Turnstile: Hybrid Information Flow Control Framework for Managing Privacy in Internet-of-Things Applications

Kumseok Jung
kumseok@ece.ubc.ca
The University of British Columbia
Vancouver, British Columbia, Canada

Mohanna Shahrad
mohanna@princeton.edu
Princeton University
Princeton, New Jersey, USA

Gargi Mitra
gargi@ece.ubc.ca
The University of British Columbia
Vancouver, British Columbia, Canada

Karthik Pattabiraman
karthikp@ece.ubc.ca
The University of British Columbia
Vancouver, British Columbia, Canada

Abstract

General awareness in privacy management has increased over the last decade, from consumers, companies, to governments. While cloud and mobile applications have taken steps forward in improving privacy management, the Internet-of-Things (IoT) domain has been behind in this aspect. Managing privacy in IoT applications is challenging, firstly because IoT applications handle data whose privacy implications change dynamically based on the information it contains. Second, the fragmented nature of the IoT ecosystem makes it difficult to apply a solution end-to-end. To provide a solution to privacy management in IoT, we design and implement Turnstile, a hybrid information flow control (IFC) framework. It identifies privacy-sensitive code paths through static taint analysis, and then integrates a dynamic information flow tracking (DIFT) mechanism into the application via selective code instrumentation. We evaluated Turnstile using 61 third-party IoT applications, and show that it can be an effective solution for managing the privacy of IoT applications.

CCS Concepts: • **Security and privacy → Information flow control**; *Domain-specific security and privacy architectures*; *Distributed systems security*; • **Software and its engineering** → *Integration frameworks*; Automated static analysis.

Keywords: Dynamic Information Flow Control, Privacy Management, Internet of Things

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769352

ACM Reference Format:
Kumseok Jung, Mohanna Shahrad, Gargi Mitra, and Karthik Pattabiraman. 2026. Turnstile: Hybrid Information Flow Control Framework for Managing Privacy in Internet-of-Things Applications. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3767295.3769352

1 Introduction

Privacy management in software systems is not just a feature – it is a requirement. Users are becoming more privacy-aware [45, 67], governments are putting regulations in place [1, 58], and companies are implementing more privacy-enhanced solutions [25, 74].

Information flow control (IFC) is a well-understood technique for managing end-to-end privacy in the domain of cloud [60, 69] and mobile computing [21, 30]. In these applications, IFC has been mechanized in various forms, from security-typed languages such as Jif [56], to dynamic information flow tracking (DIFT) systems such as TaintDroid [21]. However, it is challenging to adopt an existing approach to IFC in the Internet-of-Things (IoT) domain, because IoT applications pose three practical constraints.

First, the privacy implications of the data collected by IoT applications might vary during run-time. For example, a retail outlet might use a smart camera to capture video footage for multiple purposes, such as analyzing customer behavior or providing evidence of criminal activity. The same camera application might record an employee in one frame and a customer in another, each with distinct privacy needs. Thus, the privacy implications of the footage vary based on its content and intended use, and we need to dynamically restrict certain dataflows in the application. To effectively manage these variations during run-time, the IFC system must be both *dynamic and efficient.*

Second, different components of an IoT application are deployed on third-party platforms such as cloud-managed runtime environments or virtual machines [84]. It is challenging

to adopt a DIFT system built into the runtime environment [18, 31] or the OS [44, 81, 82], because all third-party platform providers would need to support it, which is outside the developer's control. Therefore, the IFC system must be deployable on the same runtime infrastructure as the application, without the need for modifying the runtime platform, i.e., the DIFT mechanism must be *platform-independent*.

Third, IoT applications are typically composed of modular components, each of which may be developed by different developers. It is challenging to adopt a language-based [29, 43] or a library-based [15, 32] IFC, as it would require rewriting of all the components written by third-parties. Therefore, an IoT-friendly DIFT system must integrate in a way that allows the continued use of existing code, i.e., the DIFT system must be *non-invasive*.

Each of the constraints above, when viewed individually, are not unique to the IoT domain. However, IoT represents a unique convergence point where the three practical design challenges are amplified, making existing DIFT solutions [15, 18, 29, 31, 32, 43, 44, 81, 82] infeasible and creating a compelling need for a new IFC system. This paper introduces Turnstile, a hybrid IFC system for managing privacy in an end-to-end fashion in IoT applications, that *satisfies the above three constraints. The key idea is to embed a self-contained, portable DIFT mechanism into only the privacy-sensitive code paths in the application through dataflow-aware selective code instrumentation.*

Turnstile makes the following three design decisions to satisfy the above constraints. ① Turnstile eliminates unnecessary run-time overheads, by first identifying privacy-sensitive code paths in the application that need to be managed through static taint analysis [1]. ② Turnstile's DIFT mechanism is designed to be inlined as part of the application code, such that it requires no changes to the existing runtime platforms. ③ Turnstile integrates the DIFT mechanism through automated code instrumentation, without requiring the developer to modify the original application code.

To the best of our knowledge, Turnstile is the first IFC system that achieves low performance overheads, while providing both platform-independence and non-invasive integration.

In summary, we make the following contributions:

- Propose a technique for reducing the run-time overhead of DIFT, by selectively managing privacy-sensitive code paths identified through static dataflow analysis. We developed a specialized static analysis tool for IoT applications, which efficiently identifies privacy-sensitive dataflows.
- Develop a portable DIFT mechanism that can be inlined in the application code, and thus deployed as part of the application. As a result, the DIFT-enabled application does not require any changes to the runtime infrastructure. Further, the DIFT mechanism is integrated automatically through code instrumentation, requiring no modification effort from developers.
- Perform a case study integrating Turnstile end-to-end into Node-RED [27], a popular third-party IoT framework. The case study demonstrates that Turnstile is non-invasive, as it transparently integrates with existing code, and that it is platform-independent, as it can be deployed on the same Node-RED runtime infrastructure without any modifications.
- Evaluate Turnstile using 61 third-party Node-RED packages. We compared Turnstile's privacy-sensitive dataflow detection performance against CodeQL [34], an industry-leading static analysis tool. We find that Turnstile's dataflow analyzer outperforms CodeQL's taint analyzer in terms of finding 3× *more privacy-sensitive dataflows*, while *being faster by* 67×.
- Evaluate Turnstile's run-time performance by comparing the end-to-end execution time of 27 Turnstile-managed applications against their original versions.

Our results are as follows. First, Turnstile's selective instrumentation significantly reduces the worst-case overhead from 153.8% to 15.8% at input rate of 30 Hz, which is the typical streaming rate of smart cameras. Further, selective instrumentation enables 22 of the 27 applications to have acceptable overheads, as compared to only 16 applications, with exhaustive instrumentation. Finally, we observe that the median performance overhead ranges from 0.2% to 26.8% across the range of input rates from 2 to 1000 Hz, indicating that Turnstile's DIFT mechanism is generally efficient. Thus, Turnstile is an efficient and dynamic IFC system that is both platform-independent and non-invasive.

2 Background and Threat Model

Information Flow Control (IFC). In IFC, each object is associated with a *privacy label*, and a set of *privacy rules* define the hierarchy of the privacy labels. For example, A rule stating that X can flow to Y, hereby expressed as $X \sqsubseteq Y$, establishes that the privacy class Y is "more private" than the privacy class X. The flow relation "$\sqsubseteq$" applies transitively, and thus it follows that, if $Y \sqsubseteq Z$, then $X \sqsubseteq Z$, establishing that the privacy class Z is the "most private" and X the "least private". Given this hierarchy, when there is information flow from one object to another, we either allow or disallow the flow based on the privacy labels of the objects involved. For instance, if an object m has the privacy label Y (denoted by $m \mapsto Y$), and variables a and b have the privacy labels X and Z respectively ($a \mapsto X$ and $b \mapsto Z$), then the assignment $b := m$ would be allowed while $a := m$ would be forbidden.

Additionally, privacy labels can be aggregated to produce a compound privacy label. For instance, if $a \mapsto P$ and $b \mapsto Q$, then the privacy label of $a + b$ is the compound label $\{P, Q\}$, expressed as a *set*. Following Denning's lattice model [19], we

[1] Turnstile currently targets JavaScript-based applications, which are popular in the IoT domain, but the design can be extended to other languages.

derive the rules between compound labels such that $X \sqsubseteq Y$ if $X \subseteq Y$. Thus, $P \sqsubseteq \{P, Q\}$ and $Q \sqsubseteq \{P, Q\}$.

Threat Model. The goal of Turnstile is to assist a developer to avoid unintentional mishandling of privacy-sensitive data. Modern IoT applications are rarely built with a robust privacy control from the ground-up. Instead, they are composed of first- and third-party components that are mostly privacy-agnostic – i.e., not designed with a specific privacy policy in mind. Turnstile enables the retrofitting of privacy controls onto an arbitrary collection of such components. Thus, the primary threat is the mishandling of data and privacy violations by trusted-but-fallible application components.

We do not consider adversaries that have compromised the integrity of the runtime environment or the network, or those using side channels to leak private information. Our threat model reflects the typical development environment of an IoT application developer, and is similar to prior work in this area [29, 32, 63].

In-Scope Threats. We do consider the following threats.

- *Policy violation by third-party code.* A third-party component's intended usage can cause privacy violations when used in certain application contexts. For instance, a component designed to communicate with a third-party Software-as-a-Service (SaaS) endpoint located in the USA might violate the GDPR [58] if processing the data of persons in the EU.
- *Unintentional data leakage.* A first-party component might unintentionally leak sensitive information across different parts of an application, due to the complexity of accounting for all the possible dataflows.
- *Dynamic policy enforcement.* The privacy implications associated with a piece of data depends on its run-time content and its application context. For example, a frame captured by a smart camera might have privacy implications that depend on: who is present in the frame, which subjects have provided consent, for what purpose the frame is used, where the frame is captured and stored, etc.

Out-of-Scope Threats. We do not consider the following threats.

- *Platform-level exploits.* Vulnerabilities in the underlying platform, e.g., the Node.js runtime or the OS, including those provided by a third-party Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) provider. Turnstile does not defend against attacks such as buffer overflows or code injection.
- *Malicious dependencies.* Third-party components that are intentionally malicious – e.g., a trojanized library designed to actively exfiltrate all data it can access.
- *Network-level attacks.* Adversaries who can monitor, intercept, or modify network traffic, such as through a man-in-the-middle attack.
- *Side-channel attacks.* Information leakage through side channels – e.g., timing, control flow.

3 Motivating Example

Considering the characteristics of IoT applications and their ecosystem, we identify three design constraints for an IoT-friendly IFC system, which existing systems [15, 18, 31, 32, 63] do not satisfy. This underscores the need for a new IFC system tailored to manage privacy in IoT applications. We describe an example application to highlight the constraints.

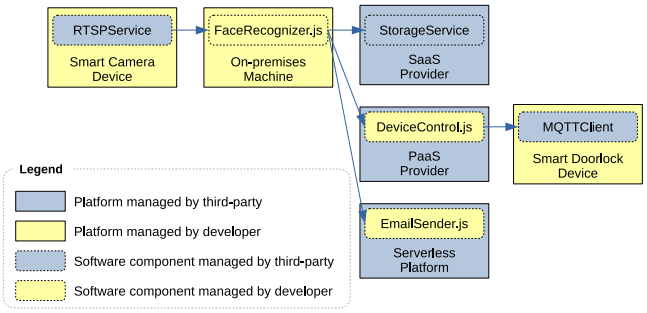

Figure 1. An example smart access control system, showing the software components and the platforms that host them.

Figure 1 illustrates the software components and platforms involved in a *smart access control system (SACS)*. This example is an adaptation of an existing open-source application [76], deployed across different runtime platforms [84]. This application recognizes faces and activates smart door locks to grant access only to authorized individuals. It also stores images of recognized faces for record-keeping and sends an email notification to administrators when specific persons arrive at the door. This application has six software components, distributed over six different runtime platforms. By *platform*, we mean the combination of hardware and software – virtual machine, operating system, and language runtime – that enables the execution of the application software. Three of these components are written (in JavaScript) by the developer, while the other three are written by device manufacturers and service providers (third parties).

- **RTSPService** (Real-Time Streaming Protocol) runs on the smart camera at the user's premises, delivering the real-time video stream to downstream components. This component is provided by the camera manufacturer and cannot be modified by the developer.
- **FaceRecognizer** runs on an on-premises machine. It identifies faces in video frames, checks for matches against a local database, and sends the frame along with the IDs of any detected faces to downstream components. Figure 2a shows a snippet of the component's code. In line 3, a `Scene` object is dynamically created from the raw frame, detecting any faces in it. It then computes a human-readable description (line 5), and sends a message to the DeviceControl component if the face is known. It also sends the frame to the EmailSender and StorageService components for further processing (line 13, 14).

```
1
2  socket.on("data", frame => {
3    const scene = analyzeVideoFrame(frame);
4    for (let person of scene.persons){
5      person.description =
6        person.action + " at " + scene.location;
7
8
9      if (person.employeeID){
10       deviceControl.send(person);
11     }
12   }
13   emailSender.send(scene);
14   storage.send(scene); });
```

(a) Original Code

```
1  const τ = (/* minified DIF Tracker and IFC policy */);
2  socket.on("data", frame => {
3    const scene = τ.label(analyzeVideoFrame(frame), "Scene");
4    for (let person of scene.persons){
5      person.description =
6        τ.binaryOp("+",
7          τ.binaryOp("+", person.action, " at "),
8          scene.location);
9      if (person.employeeID){
10       τ.invoke(deviceControl, "send", [ person ]);
11     }
12   }
13   τ.invoke(emailSender, "send", [ scene ]);
14   τ.invoke(storage, "send", [ scene ]); });
```

(b) Privacy-managed Code

Figure 2. Simplified code snippet from the FaceRecognizer component of the smart access control system described in Section 3. It receives a video frame, recognizes faces in the frame, and then sends the information to a downstream service based on the information extracted. (a) shows the original version, and (b) shows the privacy-managed version, instrumented by Turnstile (instrumentation is shown in bold).

- **StorageService** runs on the cloud, and is a cloud-based storage service (e.g., Amazon S3) that stores the received frames in a virtual directory. This software is managed by the SaaS provider, and interfaces with the rest of the application via a HTTP-based API.
- **DeviceControl** runs on a runtime platform managed by a PaaS provider (e.g., Heroku). Based on the face ID received, it sends a command to the connected smart doorlocks via MQTT, a popular publish-subscribe communication protocol.
- **MQTTClient** runs on the smart doorlock on the user's premises, and is written by the doorlock device manufacturer. It actuates the doorlock based on the command it receives via MQTT subscription.
- **EmailSender** runs on a serverless platform (e.g., AWS Lambda). It sends an email to the administrators, including the received frame as an attachment.

C1 Dynamic IFC with low overhead. In this application, the privacy of the data – i.e., the video frame (line 3 in Fig. 2a) – depends on the information extracted from it. For instance, it might be legal to store the frame containing an employee who has consented to it, but illegal to store that of an unexpected visitor. Similarly, company policies might permit a higher-ranking employee to receive emails about a lower-ranking employee but not vice versa. Since privacy implications vary with the information in a frame, privacy labels cannot be applied statically. Instead, each frame must be evaluated dynamically to determine its privacy label.

Static IFC methods, such as security-typed languages [56, 68, 73] or static dataflow/taint analysis [22, 23, 47] cannot handle dynamic privacy requirements. On the other hand, DIFT systems [18, 31, 32] can dynamically assign privacy labels but often suffer from high run-time performance overheads, sometimes up to two orders of magnitude [31], which makes them impractical. The main challenge, therefore, is to dynamically manage privacy labels, while minimizing the run-time overhead to make the IFC system practical.

C2 Platform-independence. We want the DIFT system for IoT applications to be *platform-independent* – i.e., it should not require the use of a special "DIFT-enabled" platform. In other words, the DIFT system should be deployable within the same infrastructure as the original application.

Many existing DIFT systems are implemented at the OS level [44, 81, 82] to manage information flow at the boundaries of OS resources like processes and sockets. Alternatively, there are also DIFT systems integrated into the language runtime [18, 31, 32], to control information flow between application objects. We refer to DIFT systems incorporated into the runtime *platform* beneath the application as *platform-level DIFT systems*. Platform-level DIFT has the advantage that it requires minimal or no modifications to the application.

However, it is challenging to adopt platform-level DIFT for the IoT domain. In the example, the `DeviceControl` component runs on a cloud-managed runtime, and `EmailSender` is deployed as a serverless function, both managed by third parties. To use platform-level DIFT, all PaaS providers must universally adopt the same DIFT mechanism, which is difficult to achieve in the IoT ecosystem. Therefore, an IoT-friendly DIFT system must be platform-independent.

C3 Non-invasiveness. Platform-independent DIFT can be provided in a new language [43] that compiles to an executable format compatible with the existing runtime environment. Alternatively, it can be offered as a user-level library [15], providing abstractions for controlling the information flow. We refer to such systems as *application-level DIFT systems*. However, existing application-level DIFT systems are *invasive*, either requiring the application to be rewritten in a new language or requiring significant code modifications.

In the example application, if we had to use a new language or use a special API, then we would have to rewrite

the three components `FaceRecognizer`, `DeviceControl`, and `EmailSender`. Additionally, all three components import other third-party modules, which themselves depend on many other modules. All those third-party modules would have to be rewritten, which is impractical. Therefore, the DIFT system should be *non-invasive*, allowing the existing application code and third-party library code to be used.

Design Goals. For IoT applications, the above three constraints are amplified in tandem, and a practical IFC system should aim to satisfy all three. Therefore, we propose Turnstile, an IFC system which has three design goals: (C1) have low overhead during run-time, (C2) be deployable without changing the runtime platforms, and (C3) integrate with existing code without modification efforts.

4 Approach

The key innovation of Turnstile [2] lies in its combination of *dataflow-aware selective code instrumentation* and the *self-contained DIFT system* embedded into the application code. Turnstile integrates both static and dynamic IFC techniques in a *hybrid approach* to track *only* the potentially privacy-sensitive dataflows. This hybrid approach enhances coverage compared to using static analysis alone, and significantly reduces run-time overhead compared to tracking all dataflows.

Static IFC. Turnstile performs a context-sensitive static taint analysis to identify code paths that are *potentially privacy-sensitive* (§ 4.2). Once identified, only those objects that are handled along these sensitive code paths are tracked by the runtime DIFT system. By selective tracking, Turnstile avoids the overhead of tracking all objects in the application, thereby satisfying the efficiency constraint (C1).

Dynamic IFC. The Turnstile DIFT system (§ 4.4) is designed as a portable, self-contained system that has no dependencies on special runtime APIs, platform features, or third-party software, and is fused transparently into the application. As a result, the DIFT-enabled application runs on the same platform as the original application, thus satisfying the platform-independence constraint (C2). Turnstile's DIFT system is integrated through automated code instrumentation. Turnstile takes the original application source code and produces a DIFT-enabled version of the application. This approach enables IFC on existing codebases without requiring an end-to-end rewrite of the application and its dependencies, thus satisfying the non-invasiveness constraint (C3).

4.1 System Architecture and Workflow

We provide a birds-eye view of Turnstile from a developer's perspective, by describing how a developer configures and deploys an application using Turnstile. Figure 3 illustrates the end-to-end workflow. Note that Turnstile works on applications written in JavaScript, which is the language of choice for several mainstream IoT frameworks [2, 4, 27]. However, Turnstile's approach can also be extended to other languages.

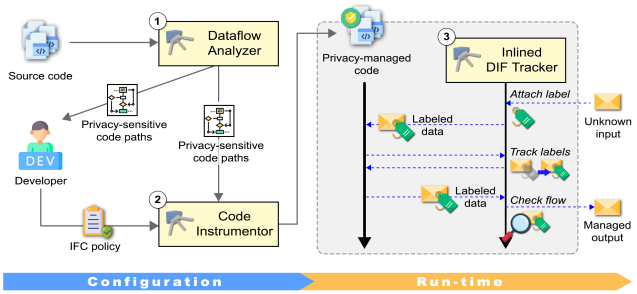

Figure 3. Application configuration and deployment workflow showing the three key components of Turnstile

Prior to deployment, the developer uses the *Dataflow Analyzer* (§ 4.2, ① in Fig. 3) to identify privacy-sensitive source and sink objects in the application code and any third-party code used. For each privacy-sensitive source and sink, the developer must provide a *privacy label function*, which produces the *privacy label* of a given object during run-time. Additionally, the developer must define the hierarchy of labels in the form of *privacy rules* (§ 2). Collectively, we refer to the set of privacy labels and rules as the *IFC policy* (Fig. 4). The IFC policy is the only part that requires active input from the developer when using Turnstile, and is written once for the whole application (discussed in § 4.3 and § 4.6).

Based on the IFC policy and the dataflow analysis, the *Code Instrumentor* (§ 4.3, ② in Fig. 3) automatically produces a privacy-managed version of the application, with the necessary instrumentations added for controlling information flow at run-time. It injects various API calls to the *Inlined Dynamic Information Flow Tracker (DIF Tracker)* (§ 4.4, ③ in Fig. 3), selectively along the privacy-sensitive code paths identified by the Dataflow Analyzer.

The developer then deploys the instrumented application – now managed by the *DIF Tracker* during run-time – on the same runtime infrastructure as the original application. The DIF Tracker operates within the application, tracks the privacy labels of different objects in the program during run-time, and flags any violations of the IFC policy.

4.2 Dataflow Analyzer

The *Dataflow Analyzer* identifies the parts of the code that might *potentially handle privacy-sensitive data*. To identify privacy-sensitive code paths, we perform a static taint analysis, targeting all Input-Output (I/O)-related POSIX APIs as taint sources and sinks. For example, an object created upon reading from a file or a socket is a *source object*. Similarly, an object passed to a file-write or a socket-write is a *sink object*. We target the POSIX API, because from the perspective of the application, that is where data enters and leaves. I/O

[2] Code available at: https://github.com/DependableSystemsLab/Turnstile-EuroSys26

interfaces are, in fact, common taint sources and sinks in most security-focused taint analyses [52, 57]. However, in contrast to prior work that targets specific function calls or I/O resources [47, 77, 85], our strategy is to cast a wide net and consider all POSIX interfaces as taint targets, to account for the wide variety of I/O access patterns in IoT applications.

We developed our own static taint analyzer instead of leveraging existing tools such as CodeQL [34] or ODGen [47], for three main reasons. First, existing tools did not perform an accurate, context-sensitive inter-procedural analysis. Second, except for CodeQL, most tools had pre-defined taint sources and sinks that were difficult to modify to capture all POSIX interfaces. Third, existing tools were often slow, with some taking tens of seconds to complete the analysis. To overcome these limitations, we optimized the dependence graph construction and taint tracking processes, and incorporated a richer context-sensitive inter-procedural analysis, using domain knowledge of commonly used JavaScript libraries.

4.3 Code Instrumentor

The *Code Instrumentor* selectively injects the appropriate *DIF Tracker* API calls along the code path between the source and sink objects in the application. Figures 2a and 2b show the original and the instrumented code for the example application, respectively. Starting from the `scene` object in line 3, Turnstile instruments only the code along the privacy-sensitive path previously traced by the Dataflow Analyzer. It injects API calls into expressions involving dataflows, such as assignments, variable declarations, and function calls. We summarize the relevant API methods in Table 1. During run-time, the instrumented application invokes these API methods to interact with the *DIF Tracker* (§ 4.4). For example, a call to the `label(target, labeller)` function is injected for each object to be managed. The DIF Tracker itself (i.e., the implementation of the API) is also injected in the application code (line 1 in Fig. 2b), along with the IFC policy provided by the developer.

```
1  { labellers: {
2      Scene: { persons: { $map: item =>
3        (item.employeeID ? "employee" : "customer") } } },
4    rules: [ "employee -> customer",
5             "customer -> internal" ],
6    injections: [
7      { line: 2, object: "scene", labeller: "Scene" } ] }
```

Figure 4. Example of an IFC policy, written for the example application in Figure 2

The API calls injected into the application are parameterized by the IFC policy provided by the developer (sample shown in Fig. 4). It consists of a set of *label functions*, a set of *privacy rules*, and a mapping between privacy-sensitive objects and the label functions. A label function $l(x) : V \rightarrow L$ returns a privacy label $p \in L$ based on the current value of object $x \in V$, where V is the set of all run-time objects and L is the set of all privacy labels. A mapping $n \mapsto l(x)$ between an object n and $l(x)$ indicates the label function to be used for the given object n.[3] During run-time, when the `label` function is invoked, Turnstile evaluates the privacy label of each managed object v by invoking the user-defined label function $l(v)$ associated with v. Thus, Turnstile supports *value-dependent privacy labels* [86], i.e. labels that are dynamically computed from the run-time values of the objects to be labelled. If an object needs to be *declassified* or *endorsed* to some privacy label Q, the developer can provide a label function that always returns Q regardless of the given v.

To determine whether a flow is allowed between two labelled objects, Turnstile builds a directed acyclic graph (DAG) that represents the hierarchy of privacy labels based on the set of privacy rules defined in the IFC policy, such as:

$$\texttt{employee} \rightarrow \texttt{customer} \rightarrow \texttt{internal}$$

In this DAG, each label is a node, and the direction of the edges indicates the hierarchy between labels, where the downstream node indicates a higher privacy level. If a cycle is detected while constructing the DAG, the IFC policy is considered invalid, and Turnstile informs the developer.

4.4 Inlined Dynamic Information Flow Tracker

After instrumentation, the application initializes a *DIF Tracker* instance at startup (line 1, Fig. 2b). The DIF Tracker dynamically assigns privacy labels to the managed objects based on their values at run-time, and then tracks the changes and transfers of these labels as the objects are processed by the application. Whenever a privacy-managed object flows into a privacy-managed sink, the DIF Trackers checks the privacy rules and either allows or disallows the flow. We discuss the DIF Tracker's three main functionalities below.

Dynamically attaching privacy labels. As shown in Figure 2b, the instrumented code invokes the DIF Tracker method `label` for the `scene` object to evaluate its value-dependent privacy label. It uses the label function `"Scene"` provided in line 2 of the IFC policy in Figure 4. The mapping in line 7 indicates that this label function should be applied to the `scene` object in line 3 of the original code in Figure 2a. The inner label function `$map` (line 2 in Fig. 4) is applied to each element of the `scene.persons` array, returning a privacy label for each element based on its value. In our example application, the privacy label would be either `"employee"` or `"customer"` depending on whether the given element in `scene.persons` has an `employeeID` property.

Tracking privacy-sensitive information flow. To track the movement of the privacy labels of the objects, the DIF Tracker maintains a global map of every tracked object and their privacy labels. For reference-type objects, this is trivial;

[3] Strictly speaking, n is a node in the abstract syntax tree (AST) representation of the source code, which is a syntactic element, not a run-time object. We refer to it as an object for ease of discussion.

Method	Description
`label(target, labeller)`	Evaluate and attach the privacy label to the given `target` object, using the given `labeller` function.
`binaryOp(operator, left, right)`	Perform a binary operation between the two objects `left` and `right`, and assign a compound label to the resulting object.
`check(data, receiver)`	Check whether the privacy rules allow the given `data` to be passed to the `receiver`, by inspecting the privacy labels attached to `data` and `receiver` respectively.
`invoke(target, func, args)`	Check whether the given function arguments `args` can be passed into the function `func` according to the privacy rules, then invoke the function `func` with the arguments `args` if the rules allow, and finally assign a compound label to the returned value.

Table 1. Description of the main API methods of the runtime DIFT system of Turnstile. Other methods are omitted.

$$v \in \mathrm{V}, \quad P \in \mathcal{P}(\mathrm{L}), \quad \longrightarrow \subseteq \mathrm{V} \times (\mathrm{V} \times \mathcal{P}(\mathrm{L}))$$

$$\text{(label)}\frac{l(v) = P}{\mathrm{label}(v, l) \longrightarrow v \mapsto P}$$

$$\text{(binaryOp)}\frac{v_1 \odot v_2 = v_3 \qquad v_1 \mapsto P_1 \qquad v_2 \mapsto P_2}{v_1 \odot v_2 \longrightarrow v_3 \mapsto P_1 \cup P_2}$$

$$\text{(assignment)}\frac{v_2 \mapsto P_2}{v_1 := v_2 \longrightarrow v_1 \mapsto P_2}$$

$$\text{(invoke)}\frac{v_1(v_2, v_3) = v_4 \qquad v_2 \mapsto P_2 \qquad v_3 \mapsto P_3}{v_1(v_2, v_3) \longrightarrow v_4 \mapsto P_2 \cup P_3}$$

Figure 5. Semantic rules for some of the expressions that create or manipulate a privacy label.

we can simply use the object itself as the key in the map. However, the problem is tricky for value-types, as different value-type *instances* with the same value would map to the same privacy label, even though the two instances represent two different pieces of information. Therefore, Turnstile wraps value-types in a container object. The wrapped values are unwrapped upon writing to a sink object, so that the values are available natively to external interfaces receiving the object from the application. While we implemented this in JavaScript, the same issue exists in other languages that distinguish between value-types and reference-types (e.g., Python, C#, Java), and the same solution applies to them.

Figure 5 shows the semantics of how labels are tracked by Turnstile. When the value of an object is computed from the values of other objects, the DIF Tracker determines the privacy label of the derived object based on the labels of the dependent objects. For example, when the assignment expression in line 5 in Figure 2b is evaluated, the resulting privacy label of `person.description` is the compound label of the labels of `person.action` and `scene.location`. It handles arithmetic operations using the `binaryOp` method (Table 1), as the privacy label of the result is derived from the arguments. Similarly, privacy labels are compounded in the `invoke` method, when a function is invoked. In other types of expressions involving a dataflow i.e., assignment, variable declarations, the operation over the privacy labels mirrors the operations over the objects.

Support for dynamic typing. A unique challenge in tracking objects in a dynamic language such as JavaScript is that properties can be dynamically created and deleted during run-time. Tracking the labels of properties that are created during run-time is not possible through static code instrumentation. We address this problem by using the `Proxy` construct available in JavaScript. The `Proxy` construct allows us to intercept property accesses before object properties are actually accessed. Turnstile automatically wraps every tracked object with a `Proxy`, so that it can track properties that are created or deleted during run-time.

Restricting policy-violating information flows. To ensure that a dataflow complies with the privacy rules, the DIF Tracker examines the privacy labels of the objects involved in the dataflow and checks whether a privacy rule permits the flow. For instance, in the expression `storage.send(scene)` (line 14 in Fig. 2b), before invoking `storage.send`, the DIF Tracker verifies that the dataflow from `scene` to `storage.send` is allowed based on their privacy labels at run-time.

To query the existence of a rule, the DIF Tracker traverses the DAG representing the privacy label hierarchy, discussed in Section 4.3. This DAG is static, and is computed once upon initializing the DIF Tracker. When comparing two labels, the DIF Tracker traverses the DAG to determine if there is a path between them. If no path is found, or if the receiving object has a lower-level privacy label, the flow is forbidden, and the DIF Tracker signals a privacy violation. Otherwise, it is allowed. The initial check for a pair of labels has a time complexity of $O(V + E)$ and space complexity of $O(V)$, where V is the number of labels (vertices) and E is the number of rules (edges). Subsequently, the result is cached, thus reducing the time and space complexity of subsequent checks to $O(1)$.

4.5 Support for JavaScript Features

Turnstile supports the ES6 (ECMAScript 2015) [35] syntax, including `class` declarations, spread expressions (e.g., `...args`), and arrow functions. By Turnstile's *coverage* of

JavaScript, we mean the extent to which Turnstile can trace the dataflow over different types of JavaScript expressions and code patterns during the static taint analysis.

Turnstile covers dynamic function calls and dynamic property access that uses the bracket syntax (e.g., `foo[x](y)`) through sound over-approximation [37]. When Turnstile encounters such dynamic property access, it traces backwards to find all the places where `foo[?]` was assigned regardless of the actual value of `x`. Dataflows across higher-order function calls and closures (e.g., `x => (y => x + y)`) are tracked through context-sensitive points-to analysis [37]. When a higher-order function is called to create a closure, Turnstile tracks both the higher-order function used to create the closure, and the objects captured by the closure. Turnstile covers `Promise` objects that are created using its constructor `new Promise(callback)`. The created `Promise` object is treated as the returned value of the callback. The same idea applies to `await` expressions – i.e., `await foo` is treated as `foo` for the sake of dataflow analysis. However, we do not currently cover chained `Promise`s, and we have not encountered them in the applications we studied. In future work, we can incorporate Functional Dependency Graphs (FDGs) [40] to cover chained `Promise`s. Turnstile does not cover `eval` because modern JavaScript discourages its use. None of the applications we studied (§ 6) uses `eval`.

4.6 Limitations and Discussion

Implicit Flows. It is important to track *implicit flows*, as control flow logic can be used as side channels to leak private information. A dynamic language such as JavaScript allows complex control flows involving dynamic function creation and event-triggered function invocation.

Turnstile currently tracks only explicit flows, similarly to several prior work on JavaScript taint analysis [10, 41, 71]. More specifically, Turnstile tracks explicit flows along all branches of conditional statements, and along all control flow paths that depend on a dynamic value. What Turnstile does not track is the leakage of information through observing which branch in the control flow logic was taken – e.g., an adversary deducing whether an authorized person was in a frame by observing whether the door opened. Tracking implicit flows requires significant improvements to the dataflow analysis and the DIFT mechanism. In future work, we can incorporate various techniques for tracking implicit flows as demonstrated in prior work [20, 31, 36].

Writing and Maintaining IFC Policy. In Turnstile, developer input is required only in writing the IFC policy. We consider the cost of writing an IFC policy to be lower than that of rewriting applications or maintaining an infrastructure of DIFT platforms. Generally, one label function is associated with one *type*. Therefore, a developer writes as many label functions as there are number of object types found at the privacy-sensitive sources and sinks. Expressing the privacy rules involves writing a small DAG with as many nodes as there are privacy labels. Mapping the label functions to different sources and sinks is straightforward, as the code locations are already identified by the Dataflow Analyzer.

As the privacy requirements of an application evolve, the only part that the developer must maintain with respect to Turnstile is the IFC policy. The effort required to maintain the IFC policy depends on the change in the application. There are three possibilities. (1) If there is a change in the application code, but no changes in the privacy of data handled, no changes would be required to the IFC policy. Only the code would be re-instrumented by Turnstile, which is automated. (2) If a new source or sink is introduced, the developer might need to define a new label function or assign an existing label function to the new source or sink. This involves writing a single label function, and indicating the new injection point. (3) If there is a change in the privacy implication of the data handled (e.g. new company policy now allows `"vendor"` data to be stored), then new privacy labels and rules might need to be introduced.

5 Case Study: Network Video Recorder

Case Study Objective. We used Turnstile to manage the privacy of an application developed in Node-RED [27], a popular IoT framework based on the flow-based programming model [54]. Our goal was to assess the practicality of using Turnstile from the developer's perspective.

Node-RED enables a developer to build an application by connecting together a set of modular components (referred to as "*nodes*") as a DAG (referred to as a "*flow*"). A Node-RED developer typically imports third-party nodes, which imposes the non-invasiveness constraint. Further, Node-RED applications are often deployed on third-party managed runtimes, which imposes the platform-independence constraint. Thus, we can assess Turnstile's effectiveness in satisfying both design constraints in this case study. While we have focused on Node-RED for this study, Turnstile is independent of Node-RED, and can be applied to other frameworks.

Target Application. We used a Node-RED flow that was publicly available from their official website – a *Network Video Recorder (NVR)* [76], which is similar to the example application we introduced in Section 3. This flow captures footage from smart cameras, stores it in a database, and sends images via email. To add more dynamism, we included an additional node to perform facial recognition on the image data captured from the cameras. The face recognition node was also sourced from a third-party repository [49].

The four main parts of the NVR flow are:

1. **Frame Capture** captures frames from video input sources, such as RTSP streams or video files.
2. **Face Recognition** uses the Deepstack Face Recognition API [49] to recognize pre-registered faces in a given frame, and attaches the face IDs identified.

```
1  function faceRecognition(msg, config, server) {
2    /* omitted */
3    let original = msg.frame;
4    deepstack.faceRecognition(original, server, config.
         confidence)
5    .then(async result => {
6      msg.payload = result.predictions;
7      /* omitted */
8      resolve(msg); });  }
```

(a) Snippet from the Face Recognition node

```
1  this.on("input", function(msg, send, done){
2    /* omitted */
3    let sendopts = {
4      to: msg.to,
5      attachments: msg.payload
6    };
7    smtpTransport.sendMail(sendopts, function(error, info){
8      /* omitted */ });  });
```

(b) Snippet from the Email Notification node

Figure 6. Face Recognition and Email Notification nodes in NVR, showing only the relevant parts. Label injection is shown in bold.

```
1  { labellers: {
2      onRecognize: { predictions: {
3          $map: item => {
4            let employee = getEmployeeById(item.userid);
5            return [ employee.region, employee.level ];
6          } } },
7      mailer: { sendMail: {
8          $invoke: (object, args) => {
9            return getEmployeeByEmail(args[0].to).level;
10         } } },
11     nodeRegion: { mydb: node => node.settings.region } },
12   rules: [ "US -> EU", "L1 -> L2", "L2 -> L3" ],
13   injections: [
14     { file: "face-recognition.js", line: 5,
15       object: "result", labeller: "onRecognize" },
16     { file: "email-notification.js", line: 7,
17       object: "smtpTransport", labeller: "mailer" },
18     { file: "frame-storage.js", line: 44,
19       object: "node", labeller: "nodeRegion" } ] }
```

Figure 7. Example of an IFC policy for the NVR flow

3. **Frame Storage** stores the frames processed by the face recognition node in an SQLite database.
4. **Email Notification** sends a set of frames and the associated faces IDs, to one or more recipients.

The NVR flow requires DIFT due to the dynamism present in three places. First, different frames can have different privacy labels depending on the faces captured. Second, the legality of storing a frame can depend on the geographic location of the database server. Third, the email recipients can have varying permission levels to view different frames.

Using Turnstile as a developer. Assume that the NVR flow is used in the context of worksite security by a large corporation. The faces of employees would be captured, stored, and sent to different employees in managerial positions. There are two types of privacy control we want to enforce. First, the faces of European Union (EU) residents are saved only in databases located in the EU, to be GDPR-compliant [58]. Second, no employee with a lower rank receives emails containing the faces of employees with a higher rank.

The developer must express the above two requirements in the IFC policy (Fig. 7), describing how to label different objects and the hierarchy of different privacy labels. In NVR, three types of objects must be labeled: ⓐ the frame containing a face (source), ⓑ the function for storing the frame (sink), and ⓒ the function for sending emails (sink).

Figure 6a shows the relevant parts of the Face Recognition node where a label is assigned to the captured frame. In line 4, the method `faceRecognition` is called, which returns a list of recognized faces asynchronously as the argument `result` in the callback function in line 5. The `result` object contains a property `predictions`, which is an array of objects with a field `userid`. Thus, the privacy label of the `predictions` array depends on the `userid` of the objects in it. The label function we use to assign label to the `predictions` array is shown in line 3 in the IFC policy (Fig. 7). The function is invoked on each of the elements of the `predictions` array. It looks up the employee data using the given `userid`, and returns a label set containing the region (either `US` or `EU`) and the employee level (one of `L1`, `L2`, and `L3`). The hierarchy of labels is established in line 12 in the IFC policy (Fig. 7), indicating that the label `EU` has a higher privacy level than the label `US`. `L3` is higher than `L2`, and `L2` is higher than `L1`.

Figure 6b shows parts of the Email Notification node, where a label is assigned to the function for sending an email. The `sendMail` function in line 7 is a sink, and the argument `sendopts` contains the captured frame in the `attachments` property. The label of the `sendMail` function should be determined dynamically based on the recipient of the email, which is provided in the `to` property of the `sendopts` object. Based on the label of the `sendMail` function at the time of invocation, and the label of the `attachments` object, Turnstile decides to allow or disallow the function invocation. For example, if the `attachments` object has the label `L3`, and the recipient is an employee of level `L2`, Turnstile should prevent the email from being sent. Line 8 in the IFC policy (Fig. 7) shows the label function used to assign label to the `sendMail` function. The label function is called when the `sendMail` function is invoked, and returns a label derived from the argument passed, which is the `sendopts` object. It looks up the employee data of the recipient using the email address, and returns the employee level as the label. We omit the steps for the Frame Storage node as they are similar.

We use the Code Instrumentor to instrument the source code of all the nodes in the flow. The instrumented code is then deployed on the same infrastructure as the original.

6 Evaluation

Evaluation Objective. We evaluated Turnstile in two parts – ① ability to identify the privacy-sensitive code paths in an application's code. ② run-time performance overhead of Turnstile's DIFT mechanism. Therefore, in our evaluation, we aimed to answer the following research questions (RQ):

- RQ1. How effective is Turnstile in identifying privacy-sensitive paths?
- RQ2. Does Turnstile's selective tracking significantly reduce the run-time overhead?
- RQ3. What is the run-time overhead of Turnstile?

In Part 1, we answer RQ1 by comparing Turnstile's performance against an industry-standard static analysis engine called CodeQL [34], used by Github. In Part 2, we answer RQ2 and RQ3 by comparing the execution time of the privacy-managed applications with those of the original versions.

Target Applications. In both parts of the evaluation, we used 61 Node-RED applications we found on Github as the experimental targets. We crawled Github and searched for potential target applications by looking up code signatures that are characteristic of each of the popular IoT frameworks. For example, to find Node-RED applications, we searched for "`RED.nodes.createNode`", which all Node-RED applications must call. We considered the following popular IoT frameworks: Node-RED [27], Azure IoT [3], HomeBridge [5], OpenHAB [6], SmartThings [4], and AWS Greengrass [2]. Table 2 shows the results of our search.

Framework	Search Results	Number of Repositories
Node-RED [27]	2676	677 (58.9%)
Azure IoT [3]	727	357 (31.1%)
HomeBridge [5]	171	57 (5.0%)
OpenHAB [6]	70	14 (1.2%)
SmartThings [4]	42	29 (2.5%)
AWS Greengrass [2]	27	15 (1.3%)

Table 2. Number of publicly available repositories on Github found for popular IoT frameworks.

We found that Node-RED was the most popular IoT framework, accounting for 58.9% of the total of 1,149 repositories found, which is almost twice as many repositories as Azure IoT, the second most popular framework. Therefore, we decided to target JavaScript applications written for Node-RED. Out of 677 Node-RED repositories, we selected 61 that had the most number of "stargazers" and "watchers", which indicate their maturity and popularity.

6.1 Part 1: Static Code Path Selection

In this part, we evaluate Turnstile's effectiveness in identifying privacy-sensitive code paths in the target applications.

Methodology. For each target application, we perform the dataflow analysis using Turnstile, and record the number of privacy-sensitive source objects, sink objects, distinct code paths between source and sink objects. We run the same analysis using CodeQL [34]. We compare against CodeQL as it satisfied the following criteria: ① work with arbitrary JavaScript applications with minimal effort, ② support customizing taint targets, and ③ produce results that explicitly map the tainted dataflow paths back to the code locations. We considered other tools such as ODGen [47], NodeMedic [10], and PDG-JS [23], but they did not satisfy these criteria.

CodeQL Query Construction. To use CodeQL's taint tracking feature, we wrote a custom CodeQL query that performs the same taint analysis as Turnstile's. Specifically, we extended CodeQL's `TaintTracking::Configuration` class to define a custom taint analysis configuration equivalent to that used in Turnstile, and defined custom `DataFlow::Node` classes that select the taint sources and sinks using the same selection criteria as Turnstile's. Figures 8 and 9 shows parts of our implementation for the two custom classes used for performing the taint analysis with CodeQL.

```
1  class TrackDownstreamConfiguration extends TaintTracking::
       Configuration {
2    TrackDownstreamConfiguration() { this = "
         TrackDownstreamConfiguration" }
3
4    override predicate isSource(DataFlow::Node source) {
5      source instanceof IOSource
6      or source instanceof ExpressSource
7      or source instanceof NodeRedSource
8    }
9
10   override predicate isSink(DataFlow::Node sink) {
11     sink instanceof IOSink
12     or sink instanceof ExpressSink
13     or sink instanceof NodeRedSink
14   }
15 }
16
17 from TrackDownstreamConfiguration dataflow, DataFlow::
       PathNode source, DataFlow::PathNode sink
18 where dataflow.hasFlowPath(source, sink)
19 select source, source, sink, "IO-to-IO Source"
```

Figure 8. One of the taint tracking configurations used to customize the CodeQL taint tracking query.

Figure 8 shows the taint tracking configuration we defined by extending the CodeQL `TaintTracking::Configuration` class. This is the "main" query file that we run using CodeQL. The purpose of this class is to define the taint sources and sinks. We achieve this by overriding the *predicates* `isSource` and `isSink`, shown in lines 4 and 10, respectively. The `isSource` and `isSink` predicates are evaluated for each node in the abstract syntax tree (AST) of the source code. The predicate logic is straightforward – it checks whether the given AST node matches the node type that we have defined. For example, `isSource` checks whether the given AST node is of type `IOSource`, `ExpressSource`, or `NodeRedSource`, which are custom nodes that we have defined. Lines 17 - 19 express the actual *query* to

```
1  class IOSource extends DataFlow::Node {
2    IOSource() {
3      this = DataFlow::globalVarRef("process").
           getAPropertyRead("stdin").getAMemberCall("on").
           getCallback(1).getParameter(0)
4      or this = DataFlow::moduleMember("fs", "createReadStream
           ").getACall().getAMemberCall("on").getCallback(1).
           getParameter(0)
5      or this = DataFlow::moduleMember("child_process", "exec"
           ).getACall().getCallback(1).getParameter(1)
6      or this = DataFlow::moduleMember("net", "connect").
           getACall().getAMemberCall("on").getCallback(1).
           getParameter(0)
7      /* omitted */
8    }
9  }
10
11 class IOSink extends DataFlow::Node { /* omitted */ }
12 /* omitted */
```

Figure 9. Snippet from the custom CodeQL library that defines different taint sources and sinks

be made, which tells CodeQL to perform the taint analysis using the given configuration.

We defined the custom AST nodes in a CodeQL library as shown in Figure 9, which shows a part of the implementation of the `IOSource` node. We extended the CodeQL `DataFlow::Node` class to define a custom node. In the body of the constructor in line 2, we defined the selection criteria. Figure 9 shows four selection criteria as examples, but we have defined a total of 14 selectors for `IOSource`, which covered the range of possible sources in the applications we studied[4]. Similarly, we have defined five other classes: `IOSink`, `ExpressSource`, `ExpressSink`, `NodeRedSource`, and `NodeRedSink`. The `Express*` and `NodeRed*` classes cover *Express.js* [26] and Node-RED respectively, which are commonly used frameworks.

Establishing ground-truth. Finally, we manually identified the privacy-sensitive code paths in all 61 target applications to compare them against those found by both Turnstile and CodeQL. Manual inspection was needed because there was no ground truth listing the privacy-sensitive code paths in these third-party applications, and we found no automated tool for identifying them.

Code Path Detection. Figure 10 shows the distribution of the number of privacy-sensitive code paths found by Turnstile and CodeQL in the 61 Node-RED applications, compared against the number of code paths found manually, indicated by the green line. Out of the 285 privacy-sensitive code paths found manually across 61 applications, Turnstile was able to identify more than three times as many code paths as CodeQL, finding a total of 190, compared to 52 found by CodeQL. We have manually verified that the code paths found

[4]For complete coverage over Node.js built-in libraries, additional selectors need to be defined.

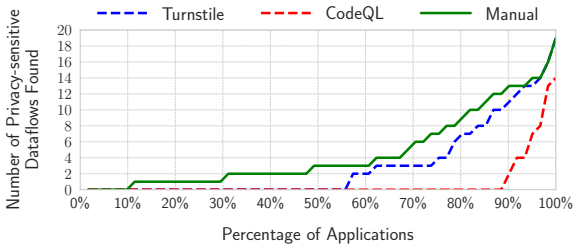

Figure 10. Distribution of the number of privacy-sensitive dataflows detected per application, by Turnstile and CodeQL.

by both solutions were correctly identified – i.e., the code paths indeed had a data flow from or to an I/O resource.

There were five applications for which both Turnstile and CodeQL identified code paths. Out of the five applications, Turnstile reported more code paths in three applications and CodeQL reported more paths in another application. In the fifth application, both Turnstile and CodeQL identified exactly the same code paths. *There were 22 applications for which Turnstile found privacy-sensitive code paths but CodeQL did not find any.* This is due to Turnstile's better support for type-sensitive interprocedural analysis compared to CodeQL. Although JavaScript is dynamically typed, we implemented type-inference mechanisms to deduce the type of the arguments passed dynamically to a function invocation. As a result, Turnstile is able to track privacy-sensitive dataflows more accurately across function calls. There were two applications in which CodeQL outperformed Turnstile. We investigated those two applications and found that CodeQL's analysis does better due to Turnstile's inability to track reflective code through the JavaScript prototype chain.

Further, there were 32 applications for which neither Turnstile nor CodeQL found privacy-sensitive code paths. Of these, 6 applications actually had no privacy-sensitive code paths, and 26 applications had one or more privacy-sensitive code paths, which both systems failed to detect. The most common pattern missed was when the privacy-sensitive data were exchanged through special framework APIs such as Node-RED's `RED.httpNode`, which is an HTTP server that handles privacy-sensitive objects such as `HttpRequest` and `HttpResponse`. Note that it cannot be statically inferred that `RED.httpNode` is indeed an HTTP server, since it is assigned dynamically by the Node-RED runtime. Thus, both tools missed dataflows involving `RED.httpNode`. To detect this kind of dataflows, additional domain-knowledge about framework APIs needs to be incorporated into the tools.

Computation Time. We measured the time taken to analyze an application and observed that Turnstile is an order of magnitude (~67x) faster than CodeQL, completing an analysis in 325 ms on average, and 1578 ms for the `nlp-js` application in the worst case. In contrast, CodeQL completed

an analysis in 59532 ms (59.5 seconds) on average, taking up to 724102 ms (12 minutes) for the `modbus` application.

The large difference between the processing times is due to CodeQL accounting for more types of object dependence than Turnstile, and compiling the dependence graph into an intermediate representation (IR). CodeQL is a general-purpose, polyglot analysis engine, which has more use cases than the Turnstile Dataflow Analyzer. On the other hand, Turnstile Dataflow Analyzer is a specialized tool that we built to detect specific dataflows that potentially violate privacy. Turnstile does not need to generate an IR and hence is faster.

Summary. We evaluate Turnstile by comparing it against an industry-standard analysis tool, CodeQL. (RQ1) Our evaluation shows that Turnstile is more effective than CodeQL at identifying the privacy-sensitive dataflows, *both in terms of the number of code paths found and the time for the analysis.*

6.2 Part 2: Run-time Performance Overhead

In this part, we assess the run-time overhead of Turnstile's DIFT mechanism, once the application has been instrumented. Out of the 61 applications we used in Part 1, we used only those 27 applications in which Turnstile identified at least one privacy-sensitive dataflow for this part, since Turnstile enables DIFT selectively only along those dataflows.

Methodology. For each application, we feed in an input workload containing 1000 messages. To emulate the continuous, periodic real-time processing that is characteristic of IoT applications, the messages are streamed in periodically at a fixed rate of f Hz (messages per second). We feed the same input stream into three different versions of an application: ① unmanaged, ② exhaustively-managed, and ③ selectively-managed. The unmanaged version is the original application, which is the baseline we compare against. The exhaustively-managed version has all the code paths instrumented, and the selectively-managed version has only the privacy-sensitive code paths instrumented. We measure the time t taken to process all the messages in the input stream for each of the 27 applications, when running the original (t_{og}), exhaustively-managed, and selectively-managed versions. We report the results in terms of the relative run-time (t/t_{og}), which we obtained as an average over 10 repeated runs, as the wall-clock execution time varies across applications. We consider a range of input rates (f), from 2 Hz to 1000 Hz; this covers a wide range of typical IoT devices, from smart meters [80] to *high frame rate* cameras [42].

Workload and Privacy Policy Preparation. The 27 target applications varied in terms of the kind of data they processed, which means they have different privacy demands in the real world. Therefore, to compare the applications fairly, we systematically generated the workload and the IFC policies. For each application, we inspected the application's Node-RED metadata files as well as the application code to understand the structure of the input data, and generated messages that were representative of real-world workloads. We also developed the privacy label functions accordingly.

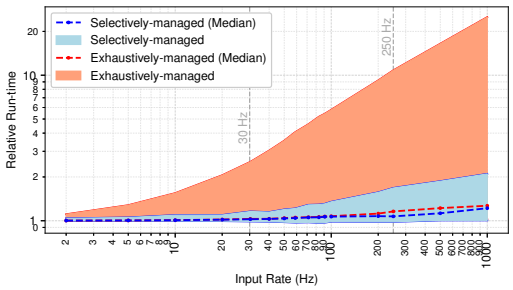

Figure 11. Relative run-time of applications observed over a range of input rates. The upper bound of the shaded area indicate the maximum run-time, and the dashed lines represent the median run-time.

For the IFC policy, we generated placeholder labels that do not carry any connotations related to the application. Thus, we could use the same set of labels consistently across all 27 target applications. In the real-world context, the placeholder labels can be replaced with the real-world labels. For example, the placeholder labels such as `Alpha` and `Beta` could be replaced with `employee` and `customer`, respectively.

Run-time Overhead. Figure 11 shows the relative run-times (y-axis) of the 27 applications observed over the range of input rates from 2 Hz to 1000 Hz (x-axis), in log-log scale. The blue shaded area covers the relative run-times for the selectively-managed applications, and the red shaded area covers those for the exhaustively-managed versions. The upper bound of each shaded area represents the maximum run-time observed (i.e., application with the worst overhead), and the lower bound represents the minimum. The dashed lines going across the shaded area represents the median.

As shown by the upper bound of the red shaded area, exhaustive tracking incurs overheads up to 2406.2% at 1000 Hz. In contrast, the maximum overhead of selectively-managed applications at 1000 Hz is 109.9%. *At 30 Hz, which is a typical input rate [42] for a camera based application such as our example (§ 3), exhaustive tracking incurs a maximum overhead of 153.8%, whereas selective tracking incurs only 15.8%.* An overhead lower than 20% (e.g., 200 ms for every 1-second worth of task) can be tolerated by a wide range of IoT applications [64, 70], barring applications with stringent timing constraints such as urgent healthcare. However, an overhead greater than 100% (2x run-time) would typically be intolerable, as the processing delay would be in the orders of seconds. Thus, exhaustive tracking fails to provide overheads low enough for one of the applications, even at the input rate of common video streams. Selective tracking, on the other hand, provides acceptable overheads for all 27 applications.

Besides the worst case, we observe that the median overhead grows from 0.2% at 2 Hz to 22.0% at 1000 Hz for the

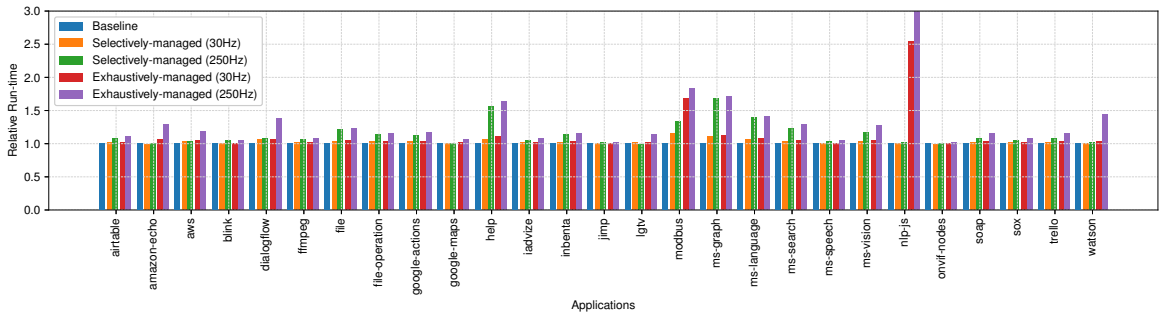

Figure 12. Relative run-times of the 27 applications, selectively-managed and exhaustively-managed, at 30 Hz and 250 Hz.

selectively-managed applications. The median overhead for exhaustively-managed applications is marginally higher, going from 0.3% at 2 Hz to 26.8% at 1000 Hz. At 30 Hz, the median overhead is only 2.2% for selectively-managed applications, and 2.7% for exhaustively-managed versions. For both versions, the overhead can be completely masked in practice by dropping a single non-critical frame (e.g., a frame that does not contain a face) every second. *At 250 Hz, which is the streaming rate of high frame rate cameras [42], the median overhead increases to 7.2% for selectively-managed applications, and 15.9% for exhaustively-managed applications.*

For both selective and exhaustive tracking, the majority of the applications had acceptable median overhead of less than 20%. This indicates that Turnstile's DIFT mechanism is generally efficient, regardless of the instrumentation strategy. *However, selective instrumentation increased the number of applications with acceptable median overhead from 16 to 22.*

Figure 12 shows the relative run-times observed for each application, at 30 Hz and 250 Hz. At 30 Hz, both instrumentation strategies provide low overheads (< 3%) across most applications, except for `modbus` and `nlp-js`. In these two applications, selective tracking has 52% and 153% less overhead than exhaustive tracking, incurring 15.8% and 0.4% overhead respectively. At 250 Hz, we observe three more applications for which selective tracking provides significantly less overhead than exhaustively-managed versions, namely, `amazon-echo` (28% less), `dialogflow` (30% less), and `watson` (42% less). In particular, `nlp-js` incurs an especially high overhead of 980.2% when using exhaustive tracking compared to only 2.5% with selective tracking.

`nlp-js` incurs such a high overhead due to Turnstile tracking all the large "dictionary" objects containing thousands of words and tokens used for natural language processing, which are not privacy-sensitive. Each string found in this dictionary is converted into a heap-allocated object, incurring significant overhead. Similarly in the other four applications for which the overhead due to exhaustive tracking was unacceptably high, exhaustive instrumentation tracks non privacy-sensitive objects – e.g., helper objects used for building HTTP requests, objects containing metadata and configuration parameters. Selective tracking eliminates such unnecessary tracking, significantly reducing the overhead.

Summary. (RQ2) We found that selective instrumentation can significantly reduce the overhead in five out of 27 applications compared to exhaustive instrumentation, bringing down the worst-case overhead from 153.8% to 15.8% at 30 Hz. This makes the selective tracking approach applicable across a wider range of applications than exhaustive tracking. (RQ3) The median run-time overhead ranged from 0.2% to 26.8% across the range of input rates in typical in IoT applications, regardless of the instrumentation strategy. This shows that Turnstile's DIFT mechanism is generally efficient.

7 Related Work

Security-typed Languages. Security-typed languages introduce the notion of *security types*, enabling IFC via type-checking over the security types. Jif[56], JFlow [55], Fabric [48], and JRIF [43] are some of the notable security-typed languages built on Java. FlowCaml [68] and LIO [73] are some other works, built on top of OCaml [46] and Haskell [33] respectively. SecVerilog [83] is a hardware design language that incorporates static IFC. Security-typed languages provide strong guarantees and fine-grained control over all objects in an application. However, they are *invasive*, as they require the adoption of the languages' specialized constructs.

Static Taint Analysis. There is a plethora of prior work on static taint analysis. We mention only a few relevant ones.

Pixy [39] and TAJ [75] are static taint analysis tools used to detect vulnerabilities such as *cross-site scripting* (XSS) in PHP and Java applications respectively. VEX [8], Mystique [13], DoubleX [23], TaintMini [77], JSTap [22], ODGen [47] are tools for detecting such vulnerable dataflows in JavaScript. Saint [11] and Soteria [12] identify vulnerabilities in IoT applications. Moore et al. [53] demonstrate a formal technique for selectively tracking privacy-sensitive objects. However, the technique was not evaluated on a real runtime system.

Static analysis is insufficient for managing the privacy of applications during run-time. Hence, Turnstile employs both static and dynamic IFC techniques inspired from prior work.

Dynamic Information Flow Tracking (DIFT). DIFT provides more precise control over the information flow, often at the cost of run-time overhead. All DIFT systems associate data with certain labels, track their propagation, and control the flow of labelled data at specific interface boundaries.

Resin [79], SafeWeb [32], DEFCon [51], Hails [29], and LWeb [59] are some of the *application-level DIFT* systems, which can provide platform independence. However, all of the above systems require significant changes to the application code, *violating the non-invasiveness constraint.*

JSFlow [31] and FlowFox [18] are *platform-level DIFT* systems integrated into specialized JavaScript runtime systems. Flume [44], HiStar [81], and DStar [82] are *platform-level DIFT* systems incorporated at the OS-level. Laminar [66] and TaintDroid [21] unify language-level IFC with OS-level IFC, by allowing labels to be tracked across the various software layers. Such platform-level DIFT systems require specialized platforms, *violating the platform-independence constraint.*

JEST [17] is an IFC monitor targeting browser-based applications. Similar to Turnstile, it prioritizes practicality and takes an inlining approach. The key difference is Turnstile's adoption of value-dependent labels, as opposed to static labels bound to data locations. This allows Turnstile to manage the privacy of data whose label depends dynamically on runtime content, rather than its source or code location. For instance, JEST associates a label with an image input element on an HTML form, whereas Turnstile associates a label dynamically to the uploaded image based on its content.

Provenance Systems. Provenance systems have several overlaps with IFC systems. Earlier systems such as Hi-Fi [65] and SPADE [28] focused primarily on logging data movement across system resources. Later systems such as DataTracker [72], ProTracer [50], and CamFlow [61] incorporated dynamic taint tracking to trace the flow of data on-the-fly. PB-DLP [9] and CamQuery [62] provide provenance-based run-time IFC. They are used to prevent sensitive data from leaving the device, which can be done by Turnstile. Unlike Turnstile, they control data flow based on the provenance, not based on the content of the data. NodeMedic [10] uses provenance graphs to identify vulnerabilities in Node.js packages, which is a dynamic approach to identifying privacy-sensitive code paths as opposed to Turnstile's static analysis.

Privacy Management in IoT. ContexIoT [38] employs code augmentation technique similar to that of Turnstile. However, ContexIoT's flow control is based on a mapping between distinct dataflow paths and explicit end-user permissions. ProvThings [78] is a provenance-based IFC system for IoT applications. PFirewall [16] is a centralized system for managing the dataflow of smart home applications based on user-defined policies. It does not track dataflows and controls flows based on the run-time information only.

FlowFence [24] provides sandboxing for handling privacy-sensitive data. It requires a rewrite of the application and relies on the developer to identify the sensitive parts of the application. Similarly, SandTrap [7] provides isolation between the user application and third-party code in trigger-action platforms (TAP), to prevent sensitive data from entering the third-party code inadvertently. However, it does not track data propagation beyond the interface between the application and the third-party component. MinTAP [14] provides a data minimization technique for TAPs, and does not provide flow control beyond the TAP interface.

Prior work on privacy management in IoT addresses concerns such as isolation [7, 24], data minimization [14], or access control [16, 38]. To the best of our knowledge, Turnstile is the first DIFT system for IoT applications.

8 Conclusion

We developed Turnstile, a hybrid IFC framework for managing privacy in distributed IoT applications. Unlike prior work, Turnstile requires changes to neither the runtime platform nor the application code. We evaluate Turnstile on 61 publicly available Node-RED IoT applications, in terms of its effectiveness and performance. We find that Turnstile's static dataflow analyzer identifies privacy-sensitive code paths within the order of milliseconds, making it a practical tool in a developer's arsenal. Further, Turnstile outperforms CodeQL both in terms of the number of code paths found and the time taken for the analysis. Finally, Turnstile provides a self-contained DIFT system that transparently controls the flow of private objects during run-time, while incurring only 2.2% median overhead when the input workload was streamed at 30 Hz, which is typical of IoT video streams.

There are three directions for future work. First, we want to provide stronger guarantees of noninterference by handling implicit flows and other dynamic control flows, which Turnstile currently does not handle. Second, we want to extend and generalize the same approach across different languages, as many real-world systems involve components written in languages other than JavaScript (e.g., Python). Finally, while we focused on privacy, we can use a different labelling framework to express more complex policies including integrity labels.

Acknowledgements

We thank our shepherd, Yinzhi Cao, for his guidance through the revision process. This work was supported in part by the Innovation for Defence Excellence and Security (IDEaS) program from the Department of National Defence of Canada, and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] 2018. California Consumer Privacy Act of 2018. *CAL. CIV, TITLE 1.81.5., Ch.55, Sec. 3* (2018). https://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.5

[2] 2024. AWS IoT Greengrass Documentation. https://docs.aws.amazon.com/greengrass/.

[3] 2024. Azure IoT Hub Documentation. https://learn.microsoft.com/en-ca/azure/iot-hub/.

[4] 2024. SmartApp SDK - Node.js. https://developer.smartthings.com/docs/sdks/smartapp-nodejs.

[5] 2025. Homebridge. https://homebridge.io/.

[6] 2025. openHAB. https://www.openhab.org/docs/.

[7] Mohammad M Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. 2021. {SandTrap}: Securing {JavaScript-driven}{Trigger-Action} Platforms. In *30th USENIX Security Symposium (USENIX Security 21)*. 2899–2916.

[8] Sruthi Bandhakavi, Nandit Tiku, Wyatt Pittman, Samuel T King, P Madhusudan, and Marianne Winslett. 2011. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM* 54, 9 (2011), 91–99.

[9] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy {Whole-System} provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*. 319–334.

[10] Darion Cassel, Wai Tuck Wong, and Limin Jia. 2023. Nodemedic: End-to-end analysis of node. js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1101–1127.

[11] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. 2018. Sensitive information tracking in commodity {IoT}. In *27th USENIX Security Symposium (USENIX Security 18)*. 1687–1704.

[12] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated {IoT} safety and security analysis. In *2018 USENIX annual technical conference (USENIX ATC 18)*. 147–158.

[13] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1687–1700.

[14] Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. 2022. Practical data access minimization in {Trigger-Action} platforms. In *31st USENIX Security Symposium (USENIX Security 22)*. 2929–2945.

[15] Winnie Cheng, Dan RK Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. 2012. Abstractions for usable information flow control in Aeolus. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 139–151.

[16] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. 2021. Pfirewall: Semantics-aware customizable data flow control for home automation systems. In *Network and Distributed Systems Symposium*.

[17] Andrey Chudnov and David A Naumann. 2015. Inlined information flow monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 629–643.

[18] 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*. 748–759.

[19] Dorothy E Denning and Peter J Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (1977), 504–513.

[20] Mohan Dhawan and Vinod Ganapathy. 2009. Analyzing information flow in JavaScript-based browser extensions. In *2009 Annual Computer Security Applications Conference*. IEEE, 382–391.

[21] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[22] Aurore Fass, Michael Backes, and Ben Stock. 2019. Jstap: a static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 257–269.

[23] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1804.

[24] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. {FlowFence}: Practical data protection for emerging {IoT} application frameworks. In *25th USENIX security symposium (USENIX Security 16)*. 531–548.

[25] IoT Security Foundation. 2019. Secure design best practice guides. https://iotsecurityfoundation.org/wp-content/uploads/2019/12/Best-Practice-Guides-Release-2_Digitalv3.pdf. [Accessed 24-06-2024].

[26] OpenJS Foundation. 2024. Express - Node.js web application framework. https://expressjs.com/. [Accessed 24-06-2024].

[27] OpenJS Foundation. 2024. Node-RED. https://nodered.org/. [Accessed 24-06-2024].

[28] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 101–120.

[29] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C Mitchell, and Alejandro Russo. 2012. Hails: Protecting data privacy in untrusted web applications. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 47–60.

[30] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[31] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.

[32] Petr Hosek, Matteo Migliavacca, Ioannis Papagiannis, David M Eyers, David Evans, Brian Shand, Jean Bacon, and Peter Pietzuch. 2011. SafeWeb: A Middleware for Securing Ruby-Based Web Applications. In *Middleware 2011: ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings 12*. Springer, 491–511.

[33] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 12–1.

[34] Github Inc. 2024. CodeQL. https://codeql.github.com/. [Accessed 24-06-2024].

[35] Ecma International. 2015. ECMAScript 2015 Language Specification. https://ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf.

[36] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*. 270–283.

[37] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*. Springer, 238–255.

[38] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. 2017. ContexIoT: Towards providing contextual integrity to appified IoT platforms.. In *ndss*, Vol. 2. San Diego, 2–2.

[39] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.

[40] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, VN Venkatakrishnan, and Yinzhi Cao. 2023. Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1059–1076.

[41] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. 2018. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1364–1379.

[42] Hamed Kiani Galoogahi, Ashton Fagg, Chen Huang, Deva Ramanan, and Simon Lucey. 2017. Need for speed: A benchmark for higher frame rate object tracking. In *Proceedings of the IEEE international conference on computer vision*. 1125–1134.

[43] Elisavet Kozyri, Owen Arden, Andrew C Myers, and Fred B Schneider. 2019. JRIF: reactive information flow control for java. *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows* (2019), 70–88.

[44] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 321–334.

[45] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. 2018. Alexa, are you listening? Privacy perceptions, concerns and privacy-seeking behaviors with smart speakers. *Proceedings of the ACM on human-computer interaction* 2, CSCW (2018), 1–31.

[46] Xavier Leroy, Jerome Vouillon, Damien Doligez, and Didier Remy. [n. d.]. Why OCaml? — ocaml.org. https://ocaml.org/about. [Accessed 24-06-2024].

[47] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*. 143–160.

[48] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. 2009. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 321–334.

[49] Joakim Lundin. 2024. node-red-contrib-deepstack. https://flows.nodered.org/node/node-red-contrib-deepstack. [Accessed 24-06-2024].

[50] Shiqing Ma, Xiangyu Zhang, Dongyan Xu, et al. 2016. Protracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting.. In *NDSS*, Vol. 2. 4.

[51] Matteo Migliavacca, Ioannis Papagiannis, David M Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. 2010. {DEFCON}:{High-Performance} Event Processing with Information Security. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[52] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 308–319.

[53] Scott Moore and Stephen Chong. 2011. Static analysis for efficient hybrid information-flow control. In *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE, 146–160.

[54] J Paul Morrison. 2010. *Flow-Based Programming: A new approach to application development*. CreateSpace.

[55] Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 228–241.

[56] Andrew C. Myers, N. Nystrom, L. Zheng, and Zdancewic S. [n. d.]. Jif: Java Information Flow. https://www.cs.cornell.edu/jif. [Accessed 24-06-2024].

[57] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.. In *NDSS*, Vol. 5. Citeseer, 3–4.

[58] Publications Office of the European Union. 2016. General data protection regulation (GDPR). *Official Journal (L 119) of the European Union* 59 (4 May 2016), 1–88. http://data.europa.eu/eli/reg/2016/679/oj

[59] James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information flow security for multi-tier web applications. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[60] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-System Provenance Capture. In *Symposium on Cloud Computing (SoCC'17)*. ACM.

[61] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*. 405–418.

[62] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime analysis of whole-system provenance. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 1601–1616.

[63] Thomas FJ-M Pasquier, Jean Bacon, and Brian Shand. 2014. FlowR: aspect oriented programming for information flow control in ruby. In *Proceedings of the 13th international conference on Modularity*. 37–48.

[64] Adrian Pekar, Jozef Mocnej, Winston KG Seah, and Iveta Zolotova. 2020. Application domain-based overview of IoT network traffic characteristics. *ACM Computing Surveys (CSUR)* 53, 4 (2020), 1–33.

[65] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 259–268.

[66] Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. 2009. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–74.

[67] Angela Sanguinetti, Beth Karlin, and Rebecca Ford. 2018. Understanding the path to smart home adoption: Segmenting and describing consumers across the innovation-decision process. *Energy research & social science* 46 (2018), 274–283.

[68] Vincent Simonet. 2003. *The Flow Caml System: documentation and user's manual*. Ph. D. Dissertation. INRIA.

[69] Jatinder Singh, Julia Powles, Thomas Pasquier, and Jean Bacon. 2015. Data flow management and compliance in cloud computing. *IEEE Cloud Computing* 2, 4 (2015), 24–32.

[70] Arunan Sivanathan, Daniel Sherratt, Hassan Habibi Gharakheili, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. 2017. Characterizing and classifying IoT traffic in smart cities and campuses. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 559–564.

[71] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schafer, Anders Moller, and Michael Pradel. 2020. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 198–209.

[72] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2015. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *Provenance and Annotation of Data and Processes: 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Revised Selected Papers 5*. Springer, 155–167.

[73] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*. 95–106.

[74] Council to Secure the Digital Economy. 2019. The C2 Consensus on IoT Device Security Baseline Capabilities. https://csde.org/wp-content/uploads/2019/09/CSDE_IoT-C2-Consensus-Report_FINAL.pdf. [Accessed 24-06-2024].

[75] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM*

Sigplan Notices 44, 6 (2009), 87–97.

[76] Csongor Varga. 2024. NVR: capture, store, email pictures/videos from IP camera streams. https://flows.nodered.org/flow/0da4d606575df13700461400178aca4e. [Accessed 24-06-2024].

[77] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 932–944.

[78] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and logging in the internet of things. In *Network and Distributed Systems Symposium*.

[79] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2009. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 291–304.

[80] Michael Zeifman and Kurt Roth. 2011. Nonintrusive appliance load monitoring: Review and outlook. *IEEE transactions on Consumer Electronics* 57, 1 (2011), 76–84.

[81] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres. 2011. Making information flow explicit in HiStar. *Commun. ACM* 54, 11 (2011), 93–101.

[82] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. 2008. Securing Distributed Systems with Information Flow Control.. In *NSDI*, Vol. 8. 293–308.

[83] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices* 50, 4 (2015), 503–516.

[84] Miao Zhang, Fangxin Wang, Yifei Zhu, Jiangchuan Liu, and Zhi Wang. 2021. Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines. In *Proceedings of the 12th ACM multimedia systems conference*. 80–93.

[85] Rui Zhao, Chuan Yue, and Qing Yi. 2015. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*. 1384–1394.

[86] Lantian Zheng and Andrew C Myers. 2007. Dynamic security labels and static information flow control. *International Journal of Information Security* 6 (2007), 67–84.

A Artifact Appendix

A.1 Abstract

The artifact package includes the source code of Turnstile, the CodeQL query we wrote for performing the taint analysis described in Section 6.1, the original data and third-party application code used to create the Figures 10, 11, and 12, the scripts used to process the raw data and produce the above figures, and a `Dockerfile` for building the container image containing all the dependencies for the execution environment.

A.2 Description & Requirements

A.2.1 How to access. The following Github repository contains all the components needed for recreating the experiments.

```
https://github.com/DependableSystemsLab
   /Turnstile-EuroSys26
```

It is archived at https://zenodo.org/records/17042223.

A.2.2 Hardware dependencies. At least 12 GB of disk space is required to store the Docker image and the output from the experiments. At least 8 GB of RAM is recommended, as we allocate 6 GB to the Node.js runtime for the experiment.

We conducted the experiments on a virtual machine with 4 vCPUs and 8 GB RAM, where the host machine was running a Intel Platinum 6548Y+ processor.

A.2.3 Software dependencies. To recreate the execution environment, Docker is required to run the provided Docker image.

A.2.4 Benchmarks. 61 third-party repositories used in Section 6 are required. These are already included in the provided Docker image.

A.3 Set-up

The recommended way to get started with running the artifact is to use the pre-built Docker image hosted at Docker Hub. Simply start a new container and enter the interactive shell with the following command:

```
docker run -it ––name turnstile-exp \
  jungkumseok/turnstile:eurosys26 /bin/bash
```

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): Turnstile is more effective than CodeQL at identifying the privacy-sensitive dataflows, finding a total of 190, compared to 52 found by CodeQL. This is illustrated by the experiment (E1) in Section 6.1 whose results are illustrated in Figure 10.*
- *(C2): Turnstile's selective instrumentation can significantly reduce the overhead in five out of 27 applications compared to exhaustive instrumentation, and Turnstile's DIFT mechanism is generally efficient as the median run-time overhead ranged from 0.2% to 26.8% across the range of input rates. This is demonstrated by the experiment (E2) in Section 6.2 whose reults are illustrated in Figures 11 and 12.*

A.4.2 Experiments.

Experiment (E1): Taint Analysis [1 human-minute + 3 compute-hour]: performs taint analysis over 61 third-party Node-RED applications from Section 6.1, using CodeQL and Turnstile.
In the interactive shell of the Docker container, navigate to the `/root/turnstile/scripts/analysis` directory. Run the Node.js script called `run-codeql-multiple.js` to run the CodeQL analyses.

```
node run-codeql-multiple.js REPOLIST.txt
```

This might take up to 3 hours [5]. Then, run the Node.js script called `run-turnstile-multiple.js` to run the Turnstile analyses, which takes about 4 minutes.

```
node run-turnstile-multiple.js REPOLIST.txt
```

The two scripts should generate two CSV files in the `/root/output` directory:

```
codeql-taint-analysis-result.DT.csv
turnstile-taint-analysis-result.DT.csv
```

`DT` is a timestamp indicating the time at which the CSV file was generated. Hereon, we assume that all output from the experiments are generated in the `/root/output` directory, and will not mention the output location explicitly.

Go to `/root/turnstile/scripts/presentation` and run `compile-analysis-results.js` to aggregate the two CSV files into a single CSV file.

```
node compile-analysis-results.js \
  turnstile-taint-analysis-result.DT.csv \
  codeql-taint-analysis-result.DT.csv
```

A CSV named `taint-analysis-compiled.DT.csv` will be generated, which can be used to generate Figure 10. Before creating the plot, the Python virtual environment must be active:

```
source .venv/bin/activate
```

Run the Python script `plot-line-dataflow.py` with the aggregated CSV file to generate Figure 10.

```
python plot-line-dataflow.py \
    taint-analysis-compiled.DT.csv
```

This will output a PNG file named `line-plot.DT.png`. This figure should be *exactly* the same as the Figure 10 in the paper.

Experiment (E2): Performance Overhead Measurement [1 human-minute + 24 compute-hour]: runs the 27 third-party Node-RED applications from Section 6.2 in a test environment, repeating the experiments across different input rates.

Navigate to `/root/turnstile/scripts/experiment` directory in the interactive shell.

Run the bash script called `run-all-experiments.sh` to run the experiments.

```
./run-all-experiments.sh
```

The script should take about 24 hours to complete. It will generate about 3.86 GB worth of raw data in one or more directories named `exp-DT`, with each file inside corresponding to a single run. If you wish to reduce the time by running only a subset of the experiments, please refer to Section A.6.

Once the script completes, navigate to `/root/turnstile/scripts/presentation` and run the `compile-experiment-results.js` script to consolidate all the raw data into a single JSON file.

```
node compile-experiment-results.js \
    exp-DT1 exp-DT2
```

This should generate a JSON file named `exp-results-compiled.DT.json`.

To generate Figure 11, run the `extract-area-data.js` script with the JSON file to generate a CSV file.

```
node extract-area-data.js \
    exp-results-compiled.DT.json
```

This generates a CSV file named `plot-area-data.DT.csv`. Run `plot-area-applications.py` with the CSV file to generate Figure 11.

```
python plot-area-applications.py \
    plot-area-data.DT.csv
```

To generate Figure 12, run the `extract-bar-data.js` script with the JSON file to generate another CSV file.

```
node extract-bar-data.js \
    exp-results-compiled.DT.json
```

This generates a CSV file named `plot-bar-data.DT.csv`. Run `plot-bar-applications.py` to generate Figure 12.

```
python plot-bar-applications.py \
    plot-bar-data.DT.csv
```

There will be two PNG files named `area-plot.X.png` and `bar-plot.X.png`, generated by the two Python scripts above. These two figures will not be exactly the same as Figures 11 and 12 in the paper, due to the platform differences. However, the overall trend should still be similar, and still support our claim C2. The median overhead shown in `area-plot.X.png` should be only slightly above 1, showing that the overhead is generally low across the applications.

[5] It took about 3 hours in our setup, but the timing depends on the hardware.

The maximum overhead shown in both figures should illustrate that the selective instrumentation reduces the overhead significantly compared to exhaustive instrumentation.

A.5 Notes on Reusability

The automation scripts internally execute smaller scripts, which can be used to perform small steps in the experiment. These "micro-scripts" can be used to test Turnstile's limits against applications not covered in the paper.

The `run-turnstile-single.js` script can be used to run Turnstile on an arbitrary code repository. It should print to the console the privacy-sensitive dataflows found by Turnstile, and an HTML page that can be used to visually inspect the dataflows.

Additionally, the artifact package contains the source code of Turnstile. The files `CodeAnalyzer.js` and `PrivacyTracker.js` can be extended to improve the coverage of Turnstile.

A.6 General Notes

Even though we included a reduced version of the experiment E2, it can take up to 24 hours, which might be too long for the purpose of trying out the artifact. If you wish to reduce the experiment time more significantly, open the `run-all-experiments.sh` file and comment out the following lines. These two lines correspond to the runs with input rate of 2 Hz, which is the most time consuming.

```
node run-experiment.js $exp V3-2fps false
node run-experiment.js $exp V3-2fps true
```

Received 14 May 2025; revised 15 Sept 2025; accepted 26 Sept 2025