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

ABSTRACT

Information flow analysis prevents secret or untrusted data from flowing into public or trusted sinks. Existing mechanisms cover a wide array of options, ranging from lightweight taint analysis to heavyweight information flow control that also considers implicit flows. Dynamic analysis, which is particularly popular for languages such as JavaScript, faces the question whether to invest in analyzing flows caused by not executing a particular branch, so-called hidden implicit flows. This paper addresses the questions how common different kinds of flows are in real-world programs, how important these flows are to enforce security policies, and how costly it is to consider these flows. We address these questions in an empirical study that analyzes 56 real-world JavaScript programs that suffer from various security problems, such as code injection vulnerabilities, denial of service vulnerabilities, memory leaks, and privacy leaks. The study is based on a state-of-the-art dynamic information flow analysis and a formalization of its core. We find that implicit flows are expensive to track in terms of permissiveness, label creep, and runtime overhead. We find a lightweight taint analysis to be sufficient for most of the studied security problems, while for some privacy-related code, observable tracking is sometimes required. In contrast, we do not find any evidence that tracking hidden implicit flows reveals otherwise missed security problems. Our results help security analysts and analysis designers to understand the cost-benefit tradeoffs of information flow analysis and provide empirical evidence that analyzing information flows in a cost-effective way is a relevant problem.

CCS CONCEPTS

• **Security and privacy → Information flow control**; *Web application security*; • **General and reference** → *Empirical studies.*

KEYWORDS

information flow control, taint tracking, JavaScript, implicit flow

ACM Reference Format:
Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An Empirical Study of Information Flows in Real-World JavaScript. In *14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3338504.3357339

PLAS'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6836-0/19/11…$15.00
https://doi.org/10.1145/3338504.3357339

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, falling 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 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:[1] *Explicit flows* [20] occur whenever sensitive information is passed by an assignment statement or into a sink. *Implicit flows* [20] arise via control-flow structures of programs, e.g., conditionals and loops, when the flow of control depends on a sensitive value. For a dynamic information flow analysis, implicit flows can be further classified into flows that happen because a particular branch is executed, so-called *observable implicit*

[1]There are other kinds of flows, such as timing and cache side-channels, which we ignore here.

```
1  // variable passwd is sensitive
2  var gotIt = false;
3  var paddedPasswd = "xx" + passwd;
4  var knownPasswd = null;
5  if (paddedPasswd === "xxtopSecret") {
6    gotIt = true;
7    knownPasswd = passwd;
8  }
9  // function sink is insensitive
10 sink(gotIt);
```

Figure 1: Program leaking the password to the network.

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 monitoring, 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 can(not) be detected when considering subsets of flows, and what costs considering 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 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 the analysis for 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.

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-crafted input. The single-threaded execution model of

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

Type	ID	Library	Policy	LoC	SBC	Upgs
Injection vulnerabilities	1	fish	module → eval and exec	69	1	0
	2	growl	module → eval and exec	270	1	0
	3	gm	module → eval and exec	1,614	1	0
	4	libnotify	module → eval and exec	54	1	0
	5	mixin-pro	module → eval and exec	168	1	0
	6	modulify	module → eval and exec	2,410	1	0
	7	mol-proto	module → eval and exec	1,696	1	0
	8	mongoosify	module → eval and exec	160	0	1
	9	m-log	module → eval and exec	243	1	0
	10	mobile-icon-resizer	file system API → eval and exec	410	1	0
	11	mongo-parse	module → eval and exec	506	1	0
	12	mongoosemask	module → eval and exec	12,750	0.78	28
	13	mongui	HTTP API → eval and exec	1,539	0.44	0
	14	mongo-edit	HTTP API → eval and exec	577	0	0
	15	mock2easy	HTTP API → eval and exec	1,217	0.07	3
	16	chook-growl-reporter	module → eval and exec	243	1	0
	17	git2json	module → eval and exec	434	1	0
	18	kerb_request	module → eval and exec	67	1	0
	19	printer	module → eval and exec	139	1	0
ReDoS vulnerabilities	20	debug	module → regex matching	360	1	0
	21	mime	module → regex matching	108	1	0
	22	tough-cookie	module → regex matching	1,145	1	0
	23	fresh	module → regex matching	59	0.5	0
	24	forwarded	module → regex matching	30	0	0
	25	underscore.string	module → regex matching	1,779	1	0
	26	ua-parser-js	module → regex matching	584	0.50	6
	27	parsejson	module → regex matching	46	1	0
	28	useragent	module → regex matching	6,827	1	0
	29	no-case	module → regex matching	33	1	0
	30	content-type-parser	module → regex matching	221	1	0
	31	timespan	module → regex matching	577	0.20	4
	32	string	module → regex matching	2,001	1	0
	33	content	module → regex matching	125	0.42	0
	34	slug	module → regex matching	375	0.5	2
	35	htmlparser	module → regex matching	2,155	0.65	5
	36	charset	module → regex matching	49	0.5	0
	37	mobile-detect	module → regex matching	612	1	0
	38	ismobilejs	module → regex matching	935	0.33	1
	39	dns-sync	module → regex matching	76	1	0
Buffer vulns.	40	ip	buffer reading → module	325	0.76	0
	41	concat-stream	buffer reading → module	132	1	0
	42	bl	buffer reading → module	206	0.72	4
	43	request	buffer reading → HTTP	2,217	0.52	0
	44	ws	buffer reading → HTTP API	2,449	0.07	1
	45	floody	buffer reading → HTTP API	94	0.8	0
	46	tunnel-agent	buffer reading → HTTP API	225	1	0
Client-side issues	47	History sniffing [27]	HTMLElement.color → img.src	42	0	3
	48	Font fingerpr. [3]	HTMLElement.offsetWidth → img.src	145	0.5	1
	49	Font fingerpr.[3]	HTMLElement.offsetWidth → img.src	44	0.02	3
	50	Font fingerpr.[4]	HTMLElement.offsetWidth → img.src	134	1	0
	51	Browser ext. fingerpr. [44]	HTMLElement.offsetWidth → request.open	1,451	1	1
	52	DoNotTrack leakage[5]	navigator_doNotTrack → HTMLElement.html	20	0	1
	53	Login state leakage[6]	onload event → document.innerHTML	191	1	0
	54	Engine fingerpr.[7]	HTMLElement.type → console.log	129	0	1
	55	Browser ext. fingerpr.[8]	onload event → HTMLElement.innerHTML	37	0	0
	56	Resource fingerpr.[9]	onload event → console.log	43	0	0

[2] https://new-iflow.herokuapp.com/download-iflow.html
[3] https://www.privacytool.org/AnonymityChecker/
[4] http://www.lalit.org/lab/javascript-css-font-detect/
[5] https://browserleaks.com/js/donottrack.js
[6] https://robinlinus.github.io/socialmedia-leak/
[7] https://www.privacytool.org/AnonymityChecker/
[8] https://popmyads.com/
[9] https://browserleaks.com/firefox#more

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.

3 METHODOLOGY

To address the research questions from Section 1, we present a methodology that combines a set of novel metrics with a dynamic information flow analysis [25, 26], a monitoring strategy [5], and an automated mechanism to insert upgrade statements [12]. The metrics summarize the flows observed during the program execution. This section provides the necessary background on information flow analysis, an informal description of our methodology, and definitions of the metrics. It also presents a formalization of the core of our methodology.

3.1 Setting: Information Flow Analysis

Security labels An information flow analysis associates each value with a *security label* that indicates how sensitive the value is. Labels are typically arranged in a lattice [19]. To ease the presentation, we focus on two labels: H (for high or sensitive) and L (for low or insensitive), where H is more sensitive than L. Given a label $\ell \in \{H, L\}$, we write v^{ℓ} to denote that a value v has security label ℓ. If a value v does not have a label, we assume it is implicitly labeled as L.

Table 2: Monitoring strategies ("Expl." = explicit, "Obs." = observable implicit, "Hid." = hidden implicit).

Strategy	Sec. condition	Tracked flows			Permissiveness
		Expl.	Obs.	Hid.	
Taint tracking	Explicit secrecy	✓			Stop when H-labeled value reaches sink.
Observable tracking	Observable secrecy	✓	✓		Stop when H-labeled value reaches sink.
No Sensitive Upgrade	Non-interference	✓	✓	✓	Stop when L-labeled variable is written in sensitive context.
Permissive Upgrade	Non-interference	✓	✓	✓	Stop when partially leaked value is used.

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.

Monitoring strategies 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()`:

```
var location; var y; var z;
if (10 < location < 20) {
  y = "Home"; }
//upgrade(y);
z = "You are at " + y;
```

Consider now an execution where the location is 15^H. 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 updating 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 `location=15`^{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:

```
var x = 3^H; var y = 5^H; var z;
x = y; // no explicit flow
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:

```
var x = true^H; var y = 3; var z;
if (x)
  y = 5; // observable implicit flow
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:

```
var x = false^H; var y; var z;
if (x)
  y = 5;          // not executed, no flow
upgrade(y);       // hidden implicit flow
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 are labeled as sensitive.

$$LCR = \frac{\text{\# sensitive variables/fields ever assigned}}{\text{\# variables/fields ever assigned}}$$

For a given monitoring strategy, the Label Creep Ratio is the ratio 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:

```
var x = false^H; var y; var z;
if (x)
  y = 5;
upgrade(y); // hidden micro flow
z = x;      // explicit micro flow
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:

```
var x = true^H; var y; var z;
if (x)
  y = 5;   // observable flow
z = x;     // explicit flow
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 tracking 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 statements [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 executed 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 depends on the completeness of the inferred upgrade statements, since missing upgrade statements may result in false negatives for hidden 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, i.e., 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:

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

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

```
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, $SBC = \frac{0}{1} = 0$.

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 implementation.

Notation: We denote empty sequences by ε. Concatenating two sequences τ_1 and τ_2 is denoted by $\tau_1.\tau_2$. Slightly abusing notation, we also use the same notation to prepend a single element α to a sequence τ by writing $\alpha.\tau$. Similarly, we write $\alpha \in \tau$ to denote that α occurs in sequence τ.

NanoJS syntax: NanoJS statements:

$$\begin{aligned} Stmt ::= \; & \textbf{skip} \mid \varepsilon \mid c_1; c_2 \mid \textbf{sink}(e) \mid x = e \mid x[y] = e \mid \\ & \textbf{if}\ (e)\ \{\ c_1\ \}\ \textbf{else}\ \{\ c_2\ \} \mid \textbf{while}\ e\ \textbf{do}\ c \end{aligned}$$

$$\text{where } x, y \in Name, \text{ and } e \in Expr$$

A terminated execution is denoted by ε. All function calls to sinks with expression e are modeled by **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 $\kappa : Cnt$ of flow counts, where $Cnt = \mathbb{N}^3$. A tuple $(e, o, h) \in Cnt$ denotes e explicit flows, o observable flows, and h hidden flows. We write $\mathbf{0}$ 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 *Base* of primitive base types, such as boolean, numbers, and strings. A heap object $o \in Obj$ maps a finite set of names to values. We write **tt** for boolean value *true* and **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. Intuitively, a non-zero flow count indicates a sensitive value, whereas if all flow counts are zero, the value is insensitive: The join of two flow counts is defined as $\kappa_1 \sqcup \kappa_2 = \kappa_1 + \kappa_2$, where $\kappa_1 + \kappa_2$ denotes the pointwise addition of the two flow counts. We write $\bigsqcup t$ to denote the join operator over a list of flow counts t.

Two flow counts satisfy $\kappa_1 \sqsubseteq \kappa_2$ if whenever $\kappa_2 = (0, 0, 0)$ then $\kappa_1 = (0, 0, 0)$.

A configuration $\langle c, \rho, h, t, \kappa\rangle$ consists of a statement $c \in Stmt$, an environment $\rho : Name \rightarrow Value$ mapping variable names to values, a heap $h : Addr \rightarrow Obj$, a stack of security levels $t \in \mathcal{L}^\star$, and a sink counter $\kappa : Cnt$ counting flows reaching sink statements; we denote the set of configurations by $Conf$. An execution of a NanoJS program yields a trace $Tr = Value^\star$ indicating outputs produced by the execution.

We now define the small-step semantics of NanoJS. A step $\langle c, \rho, h, t, \kappa\rangle \xrightarrow{\tau} \langle c', \rho', h', t', \kappa'\rangle$ denotes a single evaluation step producing trace τ. We write ε for a terminated execution. Slightly abusing notation, we define $\bigsqcup(h, v)$ as the join of all labels occurring in value v with heap h. For simplicity, we assume that there are no cyclical references on the heap.

The function $upgrade(x, \rho, h)$ denotes the pair (ρ', h'), where the hidden flow count of all components of the value of a variable $x \in Name$ is incremented by 1. To update flow counts, we use an auxiliary function $\Delta : Cnt \times Cnt \times Cnt^\star \rightarrow (Cnt)$. Intuitively, $\Delta(\kappa_{old}, \kappa_{new}, t)$ increments the explicit and observable flow counters for assigning a value with flow count κ_{new} to a location with label κ_{old} while the security stack is t. We define $\Delta(\kappa_{old}, \kappa_{new}, t) = (\Delta_e, \Delta_o, 0)$ where $\Delta_e = \begin{cases} 1 & \kappa_{new} \not\sqsubseteq \kappa_{old} \\ 0 & \text{otherwise} \end{cases}$ and $\Delta_o = \begin{cases} 1 & \kappa_{old} = \mathbf{0} \wedge \bigsqcup t \neq \mathbf{0} \\ 0 & \text{otherwise} \end{cases}$

To define observations based on references passed to sinks, we use a helper function $toVal(h, v) : \{Base \times Name^\star\}$ that, given a value, returns all references to heap objects reachable from the value. We denote evaluating an expression e in environment ρ and heap h by $[\![e]\!](\rho, h)$. The rules propagate flow counts into the result values; for example, adding two values with one explicit flow each will result in two explicit flows in the result. We assume, contrary to real-world JavaScript, that expressions do not have side effects.

E-Assign
$$\frac{[\![x]\!](\rho, h) = v_x \qquad \kappa_x = \bigsqcup(h, v_x) \qquad [\![e]\!](\rho, h) = v^{\kappa_e} \qquad \kappa' = \kappa_e + \Delta(\ell_x, \ell_e, t) \qquad v' = v^{\kappa'} \qquad \rho' = \rho[x \mapsto v']}{\langle x = e, \rho, h, t, \kappa\rangle \rightarrow \langle \varepsilon, \rho', h, t, \kappa\rangle}$$

E-If
$$\frac{[\![e]\!](\rho, h) = v^{\kappa'} \qquad i = \begin{cases} 1 & v = \mathbf{tt} \\ 2 & \text{otherwise} \end{cases}}{\langle \mathbf{if}\ (e)\ \{\ c_1\ \}\ \mathbf{else}\ \{\ c_2\ \}, \rho, t, \kappa\rangle \rightarrow \langle c_i\ ;\ \mathbf{pop}, \rho, \kappa'.t, \kappa\rangle}$$

E-Sink
$$\frac{[\![e]\!](\rho, h) = v^{\kappa''} \qquad \kappa_a = \bigsqcup(v, h) \qquad \kappa' = \kappa + \kappa_a + \Delta(\mathbf{0}, \kappa_a, t)}{\langle \mathbf{sink}(e), \rho, h, t, \kappa\rangle \xrightarrow{toVal(h, v)} \langle \varepsilon, \rho, t, \kappa'\rangle}$$

E-UpgradeL
$$\frac{[\![x]\!](\rho, h) = v^{\mathbf{0}} \qquad v' = v^{(0,0,1)} \qquad (\rho', h') = upgrade(x, \rho[x \mapsto v'], h)}{\langle \mathbf{upgrade}(x), \rho, h, t, \kappa\rangle \rightarrow \langle \varepsilon, \rho', h', t, \kappa\rangle}$$

Figure 2: Rules for NanoJS with flow counting.

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 **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 **if** (h) { $l = 1$ } **else** { $l = 2$ } ; **sink**(l) would produce the extracted programs $l = 1$; **sink**(l) or $l = 2$; **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 **skip**. If the extracted program does not leak sensitive information, then the run satisfies *observable secrecy*. For example, in the program $l = 0$; **if** (h) { $l = 1$ } **else** { **skip** } ; **sink**(l), observable secrecy would extract either $l = 0$; **if** (h) { $l = 1$ } **else** { **skip** } ; **sink**(l) or $l = 0$; **if** (h) { **skip** } **else** { **skip** } ; **sink**(l). This matches the intuition that an observable flow only occurs in the run where h is **tt**, where the assignment $l = 1$ is executed, but not in a run where h is **ff**, since this run only leaks information through a hidden implicit flow; i.e. the extracted program when $h = \mathbf{tt}$ leaks information, but the extracted program for

$h = \mathbf{ff}$ 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.call`, 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 analysts ultimately cares about: how does information from a sensitive source reaches 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.

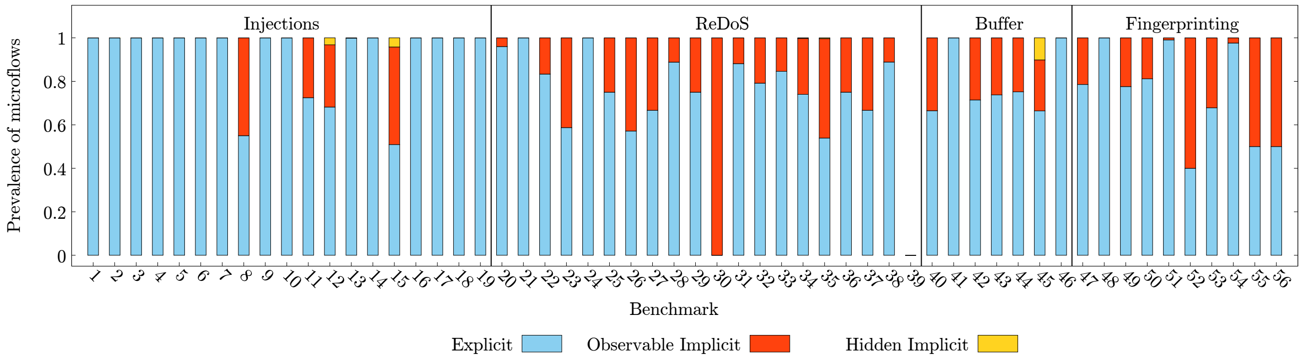


Figure 3: Prevalence of micro flows.

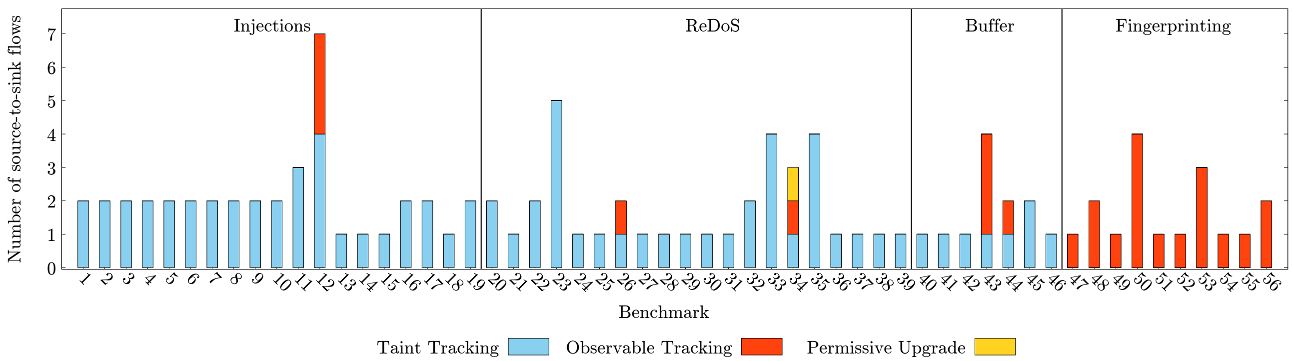


Figure 4: Number of source-to-sink flows detected at different security modes.

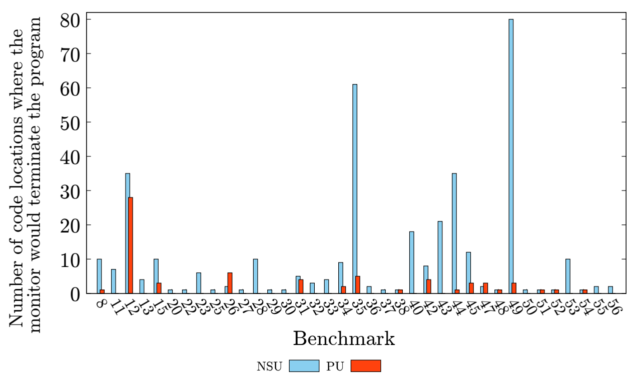


Figure 5: Number of violations, summarized by code locations, raised by NSU and PU monitors.

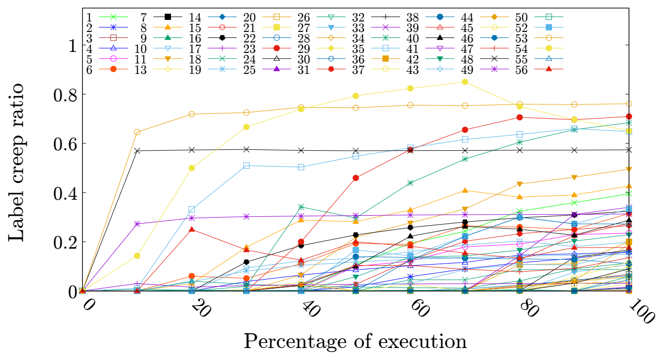
4.3 Permissiveness

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.

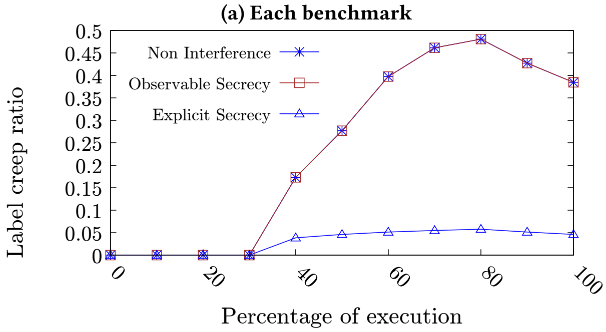
4.4 Label Creep Ratio

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



(a) Each benchmark



(b) Benchmark 11 at various security modes

Figure 6: LCR over execution time

```
1  // query marked 'sensitive':
2  function parseQuery(query) {
3    // query pushed on the stack:
4    if(query instanceof Function) {
5      var nF = eval(query); // sink call
6      return [new Part(null, '$', nF)];
7    }
8  }
9  function Part(f, operator, operand, p){
10   if(p === undefined)
11     p = [];                         // implicit
12   this.field = f;                   // implicit
13   this.operator = operator;         // implicit
14 }
```

Figure 7: Implicit flows snippet from benchmark 11.

tracking has a 0.3% smaller LCR than PU; a taint tracking analysis has a 45.4% smaller LCR than observable tracking. Figure 6b illustrates this effect with a representative benchmark (number 11). The graph shows how label creep increases for observable tracking compared to taint tracking.

We illustrate with the same benchmark 11 how label creep may translate 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.

	Explicit Secrecy			Observable Secrecy			Non Interference		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Command injection	10	59,339	1,118,862	10	59,383	1,118,910	10	59,540	1,118,941
ReDoS vuln.	3	210	2,064	3	540	6,152	3	633	7,073
Buffer vuln.	98	5,740	24,690	98	6,007	24,748	98	6,084	24,843
Client-side progr.	4	5,919	40,364	14	19,555	134,765	16	20,890	136,502

passed to the sink at line 3, which a taint tracker easily detects. In addition, observable tracking pushes the query on the security stack at line 2, which causes implicit flows at lines 10 and 11 where two constants are written to memory. For detecting code injections, these flows are irrelevant. For example, suppose we have a statement `eval(this.operator)` at line 12, for which observable tracking would report a source-to-sink flow. This source-to-sink flow would be a false positive because the attacker can only control whether the call to `eval` happens, not what value flows into it.

4.5 Runtime Overhead

The last cost metric we use is a proxy measure for the runtime overhead imposed by different monitors. For each benchmark we count the number of operations that propagate a label or that modify the security stack. Table 3 shows how the number of events depends on the kind of monitor, aggregated by the different types of vulnerabilities we consider. As expected, raising the security bar translates into larger runtime overhead. Interestingly, this increase is not uniform across the different types of benchmarks. For injections, the cost increase is relatively small, while for ReDoS and client-side programs the increase between explicit and observable secrecy is more than 2.5-fold. We hypothesize that this is due to the structure of the programs: when comparing these findings with the micro flows in Figure 3, we see that implicit flows are more common in ReDoS and client-side programs than in injections. The price paid to track implicit flows in the client-side benchmarks translates to detected source-to-sink flows, as we have seen in Section 4.2, while this is not the case for ReDoS vulnerabilities.

4.6 Threats to Validity

The validity of the conclusions drawn from our study is subject to several threats. First, our dynamic information flow analysis uses a simple model for native functions (Section 3.4), which may not accurately capture all effects of these functions. To minimize the influence of this limitation, we focus the study on subject programs that have relatively few native calls. We also wrote a set of precise models for some of the array and string native functions. Second, our results are limited to the programs we consider and may not generalize to other programs or classes of programs. In particular, we mostly consider non-malicious programs, where difficult-to-analyze flows may be less prevalent than in malicious code. Our methodology is generic enough to be easily applied to other programs. Finally, the hidden implicit flows that our methodology can observe are bounded by the upgrade statements inserted into the programs, which in turn depend on the tests we use to exercise the programs. To mitigate this threat we constructed tests in a way that increases the sensitive branch coverage. However, multiple paths cannot be covered due to a variety of reasons, e.g., error cases that

Table 4: JavaScript information flow analyses and the flows they support: ✓= considers this flow, - = does not consider this flow, NSU and PU = may abort due to NSU or PU checks, respectively, and MOD = may modify program behavior.

Work	Analysis	Explicit	Obs.	Hidden
Vogt et al. [48]	dynamic	✓	✓	-
Jang et al. [27]	hybrid	✓	✓	-
Chugh et al. [15]	hybrid	✓	✓	✓
Tripp et al. [47]	hybrid	✓	-	-
Chudnov & Naumann [14]	dynamic	✓	✓	NSU
Hedin et al. [25]	dynamic	✓	✓	NSU
Bichhawat et al. [11]	dynamic	✓	✓	PU
Kerschbaumer et al. [31]	dynamic	✓	✓	-
Bauer et al. [9]	dynamic	✓	-	-
De Groef et al. [17]	dynamic	MOD	MOD	MOD
Austin & Flanagan [6]	dynamic	MOD	MOD	MOD

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. Flax 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 Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

REFERENCES

[1] Accessed: 2018-02-08. Babel JavaScript compiler. https://babeljs.io.
[2] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. 674–689.
[3] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1129–1140.
[4] Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security (PLAS 2009)*. ACM, 113–124.
[5] Thomas H. Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security (PLAS 2010)*. ACM, 3.
[6] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. 165–178.
[7] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. 2017. We Are Family: Relating Information-Flow Trackers. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*. 124–145.
[8] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Strict control dependence and its effect on dynamic information flow analyses. In *International Symposium on Software Testing and Analysis (ISSTA 2010)*. ACM, 13–24.
[9] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Network and Distributed System Security Symposium (NDSS 2015)*. The Internet Society.
[10] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information Flow Control in WebKit's JavaScript Bytecode. In *Principles of Security and Trust (POST 2014) (LNCS)*, Vol. 8414. Springer, 159–178.
[11] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. 2017. WebPol: Fine-Grained Information Flow Policies for Web Browsers. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*. 242–259.
[12] Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. 2012. Boosting the permissiveness of dynamic information-flow tracking by testing. In *European Symposium on Research in Computer Security (ESORICS 2012)*. LNCS, Vol. 7459. Springer, 55–72.
[13] Deepak Chandra and Michael Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 463–475.
[14] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *ACM Conference on Computer and Communications Security (CCS 2015)*. ACM, 629–643.
[15] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for JavaScript. In *Programming Language Design and Implementation (PLDI 2009)*. ACM, 50–62.
[16] James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis (ISSTA 2007)*. ACM, 196–206.
[17] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. 748–759.
[18] Dorothy Denning. 1982. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[19] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243.
[20] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (1977), 504–513.
[21] Mohan Dhawan and Vinod Ganapathy. 2009. Analyzing information flow in JavaScript-based browser extensions. In *Annual Computer Security Applications Conference (ACSAC 2009)*. IEEE, 382–391.
[22] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *Internet Measurement Conference (IMC)*. 475–488.
[23] J. S. Fenton. 1974. Memoryless Subsystems. *Comput. J.* 17, 2 (1974), 143–147.
[24] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models.. In *S&P*.
[25] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing (SAC 2014)*. ACM, 1663–1671.
[26] Daniel Hedin and Andrei Sabelfeld. 2012. Information-Flow Security for a Core of JavaScript. In *Computer Security Foundations Symposium (CSF 2012)*. IEEE, 3–18.
[27] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, 270–283.
[28] D. Jang, R. Jhala, S. Lerner, and H. Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *CCS*.
[29] Martin Johns. 2008. On JavaScript Malware and related threats. *Journal in Computer Virology* 4, 3 (2008), 161–178.
[30] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*.
[31] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. CrowdFlow: Efficient Information Flow Security. In *ISC (Lecture Notes in Computer Science)*, Vol. 7807. Springer, 321–337.
[32] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. 2008. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *International Conference on Information Systems Security (ICISS 2008) (LNCS)*, Vol. 5352. Springer, 56–70.
[33] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *ACM Conference on Computer and Communications Security (CCS 2013)*. ACM, 1193–1204.
[34] Wes Masri and Andy Podgurski. 2009. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 19, 2 (2009).
[35] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In *Proceedings of the 25th Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2018.23309 To appear.
[36] Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 410–442.
[37] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 541–555.
[38] A. Russo, A. Sabelfeld, and K. Li. 2009. Implicit flows in malicious and nonmalicious code. *Marktoberdorf Summer School (IOS Press)* (2009).
[39] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.
[40] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Network and Distributed System Security Symposium (NDSS 2010)*. The Internet Society.
[41] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. 2016. Explicit Secrecy: A Policy for Taint Tracking. In *EuroS&P*.
[42] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE S&P*.
[43] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, 488–498.
[44] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. 2017. Discovering Browser Extensions via Web Accessible Resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*. 329–336.
[45] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium*. 361–376.
[46] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. 2018. Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*.
[47] Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid Security Analysis of Web JavaScript Code via Dynamic Partial Evaluation. In *ISSTA*.

[48] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007.*

[49] Zachary Weinberg, Eric Y Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. 2011. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Security and Privacy (SP), 2011 IEEE Symposium on.* IEEE, 147–161.

[50] Stephan Zdancewic. 2002. *Programming Languages for Information Security.* Ph.D. Dissertation. Cornell University, Ithaca, NY, USA.

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 program. Intuitively, the extracted program preserves the semantics of the original program execution but exposes only a subset of all flows.

To formalize non-interference, we first refer to low-equivalence on environments and heaps. Two environments and heaps are low-equivalent if they are equal on all insensitive values. For example, when considering integrity, the two states are equal on all non-attacker-controlled variables. Dually, for confidentiality, this means that the attacker cannot observe any difference between the two states. Non-interference is defined in terms of low-equivalence of initial environments and heaps. Intuitively, an execution satisfies non-interference iff the same trace can be produced for any indistinguishable starting environment and heap.

Definition 1 (Non-interference). *A program c satisfies non-interference for environment ρ_1 and heap h_1, iff whenever $\langle c, \rho_1, h_1, [], \mathbf{0}\rangle \xrightarrow{\tau}{}^{\star} \langle c', \rho_1', h_1', t_1', \kappa_1'\rangle$, then for all ρ_2, h_2, where $(\rho_1, h_1) =_L (\rho_2, h_2)$, it holds that $\langle c, \rho_2, h_2, [], \mathbf{0}\rangle \xrightarrow{\tau}{}^{\star} cnf'$ for some cnf'.*

Explicit secrecy and observable secrecy are both defined by extracting a simpler program during the execution of a program which eliminates information flows not considered by the security condition: Programs extracted by explicit secrecy contain no control-flow information, whereas programs extracted by observable secrecy discard statements in untaken branches, thus removing leaks through not executed statements in other branches in the program.

The following describes the program extraction formally. We extend each configuration cnf with an extracted statement c_e, written (cnf, c_e), where $cnf \in Conf$ and $c_e \in A$ with A referring to the set of extracted statements for a given security condition. For each security condition, we below define an extraction extraction function $E : Conf \times A \to A$ and then use E in the execution steps of the instrumented semantics: $(cnf, c_e) \xrightarrow{\tau}_E (cnf, E(cnf, c_e))$.

Explicit secrecy: For explicit secrecy, we disregard control-flow-related statements by defining an extraction function *expl*. 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:

$$\begin{aligned}
&expl(\langle x = e, \ldots\rangle, c_e) = c_e \,;\, x = e\\
&expl(\langle \mathbf{sink}(f), \ldots\rangle, c_e) = c_e \,;\, \mathbf{sink}(f)\\
&expl(\langle x.y = e, \ldots\rangle, c_e) = c_e \,;\, x.y = e\\
&expl(\langle c_1 \,;\, c_2, \ldots\rangle, c_e) = expl(\langle c_1, \ldots\rangle, c_e)\\
&expl(\langle \ldots\rangle, c_e) = c_e
\end{aligned}$$

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

Definition 2. *A program c satisfies explicit secrecy for ρ and h iff whenever $(\langle c, \rho, h, \kappa, S\rangle, \boldsymbol{skip}) \xrightarrow{\tau}{}^{\star}_{expl} (\langle c', env', h', \kappa', S'\rangle, c_e')$, then c_e' is non-interfering for environment ρ 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 *Cxt* of evaluation contexts is defined by the following grammar:

$$\begin{aligned}
Cxt ::= \;& \bullet \mid Stmt \,;\, Cxt \mid \mathbf{if}\,(e)\;\{\; Cxt \;\}\;\mathbf{else}\;\{\;\mathbf{skip}\;\} \mid\\
& \mathbf{if}\,(e)\;\{\;\mathbf{skip}\;\}\;\mathbf{else}\;\{\; Cxt \;\}
\end{aligned}$$

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 cxt by $cxt[c]$. Note that if $c \in Stmt$, then $cxt[c] \in Stmt$.

We then define an extraction function $obs : Conf \times Cxt \to Cxt$ to define observable secrecy:

$$\begin{aligned}
&obs(\langle x = e, \ldots\rangle, cxt) = cxt[x = e \,;\, \bullet]\\
&obs(\langle x.y = e, \ldots\rangle, cxt) = cxt[x.y = e \,;\, \bullet]\\
&obs(\langle \mathbf{sink}(f), \ldots\rangle, cxt) = cxt[\mathbf{sink}(f) \,;\, \bullet]\\
&obs(\langle c_1 \,;\, c_2, \ldots\rangle, cxt) = obs(\langle c_1, \ldots\rangle, cxt)\\
&obs(\langle \mathbf{if}\,(e)\;\{\; c_1 \;\}\;\mathbf{else}\;\{\; c_2 \;\}, \rho, h, \ldots\rangle, cxt) =\\
&\qquad\begin{cases} cxt[\mathbf{if}\,(e)\;\{\;\bullet\;\}\;\mathbf{else}\;\{\;\mathbf{skip}\;\}] & [\![e]\!](\rho, h) = \mathbf{tt}\\ cxt[\mathbf{if}\,(e)\;\{\;\mathbf{skip}\;\}\;\mathbf{else}\;\{\;\bullet\;\}] & \text{otherwise}\end{cases}\\
&obs(\langle \mathbf{pop}, \ldots\rangle, cxt) = leaveBranch(cxt)\\
&obs(\langle \ldots\rangle, cxt) = cxt
\end{aligned}$$

where *leaveBranch* denotes shifting the hole in the context outside of the branch of the surrounding **if**. Note that in programs not initially containing **pop** statements, any **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 $(\langle c, \rho, h, t, \kappa\rangle, \boldsymbol{skip}) \xrightarrow{\tau}{}^{\star}_{expl} (\langle \varepsilon, \rho', h', t', \kappa'\rangle, c_e)$ and $\forall (l, \kappa) \in \tau.\ \kappa(\boldsymbol{E}) = 0$, then c_e satisfies explicit secrecy for ρ and h.*

Proofs for the two theorems are provided in Appendix C.

Theorem B.2. *If $(\langle c, \rho, h, t_0, \kappa\rangle, \bullet) \xrightarrow{\tau}{}^{\star} (\langle \varepsilon, \rho', h', t', \kappa'\rangle, c_e)$ and $\forall (l, \kappa) \in \tau.\ \kappa(\boldsymbol{E}) = \kappa(\boldsymbol{O}) = 0$, then $c_e[\boldsymbol{skip}]$ satisfies observable secrecy for ρ and h.*

We omit a similar soundness statement for non-interference, as the monitor follows the same approach as a traditional permissive-upgrade-based information flow monitor, under the additional assumption that all required upgrade statements were inserted during the testing phase.

C PROOFS AND ADDITIONAL DEFINITIONS

We formally define values and objects as follows. The sets are defined by (mutual-)inductively:

$$\frac{n \in \mathit{Base} \qquad \kappa \in \mathit{Cnt}}{n^{\kappa} \in \mathit{Value}} \qquad \frac{a \in \mathit{Addr} \qquad \kappa \in \mathit{Cnt}}{a^{\kappa} \in \mathit{Value}}$$

$$\frac{V \subset_{\text{fin}} \mathit{Name} \qquad f \in (V \to \mathit{Value})}{f \in \mathit{Obj}}$$

The formal definition of joining the labels of values is given by $\bigsqcup(h, n^{(\ell,\kappa)}) = \ell$ if $n \in \mathit{Base}$ and $\bigsqcup(h, a) = \bigsqcup_{x \in \mathrm{dom}(h(a))} (\bigsqcup(h, h(x)))$ if $a \in \mathit{Addr}$. The helper function $\mathit{toVal}(h, v) : \{\mathit{Base} \times \mathit{Name}^{\star}\}$ is defined as follows: $\mathit{toVal}(h, b) = \{(b, [])\}$ if $b \in \mathit{Base}$, and $\mathit{toVal}(h, r) = \{(b', p.[x]) | (b', p) \in \mathit{toVal}(h, h(x)), x \in \mathrm{dom}(h)\}$

The remaining rules for the operational semantics are the following:

E-Skip
$$\frac{}{\langle \mathbf{skip}, \rho, h, t, \kappa\rangle \to \langle \varepsilon, \rho, h, t, \kappa\rangle}$$

E-AssignField
$$\frac{\begin{array}{ccc} [\![x]\!](\rho, h) = a_x & a_x \in \mathit{Addr} \quad o = h(a_x) & y \in \mathrm{dom}(o) \\ [\![x.y]\!](\rho, h) = v_{xy} & \kappa_{xy} = \bigsqcup(h, v_{xy}) & [\![e]\!](\rho, h) = v^{\kappa} \\ \kappa' = \kappa + \Delta(\kappa_{xy}, \kappa, t) & o' = o[y \mapsto v^{\kappa'}] & h' = h[a_x \mapsto o'] \end{array}}{\langle x.y = e, \rho, h, t, \kappa\rangle \to \langle \varepsilon, \rho, h', t, \kappa\rangle}$$

E-While
$$\frac{}{\begin{array}{c}\langle \mathbf{while}\ e\ \mathbf{do}\ c, \rho, h, t, \kappa\rangle \to \\ \langle \mathbf{if}\ (e)\ \{\ c\ ;\ \mathbf{while}\ e\ \mathbf{do}\ c\ \}\ \mathbf{else}\ \{\ \varepsilon\ \}, \rho, h, t, \kappa\rangle\end{array}}$$

E-UpgradeH
$$\frac{[\![x]\!](\rho, h) = v^{\kappa} \qquad \kappa \neq \mathbf{0}}{\langle \mathbf{upgrade}(x), \rho, h, t, \kappa\rangle \to \langle \varepsilon, \rho, h, t, \kappa\rangle}$$

E-Seq
$$\frac{\langle c_1, \rho, h, t, \kappa\rangle \xrightarrow{\tau} \langle c_1', \rho', h', t', \kappa'\rangle}{\langle c_1\ ;\ c_2, \rho, h, t, \kappa\rangle \xrightarrow{\tau} \langle c_1'\ ;\ c_2, \rho', h', t', \kappa'\rangle}$$

E-SeqEmpty
$$\frac{}{\langle \varepsilon\ ;\ c, \rho, h, t, \kappa\rangle \to \langle c, \rho, h, t, \kappa\rangle}$$

Two environments and heaps (ρ_1, h_1) and (ρ_2, h_2) are low-equivalent, written $\rho_1 =_L \rho_2$ iff

$\forall x.\ \kappa(\rho_1(x)) = \kappa(\rho_2(x)) \wedge (\kappa(\rho_1(x)) = \mathbf{0} \Rightarrow \mathit{toVal}(h, \rho_1(x)) = \mathit{toVal}(h, \rho_2(x)))$, $\mathrm{dom}(h_1) = \mathrm{dom}(h_2)$, $\forall r \in \mathrm{dom}(h_1).\ \kappa(h_1(r)) = \kappa(h_2(r)) \wedge (\kappa(h_1(r)) = \mathbf{0} \Rightarrow \mathit{toVal}(h_1, h_1(r)) = \mathit{toVal}(h_2, h_2(r)))$.

In 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 $\kappa_0 = (0, 0, 0)$. We define an safety property $I_E(\rho_0, h_0) \subseteq \mathit{Stmt}$ on extracted programs as follows: $c \in I_E(\rho_0, h_0) :\Leftrightarrow (\forall \rho, h.\ (\rho, h) =_L (\rho_0, h_0) \wedge \langle c, \rho, h, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle c', \rho', h', t', \kappa'\rangle \wedge (c' = \mathbf{sink}(e) \vee c' = \mathbf{sink}(e)\ ;\ c_2) \Rightarrow \bigsqcup(e, h) = \mathbf{0})$. Moreover, we note that only straight-line programs are extracted for explicit secrecy and such programs trivially preserve low equivalence of environments and heaps.

Additionally, we define the predicate $B(\rho_0, h_0, \rho_0', h_0') \subseteq \mathit{Stmt}$ where $c_e \in B$ iff whenever $(\rho_0, h_0) \leq (\rho, h)$, $(\rho_0, h_0) =_L (\rho, h)$, and $\langle c_e, \rho, h, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \varepsilon, \rho', h', t', \kappa'\rangle$, then $(\rho', h') \leq (\mathit{env}_0', h_0')$, where $(\rho, h) \leq (\rho_0, h_0)$ iff the counter of each value in ρ and h is related by $\sqsubseteq$ to the corresponding counter in ρ_0 and h_0. Formally, $(\rho_1, h_1) \leq (\rho_2, h_2)$ iff $\forall x\ v_1\ \kappa_1\ v_2\ \kappa_2.\ \rho_1(x) = v_1^{\kappa_1} \wedge \rho_2(x) = v_2^{\kappa_2} \Rightarrow \kappa_1 \sqsubseteq \kappa_2$, $\mathrm{dom}(h_1) = \mathrm{dom}(h_2)$, and $\forall a\ v_1\ \kappa_1\ v_2\ \kappa_2.\ h_1(a) = v_1^{\kappa_1} \wedge h_2(a) = v_2^{\kappa_2} \Rightarrow \kappa_1 \sqsubseteq \kappa_2$.

For the induction to succeed we show the stronger statement that whenever $(\langle c, \rho_0, h_0, [], \kappa_0\rangle, \mathbf{skip}) \xrightarrow{\tau_0}{}^{\star} (\langle c', \rho_0', h_0', [], \kappa_0'\rangle, c_e')$, then $c_e' \in I_E(\rho_0, h_0)$ and $c_e \in B(\rho_0, h_0, \rho_0', h_0')$.

We prove this by induction on $(\langle c, \rho_0, h_0, [], \kappa_0\rangle, \mathbf{skip}) \xrightarrow{\tau_0}{}^{\star} (\langle c', \rho_0', h_0', [], \kappa_0'\rangle, c_e')$. The reflexive case is trivial. For the transitive case, assume $(\langle c, \rho_0, h_0, [], \kappa_0\rangle, \mathbf{skip}) \xrightarrow{\tau_0} (\langle c', \rho_0', h_0', t', \kappa_0'\rangle, c_e) \xrightarrow{\tau_0'} (\langle c'', \rho_0'', h_0'', t'', \kappa_0''\rangle, c_e')$. Per the induction hypothesis we have that $c_e \in I_E(\rho_0, h_0)$ and $c_e \in B(\rho_0, h_0, \rho_0', h_0')$. We show that $c_e' \in I_E(\rho_0, h_0)$ and $c_e' \in B(\rho_0, h_0, \rho_0'', h_0'')$ by induction on $\langle c', \rho_0', h_0', t', \kappa_0'\rangle \xrightarrow{\tau_0'} \langle c'', \rho_0'', h_0'', t'', \kappa_0''\rangle$. The main interesting cases are E-Assign, E-AssignField, and E-Sink.

Case E-Assign: We have that $c_e' = c_e\ ;\ x = e$. We have that $c_e' \in I_E(\rho_0, h_0)$ follows from the fact that any sink in c_e' is also reachable in c_e, hence the claim follows from the induction hypothesis.

To show $c_e' \in B(\rho_0, h_0, \rho_0'', h_0'')$, we note that if $\langle c_e', \rho, h, [], \kappa_0\rangle \xrightarrow{\tau_0}{}^{\star} \langle \varepsilon, \mathit{env}'', h'', t'', \kappa''\rangle$ then $\langle c_e', \rho, h, [], \kappa_0\rangle \xrightarrow{\tau_0}{}^{\star} \langle x = e, \rho', h', t', \kappa'\rangle$ and $\rho'' = \rho'(x := v_e)$, where v_e is the result of the assigned expression with incremented counters. We show that the label of x is still bounded by the corresponding label of x in ρ_0''. From the induction hypothesis we have that $(\rho', h') \leq (\rho_0', h_0')$. If the new label of x in (ρ_0', h_0') is not $\mathbf{0}$, then the claim follows trivially. If it is low, it follows from the previous fact that x also receives $\mathbf{0}$ in ρ'', h''.

Case E-AssignField is analogous.

Case E-Sink: In this case $c_e' \in B(\rho_0, h_0, \rho_0'', h_0'')$ follows easily from the induction hypothesis as the sink statement does not change the environment and heap. $c_e' \in I_E(\rho_0, h_0)$ follows trivially for sink statements already reachable in c_e. Since $c_e' = c_e\ ;\ \mathbf{sink}(e)$, we need to show that this also holds when reaching c_e'. Since the explicit flow count is still 0 in κ_0'', we have that this the label of e in (ρ_0', h_0') is $\mathbf{0}$. Since $(\rho', h') \leq (\rho_0', h_0')$, we have that therefore the label of e in ρ', h' is also $\mathbf{0}$, as desired.

Clearly whenever $c \in I_E(\rho_0, h_0)$, then c satisfies per-run non-interference wrt. ρ_0 and h_0 since low equivalence between memories is preserved and only low expression reach sinks without increasing the counter. □

In the following proof sketch, we define $\pi_{1,2}((a, b, c)) = (a, b)$ and overload $\mathbf{0}$ to be any n-tuple of 0s. The $\sqsubseteq$ relation is generalized similarly to arbitrary tuples.

Proof of Theorem B.2. For the induction to go through, we show the stronger property that whenever $(\langle c, \rho, h, [], \kappa_0\rangle, \bullet) \xrightarrow{\tau}{}^{\star} (\langle c', \rho', h', t', \kappa'\rangle, c_e)$, $(\rho, h) =_L (\rho_2, h_2)$, and $\pi_{1,2}(\kappa') = \mathbf{0}$, then:

- $\langle c_e, \rho, h, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \bullet \,;\, \mathbf{pop}^{length(t')}, \rho', h', t', \kappa_1'\rangle$, and $\pi_{1,2}(\kappa_1') = \pi_{1,2}(\kappa')$ (1) and
- There exist $\rho_2', h_2', t_2', \kappa_2'$ and S_2' such that $(\langle c_e, \rho_2, h_2, [], \kappa_0 \xrightarrow{\tau}{}^{\star} \langle \bullet \,;\, \mathbf{pop}^{length(t')}, \rho_2', h_2', t_2', \kappa_2'\rangle\rangle \wedge length(t') = length(t_2')) \vee (\langle c_e, \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \varepsilon, h_2', t_2', \kappa_2'\rangle \wedge \bigsqcup t' = \mathbf{H})$ (2).
- Moreover, $(\rho', h') =_W (\rho_2', h_2')$ $(\rho_2', h_2') \leq (\rho', h')$, and $t_2' \leq t'$. (3)

where $\rho_1 =_W \rho_2$ holds iff all values labeled low in both environments are equal in value and lists of levels xs and ys satisfy $xs \leq ys$ if there exists a suffix of length $length(xs)$ of ys such that all levels are pairwise related by $\sqsubseteq$. We write c^n for n copies of c composed with sequential composition.

We show this by induction on $(\langle c, \rho, h, [], \kappa_0\rangle, \bullet) \xrightarrow{\tau}{}^{\star} (\langle c', \rho', h', t', \kappa'\rangle, c_e)$. The reflexive case is trivial. For the transitive case, assume $(\langle c, \rho, h, [], \kappa_0\rangle, \bullet) \xrightarrow{\tau}{}^{\star} (\langle c', \rho', h', t', \kappa'\rangle, c_e) \xrightarrow{\tau'} (\langle c'', \rho'', h'', t'', \kappa''\rangle, c_e')$. From part (1) of the induction hypothesis we have that $\langle c_e, \rho, h, [], \kappa\rangle \xrightarrow{\tau}{}^{\star} \langle \bullet \,;\, \mathbf{pop}^{length(t')}, \rho', h', t', \kappa'\rangle$ (1′); from part (2) we obtain $\rho_2', h_2', t_2', \kappa_2'$ for which the corresponding disjunction also holds (2′); from (3) we have $(\rho', h') =_W (\rho_2', h_2')$, $(\rho_2', h_2') \leq (\rho', h')$, and $t_2' \leq t'$. We show (1 − 3) by induction on $(\langle c', \rho', h', t', \kappa'\rangle, c_e) \xrightarrow{\tau'} (\langle c'', \rho'', h'', t'', \kappa''\rangle, c_e')$; the interesting cases are E-Assign, E-AssignField, E-IfTrue/False, E-Sink, and E-Pop. We refer to the proof obligations as (1‴), (2‴), and (3‴).

Case E-Assign: (1″) follows trivially from the semantics. We have $c_e' = c_e[x = e \,;\, \bullet]$, $\rho'' = \rho'[x \mapsto v_e]$, where v_e is the result of evaluating e in ρ', h' and incrementing the counts appropriately.

We proceed by case distinction on (2′). If $\langle c_e, \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \bullet \,;\, \mathbf{pop}^{length(t_2')}, \rho_2', h_2', t_2', \kappa_2'\rangle$, then we have that $\langle c_e', \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau} \langle x = e \,;\, \bullet \,;\, \mathbf{pop}^{length(t_2')}, \rho_2', h_2', t_2', \kappa_2'\rangle \xrightarrow{[]} \langle \bullet \,;\, \mathbf{pop}^{length(t_2')}, \rho_2'', h_2', t_2', \kappa_2'\rangle$ where $\rho_2'' = \rho_2'[x \mapsto v_e^2]$, where v_e^2 is the result of evaluating e in ρ_2', h_2'. $length(t_2') = length(t')$ follows from this case in (2′) and the fact that assignments do not modify the label stack.

In the case where $\langle c_e', \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \varepsilon, \rho_2', h_2', t_2', \kappa_2'\rangle$, then we have that $\bigsqcup(t') \neq \mathbf{0}$. Therefore, we have that the counter of $\rho''(x)$ is not $\mathbf{0}$, therefore $(\rho_2'', h_2') =_W (\rho'', h'')$ and $(\rho_2'', h_2') \leq (\rho'', h'')$ follows trivially.

The other statements follow from the induction hypothesis: If $\bullet$ is not reached, then $c_e[c_2]$ matches the evaluation of c_e for any c_2.

Case E-AssignField: Analogous to E-Assign.

Case E-If: Without loss of generality, we only discuss the case where the **then** branch is taken. In this case, we first note that $\rho'' = \rho'$, $h'' = h'$ and $t'' = \kappa_e.t'$ where $[\![e]\!](\rho', h') = v_e^{\kappa_e}$; in particular note that $length(t'') = length(t') + 1$. For (1″) note that from (1′) it follows that $\langle c_e', \rho, h, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \mathbf{if}\,(e)\,\{\, \bullet \,\}\, \mathbf{else}\, \{\, \mathbf{pop}^{length(t')} \,\} \,;\, , \rho', h', t', \kappa'\rangle \xrightarrow{[]} \langle \bullet \,;\, \mathbf{pop}^{length(t'+1)}, \rho', h', t'', \kappa'\rangle$ as desired.

We proceed by case distinction on (2′). If $\langle c_e, \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \bullet \,;\, \mathbf{pop}^{length(t_2')}, \rho_2', h_2', t_2', \kappa_2'\rangle$ and $length(t_2') = length(t')$, then we have that $\langle c_e', \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \mathbf{if}\,(e)\,\{\, \bullet \,\}\, \mathbf{else}\, \{\, \mathbf{skip} \,\} \,;\, \mathbf{pop}^{length(t')}, \rho_2', h_2', t_2', \kappa_2'\rangle$. Note that for $[\![e]\!](\rho_2', h_2') = v_2^{\kappa_e^2}$ and $(\rho_2', h_2') \leq (\rho', h')$ we have that $\kappa_e^2 \sqsubseteq \kappa_e$.

To show (2″) we proceed by case distinction on $v_2 = \mathbf{tt}$. If $v_2 = \mathbf{tt}$, we have that $\langle \mathbf{if}\,(e)\,\{\, \bullet \,\}\, \mathbf{else}\, \{\, \mathbf{skip} \,\} \,;\, \mathbf{pop}^{length(t')}, \rho_2', h_2', t_2'', \kappa_2'\rangle \xrightarrow{[]} \langle \bullet \,;\, \mathbf{pop}^{length(t')+1}, \rho_2', h_2', t_2'', \kappa_2'\rangle$ where $t_2'' = \kappa_e^2.t_2'$. Since $\kappa_e^2 \sqsubseteq \kappa_e$, we have that $t_2'' \leq t''$; we also have that $length(t_2'') = length(t'')$ trivially.

In the case where $v_2 \neq \mathbf{tt}$, we have that $\langle \mathbf{if}\,(e)\,\{\, \bullet \,\}\, \mathbf{else}\, \{\, \mathbf{skip} \,\} \,;\, \mathbf{pop}^{length(t')}, \rho_2', h_2', t_2'', \kappa_2'\rangle \xrightarrow{[]} \langle \mathbf{skip} \,;\, \mathbf{pop}^{length(t')+1}, \rho_2', h_2', t_2'', \kappa_2'\rangle \xrightarrow{[]}{}^{\star} \langle \varepsilon, \rho_2', h_2', t_2''', \kappa_2'\rangle$ where t_2''' is a prefix of t_2'' and hence, $t_2''' \leq t''$.

Note that since $v_2 \neq v_e$ and $(\rho_2', h_2') =_W (\rho', h')$, we have that $\kappa'' \neq \mathbf{0}$ as required for this case.

where v_e^2 is the result of evaluating e in ρ_2', h_2'. $length(t_2') = length(t')$ follows from this case in (2′) and the fact that assignments do not modify the label stack.

In the case where $\langle c_e, \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \varepsilon, \rho_2', h_2', t_2', \kappa_2'\rangle$ and $\bigsqcup(t') \neq \mathbf{0}$, we have that then $\langle c_e', \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \varepsilon, \rho_2', h_2', t_2', \kappa_2'\rangle$ since $\bullet$ is not reached and hence replacing it with $\mathbf{if}\,(e)\,\{\, \bullet \,\}\, \mathbf{else}\, \{\, \mathbf{skip} \,\}$ does not affect the execution. Since $t_2' \leq t'$, we trivially have that then also $t_2' \leq \kappa_e.t'$. In both cases, the rest of (3″) follows trivially.

Case E-Sink: We have that $c_e' = c_e[\mathbf{sink}(e) \,;\, \bullet]$, $\tau' = [v]$ where $v^{\kappa_v} = [\![e]\!](\rho', h')$. Environments and the heap are unchanged. Moreover, we have that $(fst(\kappa''), snd(\kappa'')) = (fst(\kappa'), snd(\kappa'))$, since the flow counts are assumed to be 0. (1‴) follows easily.

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

We can therefore assume that $\langle c_e, \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \bullet \,;\, \mathbf{pop}^{length(t_2')}, \rho_2', h_2', t_2', \kappa_2'\rangle$ and $length(t_2') = length(t')$. Hence we also have that $\langle c_e', \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \mathbf{sink}(e) \,;\, \bullet \,;\, \mathbf{pop}^{length(t_2')}, \rho_2', h_2', t_2', \kappa_2'\rangle \xrightarrow{v_2} \langle \bullet \,;\, \mathbf{pop}^{length(t_2')}, \rho_2, h_2, t_2, \kappa_2'\rangle$. Since $\pi_{1,2}(\kappa'') = \mathbf{0}$, we also have that the label of v is $\mathbf{0}$; from $(\rho_2', h_2') \leq (\rho', h')$ we have that v_2 is also labeled $\mathbf{0}$. With $(\rho', h') =_W (\rho_2', h_2')$ this yields that $v = v_2$, concluding (2″). (3″) follows easily since heaps, environments, and the label stack are not modified by executing a sink statement.

Case E-Pop: By this case we have $c_e' = leaveBranch(c_e)$. (1″) follows easily. For (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 (ρ', h'). Assume now that $\langle c_e, \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \varepsilon, \rho_2', h_2', t_2', \kappa_2'\rangle$ and $\bigsqcup t' \neq \mathbf{0}$. We proceed by case distinction on this execution reaching the branch surrounding $\bullet$ in c_e. We denote this branch by $\mathbf{if}\,(e)\,\{\, c_1 \,\}\, \mathbf{else}\, \{\, c_2 \,\}$; WLOG assume that $c_2 = \mathbf{skip}$ and $c_1 = c_1' \,;\, \bullet$. Then, we show $\langle c_e', \rho_2, h_2, [], \kappa_0\rangle \xrightarrow{\tau}{}^{\star} \langle \bullet \,;\, \mathbf{pop}^{length(t'')}, \rho_2', h_2', tl(t_2'), \kappa_2'\rangle$: Since c_e terminated without reaching $\bullet$, we have that $[\![e]\!](\rho_2', h_2') \neq \mathbf{tt}^{\kappa_2}$ for any κ_2, and, since $c_2 = \mathbf{skip}$, we reach $\bullet \,;\, \mathbf{pop}^{length(t'')}$ trivially, concluding the induction. This stronger property then trivially implies non-interference of $c_e[\mathbf{skip}]$. □