sensitive, and infers that x needs to be secret because of the potential assignment in the else-branch. In a dynamic setting however, there is no need to upgrade x further than to classified if that branch is not taken.

Delayed upgrades. Upgrading a record field at the point of its last public assignment may be premature. For example, consider the following program.

```
o = new {}; o[0] = 1; x = o[0]; if (h) o[0] = 42;
```

Labeling `o[0]` with secret right in the public assignment to it will unnecessarily cause the variable `x` to have a secret value as well. It is therefore too early to upgrade `o[0]` before entering the secret control context. Instead, the upgrade should be inserted before the conditional. However, note that `o[0]` may be any lefthand side, involving arbitrary expressions, and it may not even be the same one in both assignments. To build a syntactic lefthand side that refers to the same field as `o[0]` at a different program point is not possible in general.

Instead, the implementation uses a technique that avoids premature upgrading via *delayed upgrades*. We insert the upgrade command in the last public assignment, including a program label which refers to the conditional command where it should actually be upgraded. The semantics of such a delayed upgrade command resolves the righthand side to a reference and stores it along with the label `L1` in a list of pending upgrades. An actual upgrade of the reference is only performed just before, and if, a command with that label is reached. If the labeled command appears in a conditional block itself, the field in question is not even upgraded at all if that command is never reached. We note that the stepwise upgrading seen in Figure 7 extends to non-variables also when delayed upgrades are enabled.

```
o = new {};
o[0] = upg(1, secret, L1);
x = o[0]; // still public
L1: if (h) o[0] = 42;
```

Fig. 8. Delayed upgrade

7 Related Work

A large body of work targets language-based methods for information-flow security [25]. We discuss dynamic methods for information-flow enforcement, which are most closely related to the focus of this paper. For a general survey of dynamic information-flow techniques, we refer to Le Guernic's thesis [15].

Fenton [9] discusses purely dynamic monitoring for information flow but does not prove noninterference. Volpano [30] considers a purely dynamic monitor to prevent explicit (but not implicit) flows. Languages like Perl and PHP support *taint mode* to dynamically track explicit flows.

Shroff et al. [27] discuss a purely dynamic monitor that in addition to tracking explicit flows, provides limited support to discovering implicit flows. The monitor is based on recording dependencies discovered at runtime and propagating them to subsequent runs of the code. While this method does not guarantee noninterference, it fits a scenario of tracking common flows in a trusted application.

In a flow-insensitive setting, Sabelfeld and Russo [26] show that a monitor similar to Fenton's enforces termination-insensitive noninterference without losing in precision to classical static information-flow checkers. This line of work has progressed further to extend the monitor to a language with dynamic code evaluation, communication, and declassification [1], as well as timeout instructions [21]. Further, Russo et al. [23] investigate the impact of dynamic tree structures like the DOM on information flow. Their monitor prevents attacks based on navigating and deleting DOM tree nodes. The monitor derives the security level of presence for each node from the context of its creation. It keeps invariants such as the presence level of a parent may not exceed the presence level of a child.

As discussed earlier, Austin and Flanagan [2,3] suggest a purely dynamic monitor for information flow with a limited form of flow sensitivity. They discuss two disciplines: *no sensitive-upgrade*, where the execution gets stuck on an attempt to assign to a public variable in secret context, and *permissive-upgrade*, where on an attempt to assign to a public variable in secret context, the public variable is marked as one that cannot be branched on later in the execution. Austin and Flanagan [3] discuss inserting *privatization operations*, which are akin to our upgrade commands. The insertion takes place when a variable that was previously upgraded in secret context is about to be branched upon.

Stefan et al. [28] present a library for dynamic information-flow control in Haskell using a notion of *floating labels*, related to the concept of program counter, to restrain the side effects of computations. Even though they do not allow labels of references (c.f. variables) to change, their primitives allow for the manipulation of labels that causes related problems. Their solution to this is to demand the programmer to annotate the program, which is comparable to the use of upgrades. Magazinius et al. [16] show how to inline a no-sensitive upgrade monitor into programs in a language with dynamic code evaluation.

Russo and Sabelfeld [22] show that purely dynamic flow-sensitive monitors do not subsume the permissiveness of flow-sensitive security type systems. They

also provide a framework for hybrid monitors that allows expressing a range of hybrid monitors as one by Le Guernic et al. [14].

Hedin and Sabelfeld [10] propose dynamic information-flow control for a core of JavaScript that includes objects, higher-order functions, exceptions, and dynamic code evaluation. They discuss the usefulness of upgrade annotations but do not provide methods to generate them. Our paper shows how to relieve the programmer from the burden of upgrade annotations, making dynamic information-flow control more practical.

Chugh et al. [6] present a hybrid approach to handling dynamic execution. Their work is staged where a dynamic residual is statically computed in the first stage, and checked at runtime in the second stage.

Masri et al. [17] develop a method for detecting and debugging information flows for restricted Java bytecode (no exceptions, multithreading, or exit statements). The method is a form of dynamic program slicing that allows detecting explicit flows. They also show that static analysis and a preprocessing transformation can be used to include implicit flows into consideration.

Kang et al. [13] consider taint analysis for implicit flows in trusted code. They enhance a purely dynamic analysis to propagate selected information about control-flow dependencies, hitting a middle ground between ignoring implicit flows and propagating taint along all control dependencies indiscriminately.

Compared to the previous work, a key novelty of this paper is the usage of testing (rather than static analysis) to boost the permissiveness of dynamic enforcement.

8 Conclusion

While dynamic information-flow enforcement might seem to be a natural fit for tackling languages with dynamic data structures, there are fundamental limits of permissiveness of purely dynamic techniques. This paper demonstrates how to overcome these limits by testing. We show that testing boosts the permissiveness of dynamic information-flow enforcement by discovering places in code for automatic injection of upgrade annotations. The inference of upgrade annotations ensures that the dynamic analysis is more permissive than the static counterpart, without losing soundness. Our experiments with the QuickCheck tool suggest that we achieve the permissiveness of hybrid monitors without static analysis on a collection of scenarios with rich information flows.

Future work includes extending the formalization with functions (which we have already implemented in our prototype). The upgrade injection mechanism allows setting the upgrades before functions are called, which enables smooth integration with third-party libraries. Based on the prototype reported in Section 6 and our approach to tackling the core JavaScript features [10], we pursue the implementation of information-flow monitor enhanced with upgrade instruction injection for the full JavaScript language.

Acknowledgments. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR. Arnar Birgisson is a recipient of the Google Europe Fellowship in Computer Security, and this research was supported in part by this Google Fellowship.

References

1. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proc. IEEE Computer Security Foundations Symposium (July 2009)
2. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS) (June 2009)
3. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS) (June 2010)
4. Birgisson, A., Hedin, D., Sabelfeld, A.: Boosting the permissiveness of dynamic information-flow tracking by testing (June 2012) (full version), `http://www.hvergi.net/arnar/publications/pdf/testing-full.pdf`
5. Cavallaro, L., Saxena, P., Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 143–163. Springer, Heidelberg (2008)
6. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (2009)
7. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Proc. ACM International Conference on Functional Programming, pp. 268–279 (2000)
8. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Comm. of the ACM 20(7), 504–513 (1977)
9. Fenton, J.S.: Memoryless subsystems. Computing J. 17(2), 143–147 (1974)
10. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: Proc. IEEE Computer Security Foundations Symposium (June 2012)
11. Hors, A.L., Hegaret, P.L.: Document Object Model Level 3 Core Specification. Tech. rep., The World Wide Web Consortium (2004)
12. Hunt, S., Sands, D.: On flow-sensitive security types. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 79–90 (2006)
13. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. Network and Distributed System Security Symposium (February 2011)
14. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-Based Confidentiality Monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)
15. Le Guernic, G.: Confidentiality Enforcement Using Dynamic Information Flow Analyses. Ph.D. thesis, Kansas State University (2007)
16. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly Inlining of Dynamic Security Monitors. In: Rannenberg, K., Varadharajan, V., Weber, C. (eds.) SEC 2010. IFIP AICT, vol. 330, pp. 173–186. Springer, Heidelberg (2010)

17. Masri, W., Podgurski, A., Leon, D.: Detecting and debugging insecure information flows. In: Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE), pp. 198–209 (2004)
18. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 228–241 (January 1999)
19. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow (July 2001), `http://www.cs.cornell.edu/jif`
20. Pottier, F., Simonet, V.: Information flow inference for ML. ACM TOPLAS 25(1), 117–158 (2003)
21. Russo, A., Sabelfeld, A.: Securing timeout instructions in web applications. In: Proc. IEEE Computer Security Foundations Symposium (July 2009)
22. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proc. IEEE Computer Security Foundations Symposium (July 2010)
23. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking Information Flow in Dynamic Tree Structures. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 86–103. Springer, Heidelberg (2009)
24. De Ryck, P., Decat, M., Desmet, L., Piessens, F., Joosen, W.: Security of Web Mashups: A Survey. In: Aura, T., Järvinen, K., Nyberg, K. (eds.) NordSec 2010. LNCS, vol. 7127, pp. 223–238. Springer, Heidelberg (2012)
25. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (2003)
26. Sabelfeld, A., Russo, A.: From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 352–365. Springer, Heidelberg (2010)
27. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: Proc. IEEE Computer Security Foundations Symposium, pp. 203–217 (July 2007)
28. Stefan, D., Russo, A., Mitchell, J., Mazières, D.: Flexible dynamic information flow control in haskell. In: Proceedings of the 4th ACM Symposium on Haskell, pp. 95–106. ACM (2011)
29. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-site scripting prevention with dynamic data tainting and static analysis. In: Proc. Network and Distributed System Security Symposium (February 2007)
30. Volpano, D.: Safety Versus Secrecy. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 303–311. Springer, Heidelberg (1999)
31. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Computer Security 4(3), 167–187 (1996)
32. Zdancewic, S.: Programming Languages for Information Security. Ph.D. thesis, Cornell University (July 2002)