An Empirical Study of Information Flows in Real-World JavaScript

Cristian-Alexandru Staicu  
TU Darmstadt

Daniel Schoepe  
Chalmers University of Technology

Musard Balliu  
KTH Royal Institute of Technology

Michael Pradel  
TU Darmstadt

Andrei Sabelfeld  
Chalmers University of Technology

1 INTRODUCTION

JavaScript is at the heart of the modern web, empowering rich client-side applications and, more recently, also server-side applications. While some language features, such as dynamism and flexibility, explain this popularity, the lack of other features, such as language-level protection and isolation mechanisms, open up a wide range of integrity, availability, and confidentiality vulnerabilities [29]. As a result, securing JavaScript applications has become a key challenge for web application security. Unfortunately, existing browser-level mechanisms, such as the same-origin policy or the content security policy, are coarse-grained, failing short to distinguish between secure and insecure manipulation of data by scripts. Furthermore, server-side applications lack such isolation mechanisms completely, allowing an attacker, e.g., to inject and execute arbitrary code that interacts with the operating system through powerful APIs [46].

An appealing approach to securing JavaScript applications is information flow analysis. This approach tracks the flow of information from untrusted sources to sinks in order to enforce application-level security policies. It can ensure both integrity, by preventing information from untrusted sources to reach trusted sinks, and confidentiality, by preventing information from secret sources to reach public sinks. For example, information flow analysis can check that no attacker-controlled data is evaluated as executable code or that secret user data is not sent to the network. Because the dynamic nature of JavaScript hinders precise static analysis, dynamic information flow analysis has received significant attention by researchers [6, 9, 11, 14, 17, 25, 27, 48]. The basic idea of dynamic information flow analysis is to attach security labels, e.g., secret (untrusted) and public (trusted), to runtime values and to propagate these labels during program execution. To simplify the presentation, we assume to have two security labels, and we say that a value is sensitive if its label is secret or untrusted; otherwise, we say that a value is insensitive.

At the language level, a program may propagate information via two kinds of information flows: explicit flows [20] occur whenever sensitive information is passed by assignment statement or into a sink. Implicit flows [20] arise via control-flow structures of programs, e.g., conditions and loops, when the flow of control depends on a sensitive value. For a dynamic information flow analysis, explicit flows can be further classified into flows that happen because a particular branch is executed, so-called observable implicit
flows [7], and flows that happen because a particular branch is not executed, so called hidden implicit flows [7].

Figure 1 illustrates the different kinds of flows with a simple JavaScript-like program that leaks sensitive information. The program has a variable `passwd`, which is marked initially as a sensitive source at line 1. Using this variable in an operation that creates a new value, e.g., in line 3, is an explicit flow. Consider the case where the password is "topSecret", i.e., the conditional at line 5 evaluates to true, and line 6 sets `gotIt` to true. At line 10, the `gotIt` variable is sent to the network through the function `sink()`, which is considered to be an insensitive sink. The flow from the password to `gotIt` is an observable implicit flow because a sensitive value determines that `gotIt` gets written. Now, consider the case where `passwd` is "abc". The branch at line 5 is not taken and the `gotIt` variable remains false. Sending this information to the network reveals that the password is different from "topSecret". This flow is a hidden implicit flow because a sensitive value determines that `gotIt` does not get written.

Ideally, an information flow analysis should consider all three kinds of flows. In fact, there exists a large body of work on static, dynamic, hybrid, and multi-execution techniques to prevent explicit and implicit flows. However, so far these tools have seen little use in practice, despite the strong security guarantees that they provide. In contrast, a lightweight form of information flow analysis called taint analysis is widely used in computer security [42]. Taint analysis is a pure data dependency analysis that only tracks explicit flows, ignoring any control flow dependencies.

The question which kinds of flows to consider is a tradeoff between costs and benefits. On the cost side, considering more flows increases false positives [32]. A false positive here means that a secure execution is conservatively blocked by an overly restrictive enforcement mechanism. A common reason is that a value gets labeled as sensitive even though it does not actually contain information that is security-relevant in practice. This problem, sometimes referred to as label creep [18, 39], reduces the permissiveness of information flow policies, because the monitor will prematurely stop a program to prevent a value with an overly sensitive label from reaching a sink. Another cost of considering more kinds of flows is an increase in runtime overhead. On the benefit side, considering more flows increases the ability to find security vulnerabilities and data leakages, i.e., the level of trust one obtains from the analysis. For example, an analysis that considers only explicit flows will miss any leakage of sensitive data that involves an implicit flow. Unfortunately, despite the large volume of research on information flow analysis, there is very little empirical evidence on the importance of the different kinds of flows in real applications.

Because of this lack of knowledge, potential users of information flow analyses cannot make an informed decision about what kind of analysis to use.

To better understand the tradeoff between costs and benefits of using a dynamic information flow analysis, this paper presents an empirical study of information flows in real-world JavaScript code. Our overall goal is to better understand the costs and benefits of dynamically analyzing explicit, observable implicit, and hidden implicit flows. Specifically, we are interested in how prevalent different kinds of flows are, what kinds of security problems may be detected when flowing subsets of flows, and what costs and benefits of analyzing all flows imposes. To address these questions, we study 56 real-world JavaScript programs in various application domains with a diverse set of security policies. The study considers integrity problems, specifically code injection vulnerabilities and denial of service vulnerabilities caused by an algorithmic complexity problem, and confidentiality problems, specifically leakages of uninitialized memory, browser fingerprinting and history sniffing. Each studied program has at least one real-world security problem that information flow analysis can detect.

Our study is enabled by a novel methodology that combines state-of-the-art dynamic information flow analysis [5, 25, 26] and program rewriting [12] with a set of novel security metrics. We implement the methodology in a dynamic information flow analysis built on top of Jalangi [43]. The implementation draws on a sound analysis for a simple core of JavaScript. The formalization relates the security metrics to semantic security conditions for taint tracking [41], observable tracking [7] and information flow monitoring [24].

The findings of our study include:

1. All three kinds of flows occur locally in real-life applications, i.e., an analysis that ignores some of them risks to miss violations of the information flow policy. Explicit flows are by far the most prevalent, and only five benchmarks contain hidden implicit flows (Section 4.1).
2. An analysis that considers explicit and observable implicit flows, but ignores hidden implicit flows, detects all vulnerabilities in our benchmarks. For most applications it is even sufficient to track explicit flows only, while for some client-side, privacy-related applications one must also consider observable implicit flows (Section 4.2).
3. Tracking hidden implicit flows causes an analysis to prematurely terminate various executions. Furthermore, we find that different monitoring strategies proposed in the literature vary significantly in their permissiveness. (Section 4.3).
4. The amount of data labeled as sensitive steadily increases during the execution of most benchmarks, confirming the label creep problem. An analysis that considers implicit flows increases the label creep by over 40% compared to an analysis that considers only explicit flows (Section 4.4).
5. The analysis overhead caused by considering implicit flows is significant: Ignoring implicit flows saves the effort of tracking runtime operations by a factor of 2.5 times (Section 4.5).

Prior work (discussed in Section 5) studies false positives caused by static analysis of implicit flows [32, 38] and the semantic strength of flows [34]. Jang et al. [28] conduct a large-scale empirical study.

```javascript
1 // variable passwd is sensitive
2 var gotIt = false;
3 var paddedPasswd = "xx" + passwd;
4 var knownPasswd = null;
5 if (paddedPasswd == "xtopSecret") {
  gotIt = true;
  knownPasswd = passwd;
}
6 // function sink is insensitive
7 sink(gotIt);
```
showing that several popular web sites use information flows to exfiltrate data about users’ behavior. Kang et al. [30] combine dynamic taint analysis with targeted implicit flow analysis, demonstrating the importance of tracking implicit flows for trusted programs. However, to the best of our knowledge, no existing work addresses the above questions.

In summary, this paper contributes the following:

- We are the first to empirically study the prevalence of explicit, observable implicit, and hidden implicit flows in real-world applications against integrity, availability, and confidentiality policies.
- We present a methodology and its implementation, which enables the study, and we provide a formal basis for empirically studying information flows (Section 3).
- We show the soundness of a core of JavaScript with respect to semantic security conditions (Appendix).
- Through realistic case studies and security policies, we provide empirical evidence that sheds light on the cost-benefit tradeoff of information analysis and that outlines directions for future work (Section 4).

We share our implementation, as well as all benchmarks and policies used for the study, to support future evaluations of information flow tools for JavaScript.2

2 BENCHMARKS AND SECURITY POLICIES

Our study is based on 56 client-side and server-side JavaScript applications, which suffer from four classes of vulnerabilities. These applications are subject to attacks that have been independently discovered by existing work, including integrity, availability, and confidentiality attacks. For every application, we define realistic security policies expressed as information flow policies. Table 1 shows the applications, along with their security policies, and size measured in lines of code. The benchmarks vary in size from tens of lines of code to tens of thousands. We further explain the policies below. For each application we either create or reuse a set of inputs that trigger the attack and other inputs to increase the coverage of different behaviors.

Our goal is an in-depth study of the different kinds of information flows for a range of security policies; we do not claim to study a representative sample of JavaScript applications. Existing in-breadth empirical studies, which analyze hundreds of thousands of web pages against fixed policies, provide clear evidence for security and privacy risks in JavaScript code [27, 33, 35]. In contrast to these large-scale studies, our effort consists in identifying vulnerable scripts from different domains and analyzing the flows therein.

Table 1: Insecure programs, security policies, program size, sensitive branch coverage and number of upgrades. ‘module’ stands for the module interface.

<table>
<thead>
<tr>
<th>Type ID Library</th>
<th>Policy</th>
<th>LoC/SRC Upgr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 fish</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>2 growl</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>3 gm</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>4 libcrypto</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>5 main-pro</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>6 modulfy</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>7 mol-proto</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>8 mongojs</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>9 m-log</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>10 mobile-ion-resizer</td>
<td>file system API</td>
<td>eval and exec</td>
</tr>
<tr>
<td>11 mongo-parse</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>12 monosemask</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>13 mongui</td>
<td>HTTP API</td>
<td>eval and exec</td>
</tr>
<tr>
<td>14 mongjs-edit</td>
<td>HTTP API</td>
<td>eval and exec</td>
</tr>
<tr>
<td>15 mooseasy</td>
<td>HTTP API</td>
<td>eval and exec</td>
</tr>
<tr>
<td>16 choc: growl-reporter</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>17 gitjs</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>18 kerb_request</td>
<td>module</td>
<td>eval and exec</td>
</tr>
<tr>
<td>19 printer</td>
<td>module</td>
<td>eval and exec</td>
</tr>
</tbody>
</table>

Injection vulnerabilities on Node.js

The Node.js ecosystem has enabled a proliferation of server and desktop applications written in JavaScript. Injection vulnerabilities are programming errors that enable an attacker to inject and execute malicious code. Recent work [46] has demonstrated the devastating impact of injection vulnerabilities on server-side programs, e.g., when an attacker-controlled string reaches powerful APIs such as exec or eval. Such attacks can severely compromise integrity, e.g., deleting all files in a directory or completely controlling the attacked machine. We study 19 Node.js modules that contain injection vulnerabilities (IDs 1 to 19 in Table 1). As security policies, we consider the interface of a module as an untrusted source and the APIs that interpret strings as code, such as exec or eval, as trusted sinks.

ReDoS vulnerabilities

Regular expression Denial of Service, or ReDoS, is a form of algorithmic complexity attack that exploits the possibly long time of matching a regular expression against an attacker-controlled input. The single-threaded execution model of
JavaScript makes JavaScript-based web servers particularly susceptible to ReDoS attacks [45]. We analyze 19 web server applications that are subject to ReDoS attacks (IDs 20 to 39 in Table 1). As a security policy, we consider data received via HTTP requests as untrusted sources and regular expressions known to be vulnerable as trusted sinks.

**Buffer vulnerabilities** Buffer vulnerabilities expose memory content filled with previously used data, e.g., cryptographic keys, source code, or system information. In Node.js, such vulnerabilities occur when using the Buffer constructor without explicit initialization. Buffer vulnerabilities are similar to the infamous Heartbleed flaw in OpenSSL [22], as both allow an attacker to read more memory than intended. We analyze 7 applications subject to buffer vulnerabilities (IDs 40 to 46 in Table 1). The security policy requires that no information flows from the buffer allocation constructor to HTTP requests without initialization.

**Device fingerprinting and history sniffing** Web-based fingerprinting collects device-specific information, e.g., installed fonts or browser extensions, to identify users [2]. History sniffing attacks use the fact that browsers display links differently depending on whether the target has been visited [27, 49]. We analyze 10 client-side JavaScript applications that are subject to various forms of fingerprinting and history sniffing attacks (IDs 47 to 56 in Table 1). The security policies label as secret the sources that provide sensitive information, e.g., the font height and width, and as public sinks the APIs that enable external communication, e.g., image tags. We adapt these programs to our Node.js-based infrastructure by introducing minimal changes that emulate DOM interactions. We carefully cross-checked this adaptations in a pair-programming fashion, ensuring that all flows in the original program are preserved. The policies are application-specific and mark certain nodes in the emulated DOM as sources and sinks. In contrast to the other benchmarks, these programs can potentially be malicious [27, 37]. That is, the assumption that the analyzed code is trusted does no longer hold.


<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sec. condition</th>
<th>Tracked flows</th>
<th>Permissiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taint tracking</td>
<td>Explicit secrecy</td>
<td>✓</td>
<td>Stop when H-labeled value reaches sink.</td>
</tr>
<tr>
<td>Observable tracking</td>
<td>Observable secrecy</td>
<td>✓ ✓</td>
<td>Stop when H-labeled value reaches sink.</td>
</tr>
<tr>
<td>No Sensitive Upgrade</td>
<td>Non-interference</td>
<td>✓ ✓ ✓</td>
<td>Stop when L-labeled variable is written in sensitive context.</td>
</tr>
<tr>
<td>Permissive Upgrade</td>
<td>Non-interference</td>
<td>✓ ✓ ✓</td>
<td>Stop when partially leaked value is used.</td>
</tr>
</tbody>
</table>

**Information flow policy** The analysis checks whether data from a sensitive source influences data that arrives at an insensitive sink. The sources and sinks for a program are specified in an information flow policy, or short, policy. For integrity, the policy specifies that no information from untrusted sources (H) reaches trusted sinks (L). For confidentiality, the policy stipulates that no information from secret sources (H) reaches public sinks (L). We model sources by variables and object fields, and their security label corresponds to the label of the value that they contain initially. We denote sinks by a function sink(), which is implicitly labeled as L. Different monitoring strategies for dynamic information flow analysis address the problem of checking whether an execution violates a policy. In this work, we focus on flow-sensitive dynamic monitors, where variables can be assigned different security labels during the execution. Table 2 gives an overview of the monitoring strategies studied in this paper. Taint analysis tracks only explicit flows and stops the program only if an H-labeled value reaches a sink. In contrast to taint tracking, the other three strategies also track implicit flows. The monitors identify implicit flows by maintaining a security stack that contains all sensitive labels of expressions in conditionals that influence the control flow. When the stack is non-empty, the program executes in a sensitive context. Observable Tracking [7] tracks only explicit and observable implicit flows, but ignores hidden implicit flows. Whenever an L-labeled variable is updated in a sensitive context, observable tracking updates the label as sensitive and continues with the execution. For example, consider the following program, which is trivially secure because there is no call to sink():

```javascript
1 var location; var y; var z;
2 if (10 < location < 20) {
3  y = "Home"; }
4 //upgrade(y);
5 z = "You are at " + y;
```

Consider now an execution where the location is 15H. Observable tracking updates the labels of y and z as sensitive and does not stop the execution. The strictest monitoring strategies try to prevent also hidden implicit flows. We consider two variants of such a strategy. They both terminate the execution of the program whenever an observable implicit flow may lead to a hidden implicit flow in another execution. The No Sensitive Upgrade strategy (NSU) [4, 50] disallows...
upgrading the security labels of a variable in a sensitive context. In particular, it terminates the execution whenever such an update happens. For example, consider the execution of the above program with $y = 5^H$. The NSU strategy terminates the program at line 3 due to the update of the $L$-labeled variable $y$ in a sensitive context.

Permissive Upgrade (PU) [5] is a refinement of the NSU strategy. It labels a value as partially leaked if an $L$-labeled variable is updated in a sensitive context, and terminates the program if the updated variable is further used outside the sensitive context. Consider again the same execution of the above program. The PU strategy labels $y$ as partially leaked at line 3 because the program writes to the $L$-labeled variable in a sensitive context, and then terminates the program at line 5 because the value is used. In our work, we use the PU strategy to study the prevalence of different kinds of flows.

Upgrade statements Naively applying the PU strategy to real-world programs can be very restrictive and risks to increase the number of false positives, i.e., terminate many secure executions. To address this problem, Austin and Flanagan propose the upgrade statement [4] and the privatization statement [5]. These statements change the label of a variable to $H$ explicitly, to signal a potential hidden implicit flow to the monitor. For example, we can insert an upgrade statement before line 5 in the above example to mark $y$ as sensitive even if the branch is not taken. As a result, the program does not terminate immediately when the value is read. If the program would later call sink($y$), then the monitor would terminate the program and report a policy violation.

Permissiveness The above example illustrates the permissiveness issues of different monitoring strategies, i.e., that they terminate the program unnecessarily even though no policy violation occurs. Taint tracking and observable tracking both do not terminate the program. In contrast, both NSU and PU terminate the program unnecessarily. This overapproximation of policy violations is necessary to avoid potential hidden implicit flows. Adding upgrade statements avoids such premature termination of the program by assigning an $H$-label to $y$, independently of what branch of the conditional statement is executed. If we uncomment line 4, the execution proceeds without terminating the program unnecessarily. That is, upgrade statements may increase the permissiveness, but impose the cost of adding upgrade statements.

3.2 Security Metrics

Our approach uses program testing to measure the prevalence of different kinds of information flows. The basic idea is to test a program with an information flow monitor that implements the PU strategy, while incrementing counters that represent the number of explicit, observable implicit, and hidden implicit flows. These counters then allow us to reason about the prevalence of the different kinds of flows and about the policy violations that different monitoring strategies would detect. In contrast to the PU monitor that terminates the program when it encounters a policy violation, our monitor continues the execution to measure flows in the remainder of the execution. We refer to Section 3.3 for the formal definition of the monitor.

We consider information flows at two levels of granularity. On the one hand, we consider flows induced by a single operation in the program (Section 3.2.1). We call such flows micro flows or simply flows. Studying flows at the micro flow level is worthwhile because it provides a detailed understanding of the operations that contribute to higher-level flows. In particular, flows provide a quantitative answer to the permissiveness challenges faced by state-of-the-art dynamic monitors that implement the NSU or the PU strategy. On the other hand, we consider transitive flows of information between a source and a sink, called source-to-sink flows (Section 3.2.3). Studying flows at this coarse-grained level is worthwhile because source-to-sink flows are what security analysts are interested in when using an information flow analysis.

The metrics presented in this section measure the prevalence of flows quantitatively, and do not attempt to judge the importance of flows. To ensure that our flows represent relevant problems, our study uses real-world security problems and policies that capture these issues.

3.2.1 Micro Flows. To measure how many explicit, observable implicit, and hidden implicit flows exist, our monitor increments the counters for these micro flows as follows.

**Explicit flows** The approach counts an explicit flow for every assignment event where the written value is sensitive but the value that gets overwritten (if any) is not sensitive. The rationale is to capture program behavior where sensitive information flows to a memory location that stores insensitive information. In contrast, overwriting a sensitive value with another (in)sensitive value does not leak any new information, and therefore does not count as an explicit flow.

For example, consider this code:

```
1  var x = 3H; var y = 5H; var z;
2  x = y; // no explicit flow
3  z = x; // explicit flow
```

**Observable implicit flows** The approach counts an observable implicit flow for every assignment event that happens in a sensitive context and that overwrites an insensitive value. Similar to explicit flows, the rationale is to capture program behavior that writes sensitive information to a memory location that stores insensitive information. The main difference is that the assignment happens because of a control flow decision made based on a sensitive context. Note that it is irrelevant whether the written value is sensitive because the fact that a write happens leaks sensitive information.

For example, consider this code:

```
1  var x = trueH; var y = 3; var z;
2  if (x)
3     y = 5; // observable implicit flow
4     z = 7; // no flow
```

**Hidden implicit flows** The approach counts a hidden implicit flow for every execution of an upgrade statement of a variable containing insensitive information. The rationale is to capture assignment events that did not happen, but that could have happened during the execution if a control flow decision that depends on a sensitive value would have been different.

For example, consider this code:

```
1  var x = falseH; var y; var z;
2  if (x)
3     y = 5; // not executed, no flow
4     upgrade(y); // hidden implicit flow
5     z = y; // hidden implicit flow
```
3.2.2 Label Creep. As mentioned earlier, a common reason for false positives is label creep. Since measuring false positives would be subject to a given source-to-sink policy, we focus on measuring the prevalence of the more general phenomenon of label creep in micro flows. Recall that this concept refers to the fact that information flow analysis may quickly label a large portion of all values handled in a program as sensitive. In most of the cases, this leads to an explosion in false positives that in turn reduces the usefulness of the analysis. We propose a novel metric called Label Creep Ratio (LCR) to assess how many variables and object fields in memory involved in each source-to-sink flow. If the code locations of two more, to count the number of unique source-to-sink flows that ing what micro flows contribute to a source-to-sink flow. Further-
source-to-sink flows different monitoring strategies detect by track-
explicit micro flow and then gets passed to the sink.

For a given monitoring strategy, the Label Creep Ratio is the ra-
tio between the number of assignments of $H$-labeled values and
the total number of assignments. Intuitively, measuring the LCR
throughout an execution estimates the speed at which the memory
locations get assigned sensitive labels.

3.2.3 Source-to-sink Flows. To what degree do different kinds of
flows contribute to policy violations? To address this question, we
consider transitive flows from a source of sensitive information to
a sink of insensitive information. For instance, none of the flows in
the examples above correspond to a source-to-sink flow, since no
sink statement is present.

Now, consider the code:

```javascript
1 var x = false; var y; var z;
2 if (x)
3   y = 5;
4 upgrade(y); // hidden micro flow
5 z = x;    // explicit micro flow
6 sink(y);  // source-to-sink flow
```

The program contains two micro flows and one source-to-sink
flow. However, if the execution is analyzed with taint tracking or
observable tracking, the source-to-sink flow is missed, because it
occurs only due to the upgrade statement.

As another example, consider the following code:

```javascript
1 var x = true; var y; var z;
2 if (x)
3   y = 5; // observable flow
4 z = x;  // explicit flow
5 sink(y+z); // source-to-sink flow
```

The source-to-sink flow will be detected by all three kinds of
monitoring strategies, because the variable $z$ gets labeled $H$ via an
explicit micro flow and then gets passed to the sink.

As illustrated by these two examples, we measure how many
source-to-sink flows different monitoring strategies detect by track-
ing what micro flows contribute to a source-to-sink flow. Furthermore,
to count the number of unique source-to-sink flows that a monitor
detects, we compute the set of source code locations involved in
each source-to-sink flow. If the code locations of two
source-to-sink flows are the same, we count them as only one
unique flow. This corresponds to the way a human security analyst
would inspect warnings produced by an analysis.

3.2.4 Inference of Upgrade Statements. The approach described
so far requires a program that indicates hidden implicit flows
through upgrade statements. To obtain such a program, we adapt a
testing-based technique for automatically inserting upgrade state-
ments [12]. The basic idea is to repeatedly execute the program with
a particular policy, to monitor the execution for potentially missed
hidden implicit flows (using the PU strategy [5], see Section 3.1),
and to insert upgrade statements that signal them to the monitor
when counting micro flows. Whenever the monitor terminates the
program because it detects an access to a value $u$ that is marked as
partially leaked, the approach modifies the program by inserting
an upgrade statement at the code location where $u$ is next used;
this upgrade statement in the modified program will then be ex-
cuted whenever $u$ is used again, regardless of whether the same
branch that leads to the insertion of the upgrade statement is taken.
The process continues until it reaches a fixed point, i.e., until the
program has enough upgrade statements for the given tests.

The ability of our analysis to observe hidden implicit flows de-
pends on the completeness of the inferred upgrade statements, since
missing upgrade statements may result in false negatives for hid-
den implicit flows. How often this occurs depends on how well the
analyzed executions cover the branches of the programs. One way
to assess this ability would be to measure tradition branch coverage,
shortly, the percentage of all branches that are covered by the given test
inputs. However, traditional branch coverage is only of limited use
because inserting upgrade statements does not rely on covering all
branches in the code, but only on a subset. Specifically, the ability
to insert upgrade statements depends on the branch coverage for
conditionals that depend on sensitive values. We present a metric
called Sensitive Branch Coverage (SBC) that captures this idea:

$$\text{SBC} = \frac{|\{c \in C \text{ where both true and false branch covered}\}|}{|C|}$$

where $C$ is the set of conditionals that depend on a sensitive
value. For example, consider executing the following program with
x=false:

```javascript
1 var x; var y
2 if (x)
3   y = 5;
```

The set $C$ consists of the conditional at line 2, but since the execution
covers only the false branch, SBC $= 0 \frac{1}{1} = 0$.

3.3 Formalization of Flows and Conditions

We define the syntax and semantics of NanoJS, a simplified core of
JavaScript to illustrate the flow counting performed by our imple-
mentation.

**Notation:** We denote empty sequences by $\varepsilon$. Concatenating two
sequences $r_1$ and $r_2$ is denoted by $r_1 \cdot r_2$. Slightly abusing notation,
we also use the same notation to prepend a single element $a$ to a
sequence $r$ by writing $a \cdot r$. Similarly, we write $a \in r$ to denote that
$a$ occurs in sequence $r$.

**NanoJS syntax:** NanoJS statements:

\[ Stmt := \text{skip} \mid \epsilon \mid c_1 \cdot c_2 \mid \text{sink}(\epsilon) \mid x = e \mid x[y] = e \mid \text{if}(e) \{ c_1 \} \text{else} \{ c_2 \} \mid \text{while} e \text{do} c \]

where $x, y \in \text{Name}$, and $e \in \text{Expr}$
A terminated execution is denoted by \( \varepsilon \). All function calls to sinks with expression \( e \) are modeled by \( \text{sink}(e) \); other function calls are not considered in NanoJS.

**Semantics:** Operationally, the constructs in NanoJS behave as in standard imperative languages. To count micro flows, we associate each primitive value with a tuple \( \kappa : \text{Cnt} \) of flow counts, where \( \text{Cnt} = \mathbb{N}^3 \). A tuple \((e, o, h) \in \text{Cnt}\) denotes \( e \) explicit flows, \( o \) observable flows, and \( h \) hidden flows. We write \( 0 \) for the tuple \((0, 0, 0) \). A value is either a primitive value annotated with a flow count, or an address on the heap. We assume that there is a set \( \text{Base} \) of primitive base types, such as boolean, numbers, and strings. A heap object \( o \in \text{Obj} \) maps a finite set of names to values. We write \( \text{tt} \) for boolean value true and \( \text{ff} \) for boolean value false.

We use flow counts to track how information is propagated by a program, analogous to labels in other information flow monitors. We define a join-semilattice structure for flow counts as follows. In-}

\[
\kappa + \kappa' = \kappa + \Delta(f, e, t) + \kappa'
\]

\[
\kappa' = \kappa + \Delta(f, e, t)
\]

\[
\rho = \rho[x \mapsto u']
\]

\[
(x = e, \rho, h, t, \kappa) \rightarrow (x', \rho', h', t, \kappa')
\]

**E-If**

\[
\kappa = \kappa + \Delta(f, e, t)
\]

\[
\kappa' = \kappa + \Delta(f, e, t)
\]

\[
\rho = \rho[x \mapsto u']
\]

\[
(x = e, \rho, h, t, \kappa) \rightarrow (x', \rho', h', t, \kappa')
\]

**E-Sink**

\[
\kappa = \kappa + \Delta(f, e, t)
\]

\[
\kappa' = \kappa + \Delta(f, e, t)
\]

\[
\rho = \rho[x \mapsto u']
\]

\[
(x = e, \rho, h, t, \kappa) \rightarrow (x', \rho', h', t, \kappa')
\]

**E-Abs**

\[
\kappa = \kappa + \Delta(f, e, t)
\]

\[
\kappa' = \kappa + \Delta(f, e, t)
\]

\[
\rho = \rho[x \mapsto u']
\]

\[
(x = e, \rho, h, t, \kappa) \rightarrow (x', \rho', h', t, \kappa')
\]

**Example**

Finally, Figure 2 gives the rules of small-step operational semantics for NanoJS with flow counting. The way the rules modify the environment and heap is standard. Some standard rules are omitted and provided in the appendix. In addition to the standard execution of a program, the semantics also track flow counts for each value. For example, an assignment statement \( x = e \) propagates the flow counts of the assigned expression \( e \) and additionally increments the explicit flow count if \( e \) has non-zero flows and the observable flow count if the control-flow path is determined by sensitive data. A sink statement \( \text{sink}(e) \) increments global counts representing source-to-sink flows. Since all sink statements model writes to insensitive sinks, any write of an expression with non-zero flow counts will result in incrementing the global counters.

**Security conditions:** We also adapt existing security conditions for tracking only explicit or observable flows to NanoJS [7]. To capture only explicit flows, we use the notion of *explicit secrecy*; intuitively, a run of a program satisfies explicit secrecy if and only if the program obtained by sequentially composing all non-control-flow commands executed during that run does not leak information. For example, the program:

\[
\text{if } (h) \{ l = 1 \} \text{ else } \{ l = 2 \} ; \text{sink}(l)
\]

would produce the extracted programs \( l = 1 ; \text{sink}(l) \) or \( l = 2 ; \text{sink}(l) \) depending on the value of \( h \) in a given run. In both cases, the extracted program contains prohibited information flows, since the source program only leaks information through an implicit flow.

To track only explicit and observable implicit flows, we keep branching constructs in the extracted program, but replace not taken branches by \( \text{skip} \). If the extracted program does not leak sensitive information, then the run satisfies *observable secrecy*. For example, in the program:

\[
\text{if } (h) \{ l = 1 \} \text{ else } \{ \text{skip} \} ; \text{sink}(l)
\]

observable secrecy would extract either \( l = 0 \) ; \( l = 1 \) ; \( \text{skip} \) ; \( \text{sink}(l) \) or \( l = 0 ; \text{skip} \) ; \( \text{sink}(l) \) depending on the value of \( h \) in a given run. In both cases, the extracted program contains prohibited information flows, since the source program only leaks information through an implicit flow; i.e. the extracted program when \( h = \text{tt} \) leaks information, but the extracted program for
\( h = f \) does not. Appendix A gives formal definitions of the two notions.

**Soundness:** To establish soundness of our counting scheme, we show that if all explicit flow counts for all sinks for a given run are 0, then that run satisfies explicit secrecy. Similarly, we show that if all explicit and observable flow counts are 0, the run satisfies observable secrecy. The formal theorem statements and proofs can be found in Appendices B and C.

### 3.4 Implementation

To implement our methodology, we develop a tool for dynamic information flow analysis following Hedin et al. [25, 26]. The implementation builds on Jalangi [43], a dynamic analysis framework for JavaScript that uses source-to-source transformation. Since Jalangi supports ECMAScript 5 only, we down-compile programs written in newer versions of the language with Babel [1]. Building on top of Jalangi allows us to focus on the important parts of the analysis and let the framework handle otherwise challenging aspects of implementing a dynamic information flow analysis, e.g., on the fly instrumentation of code produced by eval, exceptional termination of functions, boxing and unboxing of primitive values [14]. We handle higher-order functions and track dynamic modification of object properties as described by Hedin and Sabelfeld [26]. Our policy language is expressive, allowing the security analyst to mark both functions and arguments of callbacks as sources.

To approximate the effects of native calls, we model them by transferring the labels from all parameters to the return value. Moreover, if one of the parameters is an object, we propagate labels from all its properties to the return value. For a set of frequently used native functions, such as `Array.push`, `Array.forEach`, `Object.defineProperty`, and `Object.defineProperty`, we create richer models that propagate labels more precisely. To increase the confidence in our implementation, we created more than 100 validation tests that assert the correctness of label propagation in typical usage scenarios. When inserting upgrades, the implementation does not modify the actual source code but it stores the source code locations of upgrades, and then performs the upgrades at runtime.

## 4 EMPIRICAL STUDY

This section presents the results of our empirical study that assesses the costs and benefits of tracking different kinds of flows.

The last two columns of Table 1 show the sensitive branch coverage (SBC) and the number of upgrades inserted while executing the benchmarks. Overall, the tests used for the study reach a high SBC, for 54% of the programs even 100%, enabling the analysis to insert upgrade statements. For each of the considered benchmarks, our tool can detect source-to-sink flows. This is hardly surprising, since we already know that the programs contain such flows, but it shows that our tool can handle complex, real-life JavaScript code.

### 4.1 Prevalence of Micro Flows

At first, we address the question of how prevalent explicit, observable implicit, and hidden implicit micro flows are among all operations that induce an information flow. Figure 3 shows the distribution of micro flows for our benchmarks. The majority of benchmarks contain both implicit and explicit micro flows. Benchmark 39 is a special case where reaching the sink is the first operation performed on the untrusted data, and hence the data flows directly from source to sink without producing any micro flow. The explicit flows are by far the most prevalent, appearing in all but one benchmarks. Five benchmarks also contain hidden implicit flows, but we can safely conclude that these cases are rare.

### 4.2 Source-to-sink Flows

We now evaluate source-to-sink flows, which are the ultimate measure of success for an information flow analysis. Source-to-sink flows are what a security analyst ultimately cares about: how does information from a sensitive source reach an insensitive sink? Information flow analysis has no way to show that such a flow is security-relevant, but it is the analyst’s job to further inspect the flows and decide. In this section, however, we have a different goal and setup: we start with a set of known security problems that produce a source-to-sink flow and proceed by showing what type of analysis is needed to detect these problems.

Our tool can enforce different security conditions (cf. Section 3.3). For example, if we are interested only in explicit and observable implicit flows, we can run the tool in observable tracking mode and enforce observable secrecy. Figure 4 presents the number of source-to-sink flows detected by different monitoring strategies. All the integrity vulnerabilities can be detected by taint tracking only, and all the security violations in our data set can be detected through observable tracking. Moreover, all the Node.js vulnerabilities can be detected by the taint tracking only, independently of whether they are confidentiality or integrity vulnerabilities. We argue that this is because our Node.js programs are expected to be trusted. That is, a security issue may arise from a programming error, but not by malicious intention. This assumption does not hold, however, for the fingerprinting and history sniffing benchmarks, where only observable implicit flows contribute to the source-to-sink flows.

A second explanation for why the implicit flows are prevalent in the browser environment is that there are already a set of security mechanisms in the browser that prevent certain type of dangerous behavior. For example when fingerprinting the login state using images, an attacker cannot directly read the bytes of the image due to same origin policy, and hence it relies on measuring its width. We analyzed in detail the additional source-to-sink flows detected by observable tracking for benchmarks 12, 26, 34, 43, and 44, and by PU for benchmark 34. In all these cases the reported flows are false positives, since they do not allow an attacker to exploit the respective vulnerability. In Section 4.4, we discuss in detail why these false positives occur when data is propagated through implicit flows.

Our results indicate that observable tracking is enough to tackle all the real-life security problems we consider and that taint tracking suffices for all the trusted code. We do not claim that there are no real-life security problems beyond observable secrecy, we just do not see any in our data set. Moreover, we believe that when strong controls are in place, attackers will be motivated to use more sophisticated attacks, possibly though the use of hidden implicit flows. However, tracking these flows is expensive as we will see in the remainder of this section.
4.3 Permissiveness

A potential problem for adopting information flow analysis in practice is its limited permissiveness, i.e., the fact that a monitor may terminate the program even though no data flows from a source to a sink. Our metrics allow us to quantify this effect both for the NSU and the PU monitoring strategies. Specifically, we measure how many code locations a user would have to inspect because a monitor terminates the program. The NSU monitor terminates the program when an update of an insensitive variable is performed in a sensitive context. This condition corresponds to observable implicit micro flows and we count the number of code locations where such a flow occurs. The PU monitor terminates the program when an insensitive variable that was updated in a sensitive context is read. This termination condition corresponds to the locations where our tool inserts an upgrade statement. Figure 5 shows the number of code locations affected by the lack of permissiveness for NSU and PU. We exclude benchmarks for which neither of the monitoring strategies raises an alarm. On average, NSU throws 5.46 times more alarms than PU, that is, PU is much more practical than NSU. However, when comparing the PU violations to the number of source-to-sink flows that require PU (Figure 4), we observe that most of the PU alarms do not translate to actual source-to-sink flows and should be considered false positives.

4.4 Label Creep Ratio

As a second metric for the cost of different kinds of flows, we use the Label Creep Ratio (LCR) defined in Section 3.2.1. For each benchmark and monitoring strategy, we measure how the LCR changes during the execution time. Figure 6a shows the ratio for PU monitoring. The metric is not monotonically increasing because the analysis is flow-sensitive, i.e., the security label of a variable may change over time. Nevertheless, the LCR steadily increases for most benchmarks, which confirms the label creep problem. Because our policies are targeted at detecting known security problems in the benchmarks, the maximum LCR reached is relatively low (20%, on average).

A comparison of different monitoring strategies shows that stricter monitoring causes more label creep. On average, observable
The graph shows how label creep increases for observable tracking source-to-sink flow is trivial since the sensitive data is directly code injection, where query is the source and eval is the sink. The source-to-sink flow is trivial since the sensitive data is directly

to track implicit flows in the client-side benchmarks translates to false positives. By revisiting Figure 4, we observe that the implicit flows do not contribute additional source-to-sink violations compared to a taint analysis. Figure 7 shows an excerpt of the source code of the benchmark. The code is vulnerable to code injection, where query is the source and eval is the sink. The source-to-sink flow is trivial since the sensitive data is directly

Table 3: Number of instrumented operations handling sensitive data for different benchmarks and monitors.

<table>
<thead>
<tr>
<th></th>
<th>Explicit Secrecy</th>
<th>Observable Secrecy</th>
<th>Non Interference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>Avg</td>
<td>Max</td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>Avg</td>
<td>Max</td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>Avg</td>
<td>Max</td>
<td></td>
</tr>
<tr>
<td>Command inj.</td>
<td>10 59.339 1,118,862</td>
<td>10 59.383 1,118,910</td>
<td>10 59.540 1,118,941</td>
</tr>
<tr>
<td>Buffer vuln.</td>
<td>3 540 6,152</td>
<td>3 633 7,873</td>
<td></td>
</tr>
<tr>
<td>Client-side</td>
<td>5 98 6,084</td>
<td>98 6,007 24,748</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: LCR over execution time

Figure 7: Implicit flows snippet from benchmark 11.
We borrow their conditions to prove the soundness of our monitor which reduces the number of upgrade statements and increases work has been created during the years to refine Dennings’ and Among the analyses that consider implicit flows, the majority stop

table 4: JavaScript information flow analyses and the flows respectively, and MOD = may modify program behavior.

<table>
<thead>
<tr>
<th>Work</th>
<th>Analysis</th>
<th>Explicit</th>
<th>Obs.</th>
<th>Hidden</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vogt et al. [48]</td>
<td>dynamic</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Jang et al. [27]</td>
<td>hybrid</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Chugh et al. [15]</td>
<td>hybrid</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Tripp et al. [47]</td>
<td>hybrid</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Chaudhry &amp; Naumann [14]</td>
<td>dynamic</td>
<td>✓</td>
<td>✓</td>
<td>NSU</td>
</tr>
<tr>
<td>Hedin et al. [25]</td>
<td>dynamic</td>
<td>✓</td>
<td>✓</td>
<td>NSU</td>
</tr>
<tr>
<td>Bichhawat et al. [11]</td>
<td>dynamic</td>
<td>✓</td>
<td>✓</td>
<td>PU</td>
</tr>
<tr>
<td>Kerschbaumer et al. [31]</td>
<td>dynamic</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Bauer et al. [9]</td>
<td>dynamic</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>De Groot et al. [17]</td>
<td>dynamic</td>
<td>✓</td>
<td>MOD</td>
<td>MOD</td>
</tr>
<tr>
<td>Austin &amp; Flanagan [6]</td>
<td>dynamic</td>
<td>MOD</td>
<td>MOD</td>
<td>MOD</td>
</tr>
</tbody>
</table>

cannot be easily triggered or unfeasible execution paths. Despite these limitations, our study produces interesting insights about the kinds of flows that appear in real-world JavaScript programs and the cost-benefit tradeoff of information flow analysis.

5 RELATED WORK

Denning and Denning pioneered the development and formal description of static information flow analyses [19, 20]. Fenton studies purely dynamic information flow monitors [23]. A huge body of work has been created during the years to refine Dennings’ and Fenton’s ideas and to adapt them to various languages. Table 4 presents some of the more recent tools and shows what kinds of flows they consider. Many analyses consider only explicit flows [42]. Among the analyses that consider implicit flows, the majority stop or modify the program as soon as a hidden flow occurs.

**Information Flow Analysis for JavaScript** Chugh et al. propose a static-dynamic analysis that reports flows from code given to eval() to sensitive locations, such as the location bar of a site [15]. Austin and Flanagan address the problem of hidden implicit flows [4, 5], as discussed in detail in Section 2. Hedin and Sabelfeld propose a dynamic analysis that implements the NSU strategy for a subset of JavaScript [26]. They develop JSFlow, which supports the full JavaScript language, but it requires inserting upgrade statements manually [25]. Birgisson et al. propose to automatically insert upgrade statements [12] by iteratively executing tests under the NSU monitor. Their approach is implemented for a JavaScript-like language, whereas we support the full JavaScript language. Our monitor implements the PU strategy to insert upgrade statements, which reduces the number of upgrade statements and increases permissiveness. Bichhawat et al. propose a variant of PU, where the program is terminated whenever a partially leaked value may flow into the heap [10]. A WebKit-based browser by Kerschbaumer et al. [31] balances performance and permissiveness by probabilistically switching between taint tracking and observable tracking and deploys crowdsourcing techniques to discover information flow violations by Alexa Top 500 pages.

**Other Work on Information Flow Analysis** Balliu et al. study a family of information flow trackers for different kinds of flows and propose security conditions to evaluate their soundness [7]. We borrow their conditions to prove the soundness of our monitor for NanoJS. Bao et al. show that considering implicit flows can cause a significant amount of false positives and propose a criterion to determine a subset of all conditionals to consider [8]. Chandra and Franz propose a VM-based analysis for Java that combines a conservative static analysis with a dynamic analysis to track all three kinds of flows considered in this paper [13]. Dytan is a dynamic information flow analysis for binaries that supports both explicit and observable implicit flows [16]. Myers and Liskov introduce Jif, a language for specifying and statically enforcing security policies for Java programs [36]. A survey by Sabelfeld and Myers provides an overview of further static approaches [39].

**Applications of Information Flow Analysis** Information flow analysis is widely used to discover potential vulnerabilities. All approaches we are aware of consider only a subset of the three kinds of flows. Flux uses taint analysis to find incomplete or missing input validation and generates attacks that try to exploit the potential vulnerabilities [40]. Lekies et al. [33] and Melicher et al. [35] propose a similar approach to detect DOM-based XSS vulnerabilities. Jang et al. analyze various web sites with information flow policies targeted at common privacy leaks and attack vectors, such as cookie stealing and history sniffing [27]. Their analysis considers observable implicit flows but not hidden implicit flows. Sabre analyzes flows inside browser extensions to discover malicious extensions [21]. Their analysis considers only explicit flows.

**Studies of Information Flow** King et al. [32] share our goal of understanding practical trade-offs between explicit and implicit flows. They empirically study implicit flows detected by a static analysis in six Java-based implementations of authentication and cryptographic functions. They report that most of the reported policy violations are false positives, mostly due to conservative handling of exceptions. Our work focuses on dynamic analysis for JavaScript-based implementations, which gives rise to a class of observable secrecy monitors that is not relevant in a static setting. Another empirical study of information flows is by Masri and Podgurski [34]. Their work studies how the length of flows (measured as the length of the static dependence chain), the strength of flows (measured based on entropy and correlations), and different kinds of information flows (explicit and observable implicit) relate to each other. Similar to our methodology, Masri and Podgurski target dynamic analysis. Our work differs by addressing different research questions, a different language, and by considering hidden implicit flows.

6 CONCLUSIONS

This paper presents an empirical study of information flows in real-world programs. Based on novel metrics to capture the prevalence of explicit, observable implicit, and hidden implicit flows, as well as the costs they involve, we study 56 JavaScript programs that suffer from real-world security problems. Our results show that implicit flows are expensive to track in terms of permissiveness, label creep, and runtime overhead. We find taint tracking to be sufficient for most of the studied security problems, while for some privacy scenarios observable tracking is needed. Our work helps security analysts and analysis developers to better understand the cost-benefits tradeoffs of information flow analysis. Furthermore, our findings highlight the need for future research on cost-effective ways to analyze hidden implicit information flows.
ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research and by the Hungarian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConSys and PerfJS projects. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

REFERENCES

A SECURITY DEFINITIONS

The previous section has formally defined the flow counting that is at the heart of our empirical study. We now related the flow counting to three previously described [7] security conditions: Explicit secrecy, which requires the absence of explicit flows, observable secrecy, which requires the absence of both explicit flows and observable hidden flows, and non-interference, which requires the absence of all three kinds of flows, i.e., explicit flows, observable implicit flows, and hidden implicit flows. To describe these security conditions in our formalization, we define an instrumented version of the semantics that, along with counting flows, extracts another of the semantics that, along with counting flows, extracts another.

Intuitively, the extracted program discards all control-flow decisions that influenced the current execution and extracts only the straight-line portion of the current execution. As a result, the extracted program no longer contains any implicit flows. The extraction function for explicit secrecy is defined as follows:

\[ \expL(x = e, \ldots), c_e) = c_e \triangleright S(x = e) \]
\[ \expL(\text{sink}(f), \ldots), c_e) = c_e \triangleright S(\text{sink}(f)) \]
\[ \expL(x, y = e, \ldots), c_e) = c_e \triangleright S(x, y = e) \]
\[ \expL(c_1 ; c_2, \ldots), c_e) = \expL(c_1, \ldots, c_e) \]
\[ \expL(\ldots), c_e) = c_e \]

Based on this extraction function, we can define explicit secrecy:

**Definition 2.** A program \( c \) satisfies explicit secrecy for \( \rho \) and \( h \) iff whenever \((c, \rho, h, \kappa, S), \text{skip} \) \( \xrightarrow{\tau} \expL ((c', \text{enc}', h', \kappa', S'), c'_e) \), then \( c'_e \) is non-interfering for environment \( \rho \) and heap \( h \).

**Observable secrecy:** To define observable secrecy we first define evaluation contexts to keep track of where in a partially extracted program the next statement should be placed. The set \( \text{Ctx} \) of evaluation contexts is defined by the following grammar:

\[ \text{Ctx} := \bullet \mid \text{Stmt} \mid \text{Cxt} \mid \text{Cxt} \mid \text{Cxt} \mid \text{Cxt} \]

Note that with the exception of \( \bullet \), symbolizing a hole in the context, evaluation contexts are a subset of statements. We denote replacing \( \bullet \) by a statement or context \( c \) in a context \( \text{ctx} \) by \( \text{ctx}[c] \). Note that if \( c \in \text{Stmt} \), then \( \text{ctx}[c] \in \text{Stmt} \).

We then define an extraction function \( \text{obs} : \text{Conf} \times \text{Cxt} \rightarrow \text{Cxt} \) to define observable secrecy:

\[ \text{obs}(x = e, \ldots), \text{ctx} = \text{ctx}[x = e ; \bullet] \]
\[ \text{obs}(x, y = e, \ldots), \text{ctx} = \text{ctx}[x, y = e ; \bullet] \]
\[ \text{obs}(\text{sink}(f), \ldots), \text{ctx} = \text{ctx}[\text{sink}(f); \bullet] \]
\[ \text{obs}(c_1 ; c_2, \ldots, \text{ctx}) = \text{obs}(c_1, \ldots, \text{ctx}) \]
\[ \text{obs}(\text{if} (e) \{ c_1 \} \text{ else } \{ \text{skip} \}) | \]
\[ \text{if} (e) \{ \text{skip} \} \text{ else } \{ \bullet \} \text{ Cxt} \]

where \( \text{leaveBranch} \) denotes shifting the hole in the context outside of the branch of the surrounding \( \text{if} \). Note that in programs not initially containing \( \text{pop} \) statements, any \( \text{pop} \) encountered during execution delimits a control-flow construct.

B SOUNDNESS

In this section, we show that if a particular execution results in zero explicit flows, this execution satisfies explicit secrecy. Similarly, if both observable and explicit flow counts are zero, the run satisfies observable secrecy.

**Theorem B.1.** If \((c, \rho, h, t, k), \text{skip} \) \( \xrightarrow{\tau} \expL ((c', \text{enc}', h', \kappa', S'), c'_e) \)
\[ \forall (l, k) \in \tau. \kappa(E) = 0, \text{then } c'_e \text{ satisfies explicit secrecy for } \rho \text{ and } h. \]

Proofs for the two theorems are provided in Appendix C.
We formally define values and objects as follows. The sets are
defined as follows:

\[ E(\text{Addr}(\text{h}, \text{h}')) = E(\text{Base}(\text{Base}(\text{h}))) \]

The helper function \( \text{toVal}(h, v) : \text{Base} \times \text{Name}^* \) is defined as follows:
\[ \text{toVal}(h, v) = \{(h, [v])\} \]

The remaining rules for the operational semantics are the following:

**E-Skip**
\[ \langle \text{skip}, \rho, h, t, \kappa \rangle \rightarrow \langle \varepsilon, \rho, h, t, \kappa \rangle \]

**E-Assign**
\[ \langle x.y \rangle(\rho, h) = \text{uv}_{xy} \quad \kappa_{xy} = \theta(\rho, h) = \text{uv}_{xy} \quad \kappa' = \kappa + \Delta(\kappa_{xy}, \kappa, t) \]
\[ \langle x: g \mapsto \kappa', h' \rangle \rightarrow \langle x, g \mapsto \kappa' \rangle \]

**E-While**
\[ \langle \text{while } e \text{ do } c, \rho, h, t, \kappa \rangle \]
\[ \langle \text{if } e \text{ then } c \text{ else } c \rangle \]
\[ \langle \text{no } c, \rho, h, t, \kappa \rangle \]
\[ \langle \text{upgraded }(x, \rho, h, t, \kappa) \rangle \]
\[ \langle \text{seq } c_1, c_2, \rho, h, t, \kappa \rangle \]
\[ \langle \text{seq } (x : c, \rho, h, t, \kappa) \rangle \]

Two environments and heaps \((\rho_1, h_1)\) and \((\rho_2, h_2)\) are low-equivalent, written \(\rho_1 = \rho_2\) iff

\[ \forall x. (\kappa(x, \rho_1(x)) = \kappa(\rho_2(x)) \land (\kappa(\rho_1(x))) = 0 \Rightarrow \text{toVal}(h, \rho_1(x)) = \text{toVal}(h, \rho_2(x))) \]

The interest of brevity, we elide lemmas about standard properties of the evaluation relation in the following proofs.

**Proof of Theorem B.1.** We define \( \text{dom}(h) = \{0, 0, 0\} \). We define an

safety property \( I_E(h_0, h_0) \subseteq \text{Stmt} \) on extracted programs as follows:

\[ c \in I_E(h_0, h_0) \Rightarrow (\forall p. (\rho, h) = \text{L}(p, h_0) \land (c, \rho, h, h_0, \kappa_0) \rightarrow (c', \rho', h', t', \kappa') \land (c' = \text{sink}(h) \lor c' = \text{sink}(h) \land c_2) \rightarrow (\varepsilon, h) = 0) \]

Moreover, we note that only straight-line programs are extracted

for explicit secrecy and such programs trivially preserve low equivalence

of environments and heaps.

**Additional Definitions.** We define the predicate \( B(p_0, h_0, h_0', h_0'') \subseteq \text{Stmt} \)

where \( \kappa_e \in B \) iff whenever \( (p_0, h_0) \leq (p, h) \), \((p, h_0) = \text{L}(p, h) \), and \((c, \rho, h, [], h_0) \); then \((p', h') \leq (\text{env}(h_0'), h_0') \)

where \((p, h) \leq (p_0, h_0) \) iff the counter of each value in \( p \) and \( h \)

is related by \( \subseteq \) to the corresponding counter in \( p_0 \) and \( h_0 \). Formally,
\[ B(p_1, h_1) \leq (p_2, h_2) \text{ iff } \forall x. v_1 x_1 v_2 x_2. \]

\[ \langle \text{upgraded } (x, \rho, h, t, \kappa) \rangle \text{ dom}(h_1) = \text{dom}(h_2) \text{ and } \forall v_1 x_1 v_2 x_2. h_1(a) = \tau_{x_1} \land h_2(a) = \tau_{x_2} \Rightarrow (h_1 \subseteq k_1) \subseteq (k_1 \subseteq k_2) \text{ and } \forall v_1 x_1 v_2 x_2. (h_1(a) = \tau_{x_1} \land h_2(a) = \tau_{x_2} \Rightarrow (k_1 \subseteq k_2) \text{ and } \forall v_1 x_1 v_2 x_2. h_1(a) = \tau_{x_1} \land h_2(a) = \tau_{x_2} \Rightarrow (h_1 \subseteq k_1) \subseteq (h_2 \subseteq k_2) \text{ and } \forall v_1 x_1 v_2 x_2. (h_1(a) = \tau_{x_1} \land h_2(a) = \tau_{x_2} \Rightarrow (h_2 \subseteq k_2) \}

For the induction to succeed we show the stronger statement

that whenever \((c, p_0, h_0, [], h_0) \), \( \text{skip} \rightarrow^* (c', p_0', h_0', [], h_0') \),

then \( c' \in I_E(p_0, h_0) \) and \( c \in B(p_0, h_0, p_0', h_0') \).

**Proof of Theorem B.1.** We have that \( c' \sigma_0 = c = e \). We have that \( c' \in I_E(p_0, h_0) \) follows from the fact that any sink in \( c' \) is also reachable in \( e \), hence the claim follows from the induction hypothesis.

To show \( c' \in B(p_0, h_0, p_0', h_0') \), we notice that if \( (c', p_0, h_0, [], h_0) \rightarrow^* (e, x, \rho', h', t', \kappa') \) then \( (c', p_0, h_0, [], h_0) \rightarrow^* (x = e, p', h', t', \kappa') \) and \( \rho'' \geq \rho'(x := v_e) \), where \( e \) is the result of the assigned expression with incremented counters. We show that the label of \( x \) is still bound by the corresponding label of \( x \) in \( \rho'' \). From the induction hypothesis we have that \( (p', h') \leq (p_0', h_0') \). If the new label of \( x \) in \( (p_0', h_0') \) is not 0, then the claim follows trivially. If it is low, it follows from the previous fact that \( x \) also receives 0 in \( \rho'', h'' \).

**Case E-Skip.** We have that \( c'' = c \in c = e \). We have that \( c'' \in I_E(p_0, h_0) \) follows easily from the induction hypothesis as the sink statement does not change the environment and heap. \( c'' \in I_E(p_0, h_0) \) follows trivially for sink statements already reachable in \( e \). Since \( c'' \sigma_0 = \text{sink}(e) \), we need to show that this also holds when reaching \( c'' \). Since the explicit flow count is in \( (p_0', h_0') \), we have that this the label of \( e \) in \( (p_0', h_0') \) is 0. Since \( (p', h') \leq (p_0', h_0') \), we have that therefore the label of \( e \) in \( p', h' \) is also 0, as desired.

Clearly whenever \( c \in I_E(p_0, h_0) \), \( c \) satisfies per-run non-interference wrt. \( p_0 \) and \( h_0 \) since low equivalence between memories is preserved and only low expression reach sinks without increasing the counter.

**Proof of Theorem B.2.** For the induction to go through, we show the stronger property that whenever \((c, p, h, [], h_0) \rightarrow^* (c', p', h', t', \kappa), c_e \), \((p, h) = \text{L}(p_2, h_2) \), and \( p_1 \sigma_0 = \emptyset \) then:
we have that $t' \leq t''$; we also have that $\text{length}(t'_2) = \text{length}(t''_2)$ trivially.

In the case where $v_2 \neq \text{tt}$, we have that (if $v \neq \text{skip}$) $\text{pop}(\text{length}(t'), \rho'_2, h'_2, t'_2, k'_2)$ and $\text{pop}(\text{length}(t''), \rho''_2, h''_2, t''_2, k''_2)$ since $v$ is not reached and hence replacing it with $\text{if}(v \neq \text{skip})$ does not affect the execution. Since $t'_2 \leq t''$, we trivially have that then also $t'_2 \leq k'_2, t''$. In both cases, the rest of (3') follows trivially.

Case E-Sink: We have that $c'_2 = c_2[\text{sink}(e); t'] = [v] = [v']$ where $v' = \epsilon_2[\rho'_2, h'_2, τ]$; we also have that $\text{length}(e_2) = \text{length}(t'_2) = \text{length}(t''_2)$ follows from this case in (2') and the fact that assignments do not modify the label stack.

For (2'), note the second alternative of the disjunction of (2') leads to a contradiction, since then $t'' \neq 0$ and this would imply that $\pi_{1,2}(\kappa''_2) \neq 0$, violating the assumption that $\pi_{1,2}(\kappa''_2) = 0$.

We can therefore assume that (if $v \neq \text{skip}$) $\text{pop}(\text{length}(t'), \rho'_2, h'_2, t'_2, k'_2)$ and $\text{pop}(\text{length}(t''), \rho''_2, h''_2, t''_2, k''_2)$ since $v$ is not reached and hence replacing it with $\text{if}(v \neq \text{skip})$ does not affect the execution. Since $t'_2 \leq t''$, we trivially have that then also $t'_2 \leq k'_2, t''$. In both cases, the rest of (3') follows trivially.

Case E-Pop: By this case we have $c'_2 = \text{leaveBranch}(c_2)$ (1') follows easily since (2') we again proceed by case distinction on the disjunction in (2'). In the first case, the conclusion follows easily, since we reach the same state as the execution in $(\rho', h')$. Assume now that (if $v \neq \text{skip}$) $\text{pop}(\text{length}(t'), \rho'_2, h'_2, t'_2, k'_2)$ and $\text{length}(t'_2) = \text{length}(t''_2)$. We proceed by case distinction on this execution reaching the branch surrounding $\bullet$ in $c_2$. We denote this branch by $\text{if}(v \neq \text{skip})$ $\text{else}$ $\{c_2\}$; WLOG assume that $c_2 = \text{skip}$ and $c_1 = c'_2$. Then, we have that $c'_2 = \text{skip}$, $c_1 = c'_2$; $\text{if}(v \neq \text{skip})$ $\text{else}$ $\{c_2\}$; WLOG assume that $c_2 = \text{skip}$ and $c_1 = c'_2$. Then, we have that $c'_2 = \text{skip}$, $c_1 = c'_2$; $\text{if}(v \neq \text{skip})$ $\text{else}$ $\{c_2\}$. We proceed by case distinction on the disjunction in (2'). In the first case, the conclusion follows easily, since we reach the same state as the execution in $(\rho', h')$. Assume now that (if $v \neq \text{skip}$) $\text{pop}(\text{length}(t'), \rho'_2, h'_2, t'_2, k'_2)$ and $\text{length}(t'_2) = \text{length}(t''_2)$, then we have that $c'_2 = \text{pop}(\text{length}(t''), \rho''_2, h''_2, t''_2, k''_2)$.

To show (2') we proceed by case distinction on $v_2 = \text{tt}$. If $v_2 = \text{tt}$, we have that (if $v = \text{skip}$) $\text{else}$ $\{c_2\}$; WLOG assume that $c_2 = \text{skip}$ and $c_1 = c'_2$. Then, we have that $c'_2 = \text{skip}$, $c_1 = c'_2$; $\text{if}(v \neq \text{skip})$ $\text{else}$ $\{c_2\}$; WLOG assume that $c_2 = \text{skip}$ and $c_1 = c'_2$. Then, we have that $c'_2 = \text{skip}$, $c_1 = c'_2$; $\text{if}(v \neq \text{skip})$ $\text{else}$ $\{c_2\}$.