Anonymous Author(s)

GuardRail: Fine-Grained Mobile App Policies for Accountable Private Data Use

Abstract: Mobile apps often raise privacy concerns because they have access to a wide array of raw personal data, and to make matters worse, sometimes leave users unaware of their reasons for collecting private information. Existing permission-based control mechanisms that mobile platforms offer do not provide insight into how the data is used and, more importantly, if the developers' claims about data use can be trusted. We present GuardRail, an approach for enabling greater accountability of apps' private data usage. App developers can use GuardRail's expressive policy language to provide a detailed behavioral description of their app's data practices, packaged with a binary version of their app. A trusted third-party, such as an app store, then applies GuardRail to rewrite the binary so that it self-enforces the developer's stated privacy policy when run by end-users. GuardRail uses a novel hybrid approach that mitigates the imprecision of static analysis by selectively inserting dynamic instrumentation to enforce the policy. We implemented a prototype for Android, and evaluated it on 125 real-world and benchmark apps. Our results show that GuardRail is capable of accurately enforcing a wide range of data-handling behaviors found in real apps; meanwhile over 90% apps have less than 3.8% worst-case expected runtime overhead, demonstrating the practical feasibility of our approach.

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

1 Introduction

Following increased calls from experts and end-users alike [20, 21, 39, 45, 76] for greater privacy and transparency into data-handling practices, legislative bodies have enacted regulations (e.g., GDPR [15], CCPA [62], PIPEDA [49]) that hold developers accountable for how they make use of private data. The growing focus on privacy has led to high-profile examples of government agencies imposing hefty fines on companies for questionable data-handling practices [52, 72], and concerns about whether applications follow through on the promises made in informal privacy policies.

Given the critical role that smartphones and mobile apps play in our daily lives, similar data privacy and transparency concerns are routinely raised in this context [37]. These concerns are particularly important for smartphones given that users carry them everywhere, and they have access to highly sensitive data like detailed location, user identity, photos [41]. The prevailing permission-based access control for sensitive data on smartphones does not provide sufficient transparency into how apps process users' private data — while they mention *what* data an app can access, they do not explain *how* the data will be processed, *why* the data is necessary, and *where* the app may send it.

Over the past decade, there have been numerous efforts to address the lack of transparency [24, 63, 71]. Much of the focus has been on requiring developers to provide human-readable privacy policies. For example, all major app stores require such policies before an app is made available to the public [28, 65]. However, previous studies have shown that these policies are often difficult for non-expert users to understand, and consumers rarely read them [63]. Furthermore, these privacy policies are not suitable for automated enforcement and verification, so there is no guarantee that the document's description matches the app's behavior.

In parallel, numerous efforts in the research community have sought ways of enabling more expressive machine-enforceable policies. For example, prior work proposes using static analysis of app code [3, 39, 51] or augmenting the OS itself [22, 64] to track information flows inside mobile apps. While these solutions are promising to detect unexpected data flows, they have drawbacks that limit their practical feasibility. Namely, the results of static analysis are usually imprecise, leading to false-positive alerts. In contrast, OS modifications require drastic changes to the underlying platform while imposing performance overheads across the system. Most importantly, the policies supported by these approaches are not expressive enough to capture real data-handling behaviors, which often include multiple information flows interleaved with one another, and involve events from the user and operating system.

We present GuardRail, a system that ensures that the app matches developers' claims about data usage. GuardRail enforces policies written in an expressive specification language using a novel hybrid instrumentation technique, mitigating the imprecision of static analysis while keeping dynamic runtime overhead to a minimum. In particular, GuardRail first performs static analysis of an app binary to identify information flows, which may involve multiple sources (e.g., access user location, or photos) and sinks (e.g., send over the network). Because the set of flows identified statically is likely to over-approximate the true behavior of the app at runtime, GuardRail selectively instruments the binary to enforce the policy conservatively. Note that while the instrumentation may reflect spurious over-approximate static flows, the additional information available at runtime allows GuardRail's enforcement to avoid flagging false-positive behaviors.

We envision GuardRail's rewriter being deployed by a third-party trusted by both developers and end-users, such as the Google Play Store. GuardRail assumes that App developers will provide a behavior specification along with their app binary for distribution. The GuardRail enforcer instruments the app according to the specification and publishes this modified version for users to download. GuardRail provides several benefits. First, privacy-conscious users can examine the behavior specification for increased transparency. Second, app developers can precisely describe their intended data-handling behaviors, specifically how it handles private data, without releasing their app's source code. Finally, developers can leverage GuardRail's instrumentation to test their apps for bugs that result in accidental data leakage, contrary to their stated policy.

To the best of our knowledge, GuardRail is the *first* end-to-end system combining an expressive behavior specification language to describe data handling with hybrid policy enforcement on Android applications. We evaluate GuardRail on 125 apps, including 40 real-world apps and 76 benchmark apps from a popular Android dataset. Furthermore, we develop multiple prototype apps based on 9 real-world data usage behaviors from popular open-source apps and decompiled closed-source apps. Finally, we evaluate the integration strategy for real-world apps using open-source apps and library SDKs. Our evaluations demonstrate GuardRail's capability of enforcing various real-world behaviors, improving precision over static analysis, and maintaining low overhead to be practical.

In summary, we make the following contributions:

- We propose an expressive specification language to describe the data usage behaviors of Android applications.[1] We use this language to describe 9 real-world data handling behaviors during evaluation.
- We design and implement GuardRail, a hybrid behavior-based policy enforcement system that rewrites application binaries to self-enforce specified behaviors. GuardRail requires no modifications to the underlying OS and leverages information from static analysis to selectively instrument parts of the app, thus minimizing runtime overhead.
- We evaluate GuardRail on 40 real-world applications, 76 benchmark apps, and prototype apps from 9 behaviors. Our results demonstrate that GuardRail can enforce diverse behaviors without requiring source code, while 90% apps experience less than 2% codebase increase.

Paper Outline. We provide a high-level overview of GuardRail in Section 2. In Section 3, we explain the GuardRail policy specification language with examples. Next, we describe the GuardRail policy-based instrumentation in Section 4, along with details on static and dynamic instrumentation components. Our evaluation in Section 5 demonstrates GuardRail's low performance overhead, expressive policy language, effective policy enforcement, and integration experiences with open-source apps and library SDKs.

2 Overview

The goal of GuardRail is to provide an expressive way for developers to publicly disclose details on how their program handles sensitive data. Privacy-concerned users can choose to install GuardRail-rewritten apps, which self-enforce application behaviors that adhere to the claims made by the app's developers. We envision GuardRail being deployed by an entity trusted by both developers and users, such as an app marketplace (e.g., Google Play store). Various app marketplaces already require developers to disclose user data handling practices through privacy policy documents [28, 65]. In GuardRail, developers need to specify their app behavior in a machine-readable policy language, providing details on private data usage, without requiring them to

1 In this paper, we use "behaviors", "signatures", "policies", and "specifications" interchangeably.

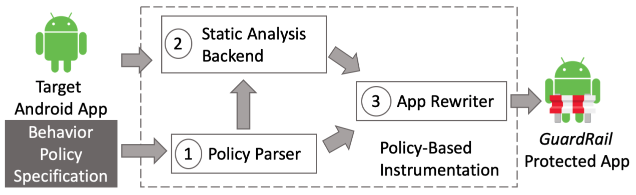

Target Android App
Behavior Policy Specification
2 Static Analysis Backend
1 Policy Parser
3 App Rewriter
Policy-Based Instrumentation
GuardRail Protected App

Fig. 1. GuardRail Architecture. Developers provide application binary as APK file as well as a behavior policy specification as inputs to GuardRail. Our system first processes the provided policy in Step 1, conducts static analysis on the APK file in Step 2, and then rewrites the application based on policy and flow results from static analysis in Step 3. The final output of GuardRail is the instrumented version of application APK file, which includes instructions to enforce the specified behaviors at runtime.

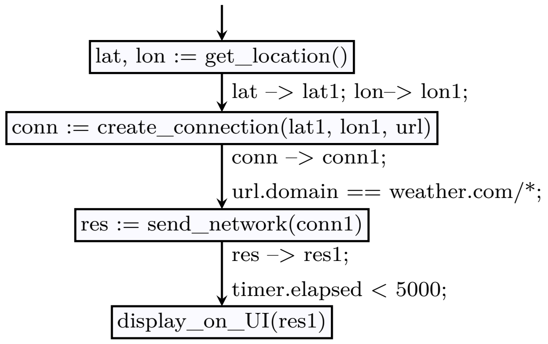

lat, lon := get_location()
lat –> lat1; lon–> lon1;
conn := create_connection(lat1, lon1, url)
conn –> conn1;
url.domain == weather.com/*;
res := send_network(conn1)
res –> res1;
timer.elapsed < 5000;
display_on_UI(res1)

Fig. 2. Behavior signature example for using location data to provide weather information customization. The constraint on `url.domain` specifies that only certain domains are allowed to receive user location data. Section 3 provides detailed explanation of behavior signatures and nodes and edges in this figure.

release source code. The app marketplace, serving as the GuardRail rewriter, rewrites the app based on the provided behavior specification and distributes the new app for users to download.

GuardRail provides key benefits to users and developers. First, users who seek transparency into how apps use their data can examine the policy specification and install the GuardRail-rewritten versions. Second, since the GuardRail rewriter conducts the instrumentation, users can trust the rewritten apps will only process private data as disclosed in the specification. Furthermore, we assume that the enforcer has significant computing resources to perform GuardRail's analysis and rewriting consistently at the scale necessary to support modern app ecosystems. Third, by providing desired app behavior specifications, developers explicitly state their honest intentions to win user trust. In case careless bugs or mistakes cause the app to accidentally leak private data, as is not uncommon as witnessed by our example below, developers can rest assured that rewritten apps will block the undisclosed behaviors. Notably, developers can use GuardRail themselves and incorporate it as part of their development process to help ensure the rewritten app behaves as intended.

GuardRail enforces a policy by selectively inserting instructions to track any event related to sensitive data handlings, such as executing platform routines to read and transmit sensitive data, and the flow of information within the target app. This selective instrumentation ensures that if the app deviates from its specified behavior policy when handling sensitive data, the modified app can take actions such as safely ignoring or raising exceptions, preempting potential violations.

Architecture. We present the overall architecture of GuardRail in Figure 1. In step ①, GuardRail processes and analyzes the provided behavioral specification. For the spec in Figure 2, GuardRail separates the signature into three stages defined by information flow constraints: *(1)* sending location data to the network; *(2)* retrieving the network response from this request, and *(3)* displaying the result.

GuardRail uses the information at each stage to guide a sequence of static analysis queries and rewriting steps. The static analysis passes (step ②) take a set of data sources and destinations and retrieve a conservative set of paths through the program that may induce information flows between the source and sink pairs. After step ②, GuardRail could have stopped with a conservative determination on whether the input app complies with the policy. However, because static information flow analysis is imprecise, it will likely return a large number of spurious flows that do not correspond to an actual execution path represented by the flow. Moreover, performing Value Set Analysis [8] to resolve constraints like the one on the *url* argument would introduce even more imprecision, inevitably leading to false-positive results on most non-trivial apps.

To address these challenges, GuardRail employs a hybrid approach that uses dynamic information to precisely evaluate edge constraints at runtime. In step ③, GuardRail takes the results of step ② and rewrites the app to track dynamic information and policy states, ensuring that the app's behavior complies with the specification. In particular, GuardRail inserts instructions to dynamically track information flow constraints, and accesses program state to evaluate value predicates immediately before executing the corresponding event. Finally, GuardRail tracks the apps' transitions through the policy state by monitoring node events using the path information provided by the static analysis.

The result of this process is a transformed application that self-enforces the data usage behaviors described in the specification. For an app with behavior in Figure 2, the GuardRail augmented version will ensure that location is only used in communications with a trusted domain, and that the results of these communications are only displayed on the user's device. In the event of a bug like the unintended location leakage, the unspecified program behavior will violate the self-enforced policy and raise an exception rather than leaking user data.

Example. To see how this process works in more detail, consider the *Transparent clock & weather* (*TCW*) app [66], a popular weather app on the Google Play store with over 50 million downloads. It is justifiable for users to give location permission for its primary function as a weather app. However, an independent third-party audit discovered an alarming behavior in TCW, apart from its main functionality [13]. TCW was transmitting user location data every 15 seconds, without rendering the network response from the server. The developers recently responded to these claims, attributing the seemingly problematic behavior to a bug, stating that this behavior was unintentional [13].

By asking developers to provide machine-readable data-handling policies, GuardRail is well-suited to prevent such undisclosed data leakages. Suppose that the TCW developers provided a policy describing a "location for weather customization" behavior. This behavior encodes the process of reading location data, performing a one-time search for local weather information, and then displaying the search results on the App's UI.

Figure 2 shows this specification in GuardRail's graphical illustration, which is described further in Section 3. Each node in the graph describes a policy state. GuardRail's runtime monitoring transitions to a new policy state when the program executes the event denoted by the node, with arguments that satisfy each of the predicates on the incoming edges. In Figure 2, for example, the predicate $lat \rightarrow lat1$ denotes that there must be an information flow dependency between *lat* and *lat1*. In addition, $url.domain = \texttt{weather.com/*}$ gives a constraint on the value that the *url* argument to `create_connection`. Finally, the $\texttt{timer.elapsed} < 5000$ predicate denotes that the network operation's result must be displayed to the user within 5 seconds.

This behavior signature will flag the problematic behavior for this app described earlier. The unintended bug causes the TCW app to constantly upload the users location without waiting for a response or displaying it on the UI, violating the bottom half of the state diagram. With GuardRail enforcement, the buggy behavior will be exposed during execution, because location is leaked on execution paths that do not match any sequence of transitions represented by the provided behavioral signature. Once a violation is detected, the GuardRail-modified TCW app can automatically terminate, or disable specific sensitive data execution paths at runtime, to prevent these behaviors from executing.

Threat Model. GuardRail requires app developers to provide specifications describing their app's use of sensitive data. It is however feasible that a developer may inadvertently not specify some behaviors, or not include the behavior of a third-party library they use. In these cases, GuardRail will block undisclosed behaviors during enforcement. As mentioned earlier, GuardRail can be used during the development process to detect these cases and remove the unintended behavior or update their behavioral specification. FlowDroid, the open-source static analysis engine we build our GuardRail prototype upon, has some inherent limitations such as being unable to handle reflection, native code, or inter-app communications. We will discuss these limitations and alternative solutions in Section 6. Finally, we assume that the entity applying GuardRail to rewrite an app has the app's signing key. For example, many developers use Google Play's signing service [29], and hence the marketplace can rewrite the app and sign it before publishing.

3 Specifying Data Usage Behaviors

In this section, we present the specification language for describing stateful behaviors with multi-stage information flow constraints (Section 3.1) in GuardRail. We then use an illustrative example to demonstrate how to use this language to capture a rich set of behaviors in mobile apps (Section 3.2).

3.1 Behavior Graphs

We adapt and extend the concept of *behavior graphs* proposed in prior work [42, 46]. A behavior graph consists of a set of application events and expresses relationships across different events. In contrast to how behavior graphs have been used for malware detection [42], a GuardRail specification requires that once an app has triggered an event that returns sensitive information, all flows of that information must follow transitions specified in the policy, until it reaches a node with no outgoing transitions. If it fails to do so, it is deemed to violate the policy, and we find an execution trace inconsistent with the provided specification.

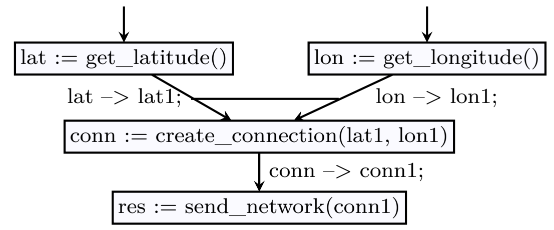
Definition 1. *Behavior Graphs:* A behavior graph is a directed graph $G = (V, E, \alpha, \beta)$ over a set of event space Σ, where:

– the event space Σ includes the set of application events,
– the vertices V correspond to events in Σ,
– the edges $E \subseteq V \times V$ represents temporal and data dependencies between different events,
– the labeling function $\alpha : E \to \mathbb{P}$ associates edges with edge predicates,
– the labeling function $\beta : E \times E \to \{AND, OR\}$ associates edge pairs with edge relationships.

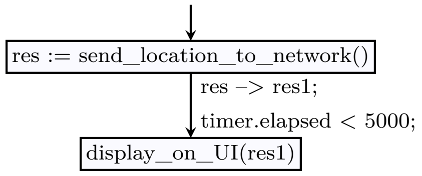
Events. Events in Σ correspond either to Android APIs, or to derived compositional events that refer to other behavior graphs. We construct a lookup table to represent the mapping relationship between events to low-level APIs. For example, a `get_location` event can translate to API calls to `get_latitude()`, `get_longtitude()` or `getLastKnownLocation()`.

The compositional events are program behaviors expressed in separate behavior graphs. By using a compositional and inductive definition, we can abstract common program behaviors in smaller components and reuse definitions. For example, a `network_send` event corresponds to a graph that represents all the ways to send data over the network with Android APIs and common libraries. Then for other events like sending location to network, they can use `network_send` as one event node in their graph construction.

Edge Predicates. The set of edge predicates $\mathbb{P}$ represents constraints and data dependencies across different node events. We currently support Boolean Java expressions and a taint propagation predicate ($\to$) that denotes an information flow dependency between events. Boolean Java expressions can be used to express data constraints between events such as the set of network domains that are allowed to receive location data as in Figure 2, and more generally, any expression over native types that may appear in a conditional. In addition, GuardRail provides a timing constraint, `timer.elapsed < t`, which ensures that the policy transitions through an edge within t milliseconds; if it does not do so, then the execution is regarded as violating the specification. The timing constraint is useful when a partial match



(a) Program behavior of sending location information to the network. The special edge relationship between (`get_latitude`, `create_connection`) and (`get_longitude`, `create_connection`) is a pair of AND edges.



(b) Program behavior for sending location to the network, and then display the response on the UI element. This is a common behavior used in apps like using location data to provide penalization or customization contents. The event `send_location_to_network` is a compositional event generalized by the previous behavior graph.

Fig. 3. We present 2 behavior signature examples. The edge predicate a –> b denotes the taint propagation predicate from variable a to b.

constitutes a policy violation, as in the case of rendering network responses from the example in Section 2.

Edge Relations. For multiple incoming edges into the same node, we use AND/OR edge relations and a labeling function $\beta : E \times E \to \{AND, OR\}$ to represent their relations. Conjunctive AND edges constrain the behavior to match if and only if all incoming edges happen during execution, while disjunctive OR edges denote matching as long as any path is executed. For notational simplicity, unspecified edge relations are assumed to be OR relations.

3.2 Examples

Figure 3 shows two visual representations of the behaviors sending location to the network and displaying information on the UI. Both of them can be represented with our specification language formally.

Figure 3a describes the behavior of an app retrieving user location information and then sending it over the network. Taint propagation predicates are denoted as a–>b. For each predicate, we use different variable names for taint sources and sinks, because they may be in different methods or classes. The relationship between edge (`get_latitude`, `create_connection`) and edge (`get_longitude`, `create_connection`) is an example of AND edge pairs. In order for behavior to match, the `create_connection()` call must contain both latitude and longitude information.

Figure 3b shows compositional behavior signatures. It encodes the behavior of getting location information, retrieving customization data based on user location, and *eventually* displaying results on UI elements. Since the previous example already expresses the first two steps, this signature simply includes a compositional event to reuse definitions. The concept of eventuality is expressed with `timer.elapsed` constraint, ensuring the next event must happen within the timer limit.

4 Policy-Based Instrumentation

In this section, we describe how GuardRail rewrites applications to enforce behavior signatures. The goal of the instrumentation component in GuardRail is to take a source application APK and accompanying behavior signature as inputs and to generate a modified application APK that self-enforces the behavior specified in the signature. The app can provide multiple behavior signatures all at once. More specifically, we have the following requirements for GuardRail as it pertains to our policy-based instrumentation:

Compatibility with existing platforms. The transformed applications generated by GuardRail should work well when running on stock Android, and not require modifications to the default OS or runtime, to enable a wide majority of users to benefit from it.

Sound dynamic enforcement. The instrumentation process should be conservative in that it should not "miss" any traces that violate the data usage specification. However, in keeping with our threat model in Section 2, we do not consider program obfuscation techniques that attempt to transmit sensitive information through side channels, for example. In addition, as we will discuss in Section 6, GuardRail will inherit limitations from the chosen static analysis backend, and so we do not insist on soundness beyond what is provided by the static analysis.

Minimal Impact on Correct Behavior. Instrumentation should not affect the behavior of apps that satisfy the policy. Benign programming behaviors and explicitly-disclosed behaviors following the policy should execute normally. Correct instrumentation should ensure behaviors that adhere to the policy get executed without interruptions. Meanwhile, GuardRail instrumentation should impose minimal runtime performance overhead, even in the worst-case scenarios, for it to be practical and useful.

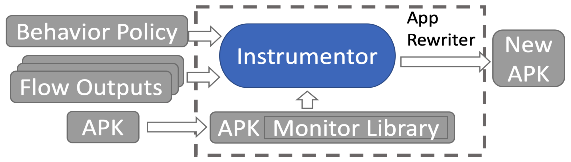


Fig. 4. System architecture of App Rewriter. The target APK file is first patched with runtime monitoring library and then instrumented based on policy and reported sensitive information flows.

No source code required. GuardRail relies on a trusted third party to perform app rewriting. Developers will legitimately be concerned about sharing their source code with a third-party, even if it is for security or privacy. To address this, we expect developers to only provide application binaries as Android Package (APK) files, as they do currently, to publish their apps in the Google Play Store.

4.1 Instrumentation Workflow

We provided an overall view of our approach in Figure 1. A key aspect of our workflow is the App Rewriter module, which selectively instruments the source app to create a target APK. As shown in Figure 4 the rewriter module first patches the APK with our runtime monitoring library and then instruments the code based on flows provided by the static analysis pass. For the rest of this section, we describe the challenges we faced in designing and implementing these modules. The static analysis backend and its driver are explained Section 4.2. Then we explain the challenges we faced for runtime monitoring and program instrumentation in Section 4.3. Finally, we briefly discuss the implementation in Section 4.4.

4.2 Static Analysis Backend

For the purposes of our instrumentation, the static analysis backend needs to provide information regarding: *(1)* the initial source and ultimate destination within the application of any sensitive data accesses mentioned in the behavior spec; *(2)* a complete listing of the instructions that may be involved in the transmission from source to destination.

We need the static analysis to be conservative to not miss any true positive flows in its report. To collect as many possible flows as possible, we configure options for static analysis to over-approximate flow results. Over-approximation is a double-edged sword because it can come in either spurious source-destination flow pairs or additional instructions on the transmission path for a flow. These results will not affect the soundness of enforcement, and we leverage our dynamic flow tracking component to recover precision by correctly propagating flow information at runtime.

A drawback of being conservative is an increase in the number of potential flows to analyze. Static analysis often has trouble scaling to a large number of sources and sinks. To address this scalability bottleneck, GuardRail takes a compositional approach where we perform multiple rounds of static analysis, each with a small subset of source and sinks. In the end, we can aggregate all information flows collected in previous runs to the next step of GuardRail instrumentation. Similarly, the provided behavior signature can include multiple edges across different vertices. For each taint propagation edge in the behavior graph, GuardRail performs static analysis with the given source and sink pairs.

There are many existing static analysis frameworks for binary Android applications [3, 6, 30, 69] and Java code [31], with different tradeoffs in terms of precision, runtime, memory cost, and stability. GuardRail is designed to be modular with the possibility of switching between different tools in the future, as long as they provide compatible flow information.

4.3 Runtime State & Dynamic Tracking

The most challenging aspect of the GuardRail app rewriting process is to ensure a precise and efficient dynamic information flow tracking by selectively inserting instructions. We describe several technical challenges to accomplishing this and highlight our solutions.

Runtime Monitoring Library. Given the goal of executing modified APKs on stock Android devices, the modified APKs need to be self-contained with all necessary functionality to self-enforce a given policy. To achieve this, we create a standalone runtime monitoring library, which is patched directly to the target APK as the first step in the app rewriting process. This component manages the taint tag storage for program variables at runtime and provides API interfaces for dynamic taint propagation. The rewriter only needs to insert instructions to invoke the propagation methods along the information flow paths reported by the static analysis backend in the app rewriting step. At runtime, the monitoring library also manages the policy state and ensures that the application proceeds along the edges specified in the behavior policy, in a manner similar to inlined reference-monitoring of security automata [58].

Spurious Paths. Since the static analysis phase reports over-approximated flow paths, there are many overlapping and spurious flows in the result. A simple solution is to instrument each path independently. This is easy to achieve programmatically, and all true positive paths would be preserved. However, this instrumentation is extremely expensive since it would lead to duplicate tracking code on the same path elements, as they are reported in different flow paths.

Given our goal of low runtime overhead of GuardRail modified apps, we needed a more efficient instrumentation approach. In GuardRail, the app rewriter pre-processes all the reported flows and construct a graph representation of all instructions across different taint propagation paths. Subsequently, in the instrumentation process, our app rewriter looks up the target instruction for the corresponding node in the graph. We insert taint retrieval commands to merge incoming taints from all of the instruction's predecessors and propagation commands to all of the successors. The end result is that during program execution, taints coming from spurious paths would be empty, but the taint propagating through the correct path will be retained.

Function Calls. To track taint across function calls, the caller needs to register the tainted argument prior to the function invocation. Inside the callee function, the app rewriter inserts code to retrieve taint tags associated with function arguments. To differentiate different call sites onto the same callee function, the app rewriter associates additional call stack traces with each tainted argument as well.

The call stack trace method does not work with multi-threaded programs and asynchronous methods. For example, invoking `AsyncTask.execute()` method on caller thread will trigger the corresponding `doInBackground()` to be executed on a new forked thread. From the callee function, the call stack trace contains no information about the caller stack, only starting from OS system forks. To consistently track taints across AsyncTasks, we leverage object-level sensitivity to find taints related to a particular callsite, a technique that has been used in prior work [68]. When the caller register tainted calls, the rewriter associates the taint with the specific AsyncTask object memory reference to monitor the library's storage. Inside the callee function, the rewriter inserts a taint retrieval method using the callee object's `this` pointer. The runtime monitoring library looks up the corre-

```
// foo.jimple
void foo() {
  $r1 := com.storage;  // S1
  $d0 = getLatitude();  // S2
  $r2 = $r1.<com.storage double: userLat>; // S3
  $r2 = $d0;   // S4
}
```

Fig. 5. Example of an information flow involving object fields. The static analysis reports a flow path as $S2 \to S4 \to S3 \to S1$.

sponding AsyncTask object from the taint call registry and propagates taints to the callsite accordingly.

Object Sensitivity. Another case in which our app rewriter requires object sensitivity is for tracking taints from object fields or static variables. One example is shown in Figure 5. The static analysis tool reports an information flow path of $S2 \to S4 \to S3 \to S1$. Based on this path, the app rewriter will propagate taint from `foo:$d0` to `foo:$r2` and eventually to `$r1.<userLat>` field value. If GuardRail only associate taints with canonical local variable with class names, it is insufficient to track instance field usage in other functions correctly. Instead, GuardRail stores the corresponding object reference associated with the tainted fields. To dynamically resolve object reference, the app rewriter needs to insert instructions to de-reference base objects and construct storage key for taint tags at runtime.

Instruction Placement. Deducing the correct location to insert the instrumentation code is also an interesting design challenge. One option is to add taint propagation code before each instruction directly. During program execution, the monitoring and taint propagation code will execute before the original instruction. However, this placement strategy interferes with the correctness of object sensitivity. When the target instruction is an object initialization or a class constructor call, executing propagation code beforehand leads to de-referencing non-existing objects.

To fix the null pointer exception, we could apply a quick fix by inserting code *after* the instruction. However, we have to make several exceptions to ensure correctness. Behavior checking and taint propagation must happen before taint sinks because we need to interrupt the execution in case of behavior violation. We also need to insert code *before* return statements and function invocations. However, special cases (such as object initialization *function call*) interferes with this simple strategy. As a result, we develop a complex ruleset to determine the best placement and instrumentation strategies for each special case. Our evaluation on the system's correctness (Section 5.4.1) shows our ruleset has comprehensive coverage.

4.4 Implementation

We implemented GuardRail's core Policy-Based Instrumentation module in 2746 lines of Java code. Our standalone runtime monitoring library contains 777 lines of Java code. We utilize several existing Android and Java analysis tools. We adapt FlowDroid [3] for static taint analysis since it is the most popular open-source option for Android applications. We configure our static analysis backend to execute with a flow-insensitive dataflow solver considering all possible instruction combinations, a sound call graph algorithm (Class Hierarchy Analysis [18]) covering all control flow paths, a precise path reconstruction algorithm, and options to report all paths so that we can collect as many taint propagation path as possible. We choose these options based on the design decision explained by the author of the FlowDroid tool [2]. We use Soot [67] framework to implement the app rewriter module so that we can instrument application DEX code in Jimple intermediate representation format. We use Apktool [1] to patch runtime monitoring library to target APK binary, and `jadx` [34] to decompile app binary for manual inspection. We encode the behavior signature policy in JSON.

5 Evaluation

We evaluated our prototype implementation of GuardRail on a set of real and prototype apps to gain insight into the following questions:

- How expensive is GuardRail's rewriting step, and what is the performance overhead due to GuardRail's instrumentation? In particular, does GuardRail's analysis scale to real apps, and if so, does its instrumentation result in reasonable runtime overhead?
- How does GuardRail's hybrid policy enforcement compare against purely static or dynamic alternatives? In particular, how many spurious flows recovered by static analysis does it remove to avoid false positives, and how does its runtime overhead compare to exhaustive dynamic taint analysis [22]?

– Can the policy specification language capture a diverse set of real-world data-handing behaviors? Can GuardRail enforce them all with minimal overhead?
– How well does GuardRail integrate with developing an application or library to enhance the developer's confidence in the resulting app's data-handling behaviors?

Our evaluation on 40 real-world applications successfully reports behavior matching in 6 apps while filtering out 16 apps false-positively reported by static analysis. Moreover, an extensive analysis of these real apps and 76 benchmark apps demonstrates that GuardRail incurs less than 2% codebase increase and 3.8% expected runtime overhead under the worst case. We also evaluate GuardRail with multiple prototype apps developed from 9 real-world data-handling behaviors and case studies on integration with open-source apps and libraries. These results show that the system is capable of enforcing a wide variety of behaviors in a real-world setting.

In Section 5.1 we discuss the app datasets used for our evaluation. Section 5.2 evaluates GuardRail performance overhead. Section 5.3 compares GuardRail with other static and dynamic approaches. Section 5.4 validates GuardRail's effectiveness against various data handling practices. Section 5.5 discusses our integration strategy for a real app and a third party library.

5.1 Setup and Data Sources

We execute GuardRail on a virtual machine with 8 CPU cores and 45GB RAM. The host machine has dual Xeon CPUs (28 cores) and 256 GB RAM, enabling us to use multiple VMs for running experiments in parallel. We limit FlowDroid to 40 GB of RAM, and install apps on Nexus 5x phones (Android 7.0) when applicable.

We configure FlowDroid with conservative options explained in Section 4.4. We set timeout limits for FlowDroid analysis to be 20 minutes for the data flow solver, and 10 minutes for path reconstruction. Empirically, the majority of apps for which FlowDroid fails to terminate within these limits typically take from several hours to days to complete. We believe that if GuardRail should be deployed by an App store operator, such as Google, they will have enough computation resources to run the analysis for more extended periods.

Benchmark and Prototype Apps. We select 76 apps from the DroidBench [19] test suite, covering several programming patterns. We exclude DroidBench apps using implicit flows, inter-component communications, and emulator circumvention from our selection. We also exclude true-negative apps in DroidBench, since FlowDroid doesn't report any flows. In addition, we develop 9 prototype apps to evaluate different real-world data-handling practices in Section 5.4.

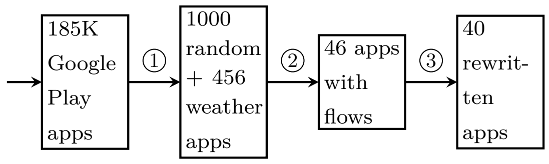


Fig. 6. The selection process for real-world apps. We begin with 185K apps from the Google Play store. In Step 1, we randomly select 1000 apps from this dataset, and add all 456 apps in the "Weather" category to improve the probability for apps to access private location data. Step 2 performs static analysis on these apps, resulting in 46 apps terminating successfully with positive flows within our 20min timeout. As for the rest, 393 apps trigger timeout or failures with FlowDroid, while FlowDroid doesn't report any flows for the remaining 1017 apps. In Step 3, we use GuardRail to instrument 40 apps, omitting 6 apps triggering bugs in Apktool, one of our dependencies.

Real App Dataset. To experiment with real-world Android apps, we start with a dataset of 185K apps, shown in Figure 6. We randomly select 1000 apps from this dataset, and add all 456 apps from the "Weather" category, to create a collection of 1456 real-world apps. We added the weather apps since they have a higher probability of accessing private information, such as using location data for personalization. Location tracking in mobile apps is regarded as highly sensitive to users [14, 25], and it is often accessed by many real apps [11]. The authors of Mobipurpose shared their app dataset and network traces with us. We use these traces to filter certain apps in Section 5.4.

5.2 Performance

We evaluate the cost GuardRail's rewriting for policy enforcement, and the runtime overhead of the policy instrumentation for 40 real apps and 28 adapted DroidBench apps. As Step ② and ③ in Figure 6 show, FlowDroid successfully terminate and returns flows for 46 Apps. Of these, 6 apps trigger a bug in Apktool, one of GuardRail's dependent tools, for patching Monitor Service smali code into APK. Thus, GuardRail successfully instruments 40 apps. Note that this is not an inherent limitation of our approach, but rather a result of FlowDroid's low termination rate, which we will discuss in Section 6 and well-documented in prior works [51].

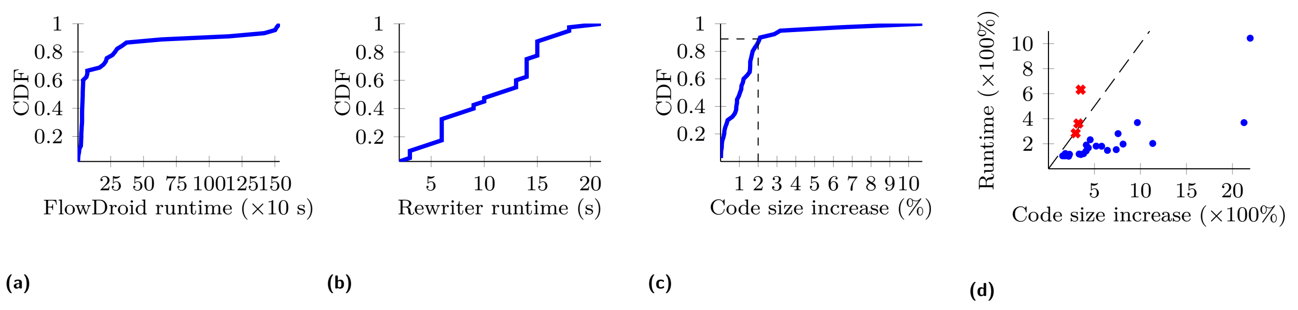


Fig. 7. Performance evaluation results. (a) shows FlowDroid analysis time on 46 real apps. We explain the selection process in Section 5.2. Plots (b) and (c) show rewriting time and code size increase on 40 real apps. 6 apps from FlowDroid analysis fail at rewriting due to bugs in external tools. Plot (d) shows the correlation between runtime overhead and relative code increase in execution paths, measured in 28 DroidBench apps. Dash line is $y = x$. Red crosses are cases where sensitive information propagate in loops, showing the limitation of using instruction counts in codebase as a predictor of runtime overhead.

Instrumentation Time and Code Size. Of the two tasks in our instrumentation workflow, the static analysis takes the most time (Figure 7a) as compared to app rewriting, which takes < 30s for all our test apps (Figure 7b). We also measure the relative code size increase for the 40 apps GuardRail successfully rewrites (Figure 7c), in terms of the additional Jimple instructions.90% of the real apps exhibit a codebase increase of <2%, showing a low overhead of GuardRail.

Runtime Overhead. Because Android apps are typically event-driven and GuardRail only selectively instruments part of the programs, common Java performance benchmarks (e.g., CaffeineMark [16] are not well-suited for overhead measurement. The amount of instrumentation added by GuardRail depends on the number of instructions in policy-relevant flow paths from static analysis, as well as the policy's overall complexity. Since standard benchmarks are unlikely to contain many instructions relevant to real-world policies, the overhead would not be representative of real apps.

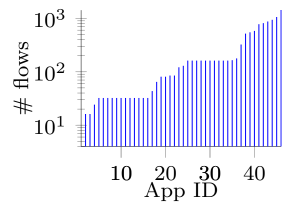
Instead, we examined 28 DroidBench apps, whose execution time we can measure reliably. We skip DroidBench tests propagating data asynchronously due to the difficulty in measuring runtime. For each app, we execute the code paths transmitting sensitive data 100 times, measuring the execution time and the count of instructions executed before and after GuardRail instrumentation. Figure 7d shows a sublinear relationship between codebase increase and runtime overhead, implying that the runtime overhead is often smaller than the relative codebase increase. This experiment has a large codebase increase ratio because DroidBench tests have short data transmission paths. For a simple program with dozens of instructions, GuardRail instrumentation can inflate the codebase multiple times due to the added propagation instructions. Three apps use loops to encode and obfuscate private information (red X's). These data points show that codebase increase is not a perfect indicator of runtime overhead.

Non-determinism in real-app execution makes it challenging to measure and accurately predict GuardRail runtime overhead. Nevertheless, we are optimistic about GuardRail's performance impact for the following reasons. First, the overhead only affects program paths that potentially send sensitive data. Most of the app's functionality is unaffected. Second, the relative overhead depends heavily on the length of the execution path and the portion of code instrumented. For DroidBench apps, the execution paths are concise, and most of the program require instrumentation, which is very unlikely in real apps. With longer execution paths of real apps, the overhead will be less significant. Third, real apps use slow system APIs and wait for UI events, which reduces the perceived performance overhead [22].

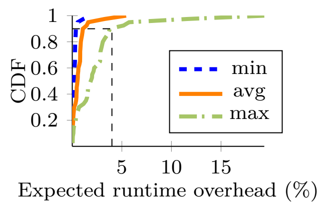
5.3 Comparing Enforcement Methods

We conducted a large-scale evaluation of GuardRail on 40 real apps to show the difference in enforcement when using GuardRail's hybrid approach, as compared to purely-static or dynamic approaches. In particular, we compare against static methods that rely only on a dataflow analysis tool like FlowDroid to detect information flows and dynamic approaches that depend on propagating taint globally at every program instruction.

Static taint analysis is known to be conservative and thereby return many false positives [38]. Thus, to compare with a static-only approach, we measure the reduction in false alarms when enforcing a simple policy that tracks the data from location APIs to the network.



(a)



(b)

Fig. 8. Comparison to static and dynamic approaches on real apps. (a) shows the number of flows reported by FlowDroid. 16 apps have up to 32 flows, while the max flow is 1424. (b) shows the expected runtime overhead for real apps. Under worst-case, 90% apps would expect less than 3.8% runtime overhead.

On the other hand, a purely-dynamic approach (e.g., TaintDroid [22]) will not suffer from false positives, but instead introduces excessive runtime overhead because the instrumentation is applied indiscriminately over the entire program. In contrast, GuardRail only introduces instrumentation as-needed for a specific policy.

Our results in Figure 8 show that GuardRail performs enforcement on real apps while filtering out many spurious, false-positive flows reported by the dataflow analysis. Additionally, GuardRail's selective instrumentation leads to an insignificant codebase increase in real apps, avoiding the full program runtime overhead of dynamic taint tracking solutions. In the rest of this subsection, we provide more details on the experiments and results leading to these conclusions.

Methodology. We evaluate a set of real-world apps with a common single behavior of sending location data to the network. An alternative would be to manually construct highly precise, detailed signatures for each app individually. This process would take a tremendous amount of time and effort; instead, we defer the study on complex behavior enforcement to Section 5.4.

For each app in the dataset (Section 5.1), GuardRail first performs the static analysis. We discard apps for which Flowdroid exceeds the 20-min timeout limits because partial results on potential flows are insufficient to decide behavior matching conclusively. We then instrument these APKs with GuardRail and install them on test phones. We manually interact with these apps through all visible UI elements and navigate through different activities and lifecycle callbacks for up to a minute. We add instructions for the modified app to log down execution traces in the output. We collect and analyze the runtime execution logs to decide whether a behavior matching is detected.

Reduction in Static Flows. As explained in Section 5.2, 46 out of 1456 successfully terminate with FlowDroid with positive flows, of which GuardRail successfully rewrites 40 apps. Figure 8a shows the number of flows reported by FlowDroid. While 16 apps have no more than 32 flows, others have many more flows that one can manually inspect. Analyzing an app's behavior based on static flows alone would be an imprecise and time-consuming. We now present a detailed analysis of the 40 apps GuardRail successfully rewrites. GuardRail detects matching behavior with the signature in 6 apps and reports no matches in 23 cases. 11 apps have bugs that prevent normal execution and runtime analysis. In particular, *GuardRail eliminates 16 apps in which static analysis reports false-positive flows.*

To ensure correctness, we manually verified the 23 no-match cases and their program execution traces. The execution traces from 16 apps did not have any network requests containing user location data. We cross-examine traces with paths in FlowDroid reports and conclude that GuardRail successfully filtered the false-positive static analysis results in these cases. For the remaining 7 cases, GuardRail detects no matching behavior, but the location data indeed transmit through the network at runtime. We inspect the actual program execution trace, decompiled app bytecode, and paths reported from FlowDroid. We discover that FlowDroid reports incomplete results for these apps. Every path in the FlowDroid report is false-positive, and it misses the actual data transmission flows in the result. These cases reveal GuardRail's reliance on sound static-analysis backend. Section 6 discusses the impact to GuardRail from limitations from static analysis.

Reduction in Runtime Overhead. GuardRail doesn't encounter the same performance penalty as full-system taint tracking approaches. For example, TaintDroid [22] incurs 29% and 27% slowdown for operations like taking a picture and making IPC calls, respectively. Instead of evaluating overhead on real-apps, TaintDroid measures changes in benchmark scores from CaffeineMark [16] as an indicator of runtime overhead. In comparison, Figure 8b shows the expected runtime overhead for GuardRail-modified real apps, which we calculate using methods described in Section 5.2. We observe different degrees of impact codebase increase has on runtime overhead from the DroidBench tests (Figure 7d). We take the min, average, and max impact ratio from these results, and multiply them with real app codebase increase to calculate expected values. Based on our calculation, 90% of the apps will expect at best 0.4%, on average 1.1%, and at worst 3.8% runtime overhead.

5.4 Enforcing Real Data-Handling Policies

In this section, we show that GuardRail's specification language encompasses a wide range of complex data-handling behaviors that are typical of real apps. We start by analyzing a selection of popular applications to generalize private data handling behaviors. Then we develop 9 prototype apps following these behaviors. Finally, we evaluate the correctness of GuardRail's runtime enforcement of these policies on the prototype app suite, for which we have ground truth, as well as "stress tests" with unusual programming behaviors on a broader set of micro-benchmark apps. In principle, GuardRail should correctly enforce any given policy when provided with correct information from the static analysis. We empirically validate this assumption to ensure our evaluation results reflect the correct functioning of GuardRail's intended behavior.

Finding Signatures. To generalize common behavior signatures, we decompile 12 closed-source app binaries and obtain the source code of additional 4 popular Android apps for manual analysis. We include a detailed list of these apps in Appendix A. We analyze these app's source code or decompiled binary to understand how they handle private data processing. Eventually, we generalize several behavior specifications and use them for prototype development in Table 1.

Because reverse engineering and manual examination are time-consuming, we selected a shortlist of representative apps: 12 app binaries and 4 open-source apps. We analyzed 12 closed-source app binaries from 185k Google Play apps to understand location usage at large. We examined all weather apps and nearby-search apps from the dataset, and execute FlowDroid to collect flow information. We identified weather apps from their "Category" metadata, and applied a classification technique from prior work [36, 56, 60] to infer apps performing nearby-searches based on network traffic. We then selected the top apps with positive information flows and the highest popularity, sorted by review counts, to manually analyze their decompiled binary.

To incorporate apps that collect data from additional sources and more behaviors, we analyzed additional 4 popular open-source Android apps from Github, sorted by their stars. Their source code provides insights on more behavior signatures in different ways of using location data and camera photos.

Prototype App Development. We develop one prototype app for each behavior shown in Table 1, resulting in a test suite with 9 prototype apps with verified ground truth. Our prototype apps cover a variety of private data type and usage behavior. While several behaviors (Behavior 1-5) are closely related, each possesses unique features. Their small differences can potentially challenge GuardRail to distinguish similar signatures.

5.4.1 Correctness of Prototype Enforcement

In this section, we validate GuardRail's enforcement of behavior policies on the prototype apps described earlier and on a range of Java programming behaviors for transmitting sensitive information. For all tests in the prototype apps, GuardRail successfully reports behavior matching. When we evaluate apps with different behavior policies, GuardRail detects policy violations as well. One exception is that Once-Sent-Fetched behavior specification matches the Once-UI-Updated app This seeming mismatch is correct because the app performs a superset of the behavior described in the signature. Meanwhile, variations in Java program implementations challenge GuardRail's rewriter and dynamic-enforcement. We extend DroidBench benchmark to validate a broad coverage programming patterns in GuardRail.

Benchmark Framework. We develop our benchmark apps based on 76 DroidBench [19] test apps to evaluate GuardRail against a variety of programming behaviors, as explained in Section 5.1. Compared with the randomly-selected real apps, standard Android benchmarks have the following advantages. First, DroidBench covers a broad set of possible programming behaviors, some specifically designed to challenge analysis tools. Second, test apps in DroidBench contain clear ground truth about the programming patterns used. Finally, DroidBench is widely accepted and used in other Android program analysis research [57, 73, 78]. Hence, we believe our approach of evaluating the coverage and completeness of GuardRail on different program patterns using DroidBench is systematic.

To execute GuardRail with DroidBench apps, we provide a simple signature to express the shared behavior among all apps. This behavior describes transmitting tainted information directly to a specific sink. Although it isn't a sophisticated signature, it is sufficient to validate GuardRail's correctness across various programming patterns.

Results. Table 2 shows the results of executing GuardRail on 76 DroidBench test cases. GuardRail successfully instrument and track taint information correctly on 73 cases. 55 tests finish automatically, and 12 tests require interaction with UI elements to trigger sensitive information flow transmission. Addition-

Data Type	ID	Behavior Name	Behavior Explanation
Generic	1	Once Sent Fetched	App requests private data **once**, sends it to the network, fetches the result.
	2	Once UI Updated	Based on "Once Sent Fetched", the result is then displayed on the UI element.
	3	Polled Sent Fetched	Similar to "Once Sent Fetched", but the behavior repeats constantly.
	4	Polled UI Updated	Based on "Polled Sent Fetched", with additional step of updating the UI element.
	5	Polled Sent With Volley	Similar to "Polled Sent Fetched", but with Volley network library.
Location	6	Location for Customization	Based on our analysis of real-world weather apps, we use the same APIs to fetch location, retrieve weather information, and then display the update.
	7	Resolve Location and Display	Get user's location, resolve the physical address, and display the address on the UI fragment.
Camera	8	Decode QR Code	Use Camera to scan for QR code, then decode the embedded message.
	9	Send Media Content	Take picture from camera, and then send it with network traffic.

Table 1. Behavior descriptions of the prototype test apps we develop and use in Section 5.4. For **Generic** data, we apply behaviors for a variety of sensitive information sources, including location, phone state data (i.e., phone identifier), call logs, and SMS messages.

Successful	**73**
Automatic pass	55
Require user interaction	12
Manual inspection	6
Unsuccessful	**3**
Bug in dependent tools	1
Unsupported programming patterns	2

Table 2. DroidBench evaluation results. GuardRail successfully reports matching behavior in 73 of 76 apps. Most of the apps can be tested automatically or require minimal user interaction. In addition, we manually inspect and verify the instrumentation correctness on the decompiled app binaries of 6 apps, which either explore hard-to-trigger callback conditions or have internal bugs.

ally, 6 test cases can not be tested pragmatically, either because they use hard-to-trigger callback methods (`onLowMemory()` when OS experiences high memory pressure) or because they contain bugs preventing them from executed at runtime. For those 6 cases, we decompile the instrumented APK files and manually verify the taint tracking instruction instrumented correctly.

GuardRail encountered issues with 3 test cases. One test triggers a bug in Soot, a dependency for GuardRail instrumentation. Soot reports mismatching instructions from FlowDroid reports. The other 2 test cases use programming patterns not yet supported by GuardRail. One case requires tracking specific UI elements (e.g., password input field) as taint source, while GuardRail currently supports generic UI tracking (i.e., whether tainted data can flow to *any* UI element). Another case transmits tainted data through exception message objects. GuardRail passes 4 other exception tests in the benchmark, and we do not observe this unsupported behavior in the real app evaluation.

5.5 Integration Case Study

In this section, we examine the integration experience for existing applications and library SDKs with two real-world examples: one note-taking Android app and one advertising library SDK.

Note-Taking App. We use one of the popular open-source Android apps from previous subsection, Omni Notes [33], to evaluate the experience of creating a behavior specification and using GuardRail to enforce the spec as developers. App users can choose to attach their location when composing their notes. If location data is stored, the app will display the location information either in address or coordinates on the UI. To describe this behavior, we construct a specification based on the "Once UI Updated" behavior. Because developers implement a custom `Note` class to store the intermediate location information, we add APIs related to storing and accessing location data in Node objects into the event space (and hence the nodes in behavior graphs). Afterwards, we compiled the app into binary and ran it through the GuardRail instrumentation. The final execution successfully report matching behaviors when adding location data into a new note.

Third-Party Ad Library. Third-party libraries that make use of sensitive information may pose a challenge for app developers who want to use these libraries in conjunction with GuardRail. If the library provides neither source code or behavioral signature, then it may be difficult for the developer to write correct signatures that account for the library's behaviors on all paths.

We envision that library developers who wish to enable transparent data-handling practices will provide a behavior specification when distributing their library SDK. Previous studies show that only a small fraction of popular libraries account for a large percentage of the private data usage [12] in real apps, which suggests that successful deployment of GuardRail will only require the cooperation of a small number of well-established library developers in this manner. Even if a library does not provide a behavior signature, GuardRail can still rewrite and enforce developer-specified behaviors, blocking undisclosed behaviors in the library.

To evaluate the feasibility of this approach, we examined the open-source ad library MoPub [43]. We found that given a generic existing behavior signature such as “Once-UI-Updated”, adapting it for use with MoPub was straightforward for library developers; with source code access, we only needed to add a small number of additional API calls unique to MoPub’s pattern of constructing request URLs. We also validate the ease of adoption for third party library users, who only have access to library binary and library signature. With this single behavioral spec in hand, we were able to run GuardRail on an app that incorporates MoPub, successfully generating a rewritten APK that correctly enforces the specified location-data policy.

6 Discussion

In this section we discuss potential limitations of our approach and current prototype, and possible extensions that may address them.

Static Analysis Backend. As shown in Section 5.3, GuardRail introduces less runtime overhead than full dynamic taint analysis. The overhead reduction is because GuardRail keeps many instructions unmodified based on static analysis results. However, if the static analysis backend is not sound, or cannot scale to a targeted application, then GuardRail will inherit these limitations.

In particular, our current implementation relies on FlowDroid. Many previous research efforts report several limitations of FlowDroid, which echo our observation as well. These issues include low termination rates on real-world applications [51]; small fractions of reported network flows in real apps [4, 10]. Our experience with FlowDroid yields comparable termination and flow reporting rates in line with these prior works. Moreover, the incomplete system API coverage and scalability optimizations in FlowDroid can lead to under-approximation in results reported. FlowDroid may not be able to report apps using undocumented methods of extracting privacy sensitive information [55].

We enable the most conservative options during our experiments to avoid under-approximation. If FlowDroid fails to terminate, then to ensure correct enforcement of a given policy, the rewriter need to assume that *all* instructions in the app are on sensitive information paths, resulting in instrumentation that is as expensive as complete dynamic taint analysis. We also design the static analysis backend as a modular component in GuardRail, so that we make use of alternative dataflow engines if better options become available in the future.

Unsupported Platform Features. Our current prototype of GuardRail may fail on certain apps that use Android features that are currently unsupported. The prototype does not support programs that make use of native code or reflection for two primary reasons: FlowDroid does not support these features, and they are challenging for static analysis in general. However, it is possible to enforce policies on such programs by falling back to a purely-dynamic approach, which assumes that every reflection or native code instruction must be instrumented. Extending GuardRail to support them thus entails adding these instructions to the rewriter; as programs that use native code and reflection are somewhat rare, we did not see the need to do so for the purposes of our evaluation.

GuardRail’s static analysis backend requires summaries of system library APIs to decide propagation rules. Our current prototype implements a sufficient set of these to function correctly on the set of real apps examined in our evaluation. While it is not difficult to add these to the prototype, doing so requires care as inaccurate summaries can introduce false-positive results.

7 Related Work

There is a significant amount of prior work on improving the security and privacy of mobile platforms and applications. To facilitate our comparisons, we group prior approaches into the following categories: behavior signature-based techniques, static analysis and type-based techniques, dynamic policy enforcement, and human-readable privacy policy analysis.

Behavioral Policy Enforcement. Numerous prior work has applied behavior signatures to detect and mitigate malware on mobile platforms. Astroid [26] abstracts data flows, intents, and certain API calls as inter-component call graphs to augment malware signatures. DroidSIFT [79] uses weighted contextual API dependency graph to generate malware signatures. SARRE [40] and Malton [75] use dynamic runtime information to generate signatures for detecting malware and information leakage. All of these work use heuristics to represent malware signatures, and attempt to blacklist undesirable behavior against an adaptive adversary. In contrast, GuardRail uses behavior signatures to enable developers to disclose private data handling behavior, proposing a solution for more opaque data transparency.

Similar behavior graph-based policy enforcement has been proposed in other platforms (e.g. Windows malware [42], browser extensions [27]). GuardRail focuses on enforcing behavior policy in mobile applications, which introduces new challenges related to the execution model and resource constraints that GuardRail is designed to address.

It is a well-established approach to rewrite program and inline reference monitoring to enforce security policies [23, 58]. Several prior works adopt this idea to enforce policies in Android [7, 9, 17, 35, 50, 74]. Moreover, a few research projects bring the static and dynamic combination into Android applications [5, 32, 59]. The main differentiating factor for GuardRail is the newly proposed specification language, which can be used to describe a more diverse set of behaviors and safety properties. We then combine the behavior specification with the hybrid enforcement efficiently and accurately.

Program Analysis and Types. We build GuardRail upon FlowDroid [3], a renowned Android static analysis tool. Several research work have improved the state-of-the-art static analysis tool [39, 47, 48, 69]. On the other hand, TaintDroid [22] and TaintART [64] employ dynamic taint analysis for Android, and Weir [44] proposes a dynamic Information Flow Control system with label context-sensitivity. AppsPlayground [53] and AppAudit [73] use a hybrid approach instead, combining static and dynamic analysis to track information leakage, but only test applications for leaks in general rather than checking for policy compliance. GuardRail takes a similar hybrid approach combining static analysis and dynamic enforcement. However, it uses the information provided by static analysis to selectively instrument the target application only as needed for the policy, and therefore imposes less overhead compared to full system dynamic tracking.

Several approaches incorporate extra program context information, such as information from UI elements and system environments [51, 77], or transmission and reception sensitive flows [10], with taint analysis results to improve greater precision. In comparison, GuardRail maintains context-sensitivity throughout function execution to differentiate flows, while leaving out context information from user interface events to simplify behavior specifications.

Ernst *et al.* [24] propose a behavioral policy specification and enforcement based on a novel type system. Their system needs source code access, and requires developers to learn a new type system and migrate their application to the new framework correctly. In contrast, GuardRail works on application binaries, and does not require developers to use new or unconventional tools, addressing concerns about intellectual property and related hurdles for adoption.

Transparent Data Practices. A growing body of research examines applications' privacy policies expressed as legal or human-readable documents. However, prior work [63, 71] suggests that even policies intended to be readable may be challenging for non-experts to understand and incorporate into decisions about privacy. Several works apply network-based analysis or execution context to understand how app use private data [54, 56, 61, 70]. In GuardRail, the type of behavior policy and signature we use to enforce accountable data usage are of a different nature. We propose a novel machine-enforceable policy description of app's behavior regarding private user data. Thus, we see efforts that aim to address shortcomings in readable policies as complementary to our vision for GuardRail.

8 Conclusion

This paper presents GuardRail, a system that addresses the lack of transparency in mobile application's private data processing, and increases developers accountability in preserving user privacy in the app. GuardRail proposes a novel expressive behavior policy enforcement system with its own high-level behavior specification language. We build GuardRail with many tools available for Android, including a static analysis backend and instrumentation frameworks.

We adapt the concept of behavior graphs from malware signature research into Android applications. With extensions and adaptations into mobile application events, we demonstrate the expressiveness and usefulness of our specification language in describing high-level program behaviors related to privacy sensitive data usage and egress by Android apps.

We implement and evaluate GuardRail on a set of 76 DroidBench benchmark apps, 9 prototype applications developed from complex data handling practices, and 40 real world applications show that GuardRail is a useful approach for enforcing rich, stateful privacy policies while imposing low runtime overhead.

References

[1] Apktool - a tool for reverse engineering 3rd party, closed, binary android apps. https://ibotpeaches.github.io/Apktool/, 2019.

[2] Steven Arzt. *Static Data Flow Analysis for Android Applications*. PhD thesis, Darmstadt University of Technology, Germany, 2017.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[4] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 426–436, Piscataway, NJ, USA, 2015. IEEE Press.

[5] M. Backes, S. Bugiel, O. Schranz, P. Von Styp-Rekowsky, and S. Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 481–495, 2017.

[6] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. R-droid: Leveraging android app analysis with static slice optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 129–140, New York, NY, USA, 2016. ACM.

[7] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard - fine-grained policy enforcement for untrusted android applications. In *DPM/SETOP*, 2013.

[8] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[9] Xin Chen, Heqing Huang, Sencun Zhu, Qing Li, and Quanlong Guan. Sweetdroid: Toward a context-sensitive privacy policy enforcement framework for android os. In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*, WPES '17, pages 75–86, New York, NY, USA, 2017. ACM.

[10] Xin Chen and Sencun Zhu. Droidjust: Automated functionality-aware privacy leakage analysis for android applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 5:1–5:12, New York, NY, USA, 2015. ACM.

[11] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. Is this app safe?: A large scale study on application permissions and risk signals. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 311–320, New York, NY, USA, 2012. ACM.

[12] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I. Hong, and Yuvraj Agarwal. Does this app really need my location?: Context-aware privacy management for smartphones. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3):42:1–42:22, September 2017.

[13] Civilsphere. Mobile (in)security series: Location leaked over the network by android application. https://www.civilsphereproject.org/blog/2018/8/30/mobile-insecurity-series-application-leaks-location-over-the-network, 2019.

[14] CNET. Google sued over tracking user location amid privacy concerns. https://www.cnet.com/news/google-sued-over-keeping-location-data-amid-privacy-concerns/, 2018.

[15] European Commission. 2018 reform of EU data protection rules. https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en, 2018.

[16] PENDRAGON SOFTWARE CORPORATION. Caffeinemark 3.0. http://www.benchmarkhq.ru/cm30/.

[17] Benjamin Davis, Ben S, Armen Khodaverdian, and Hao Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In *In Proceedings of the Mobile Security Technologies 2012, MOST '12. IEEE*, 2012.

[18] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[19] Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for android. https://github.com/secure-software-engineering/DroidBench, 2019.

[20] Peter Eckersley. How unique is your web browser? In Mikhail J. Atallah and Nicholas J. Hopper, editors, *Privacy Enhancing Technologies*, pages 1–18, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[21] Pardis Emami-Naeini, Henry Dixon, Yuvraj Agarwal, and Lorrie Faith Cranor. Exploring how privacy and security factor into iot device purchase behavior. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.

[22] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[23] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[24] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1092–1104, New York, NY, USA, 2014. ACM.

[25] Kassem Fawaz, Huan Feng, and Kang G. Shin. Anatomization and protection of mobile apps' location privacy threats.

In *24th USENIX Security Symposium (USENIX Security 15)*, pages 753–768, Washington, D.C., August 2015. USENIX Association.

[26] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[27] Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas Reps, Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 548–563, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] Google. Google play: Privacy, security, and deception. https://play.google.com/about/privacy-security-deception/, 2019.

[29] Google Play Store. App Signing by Google Play. https://developer.android.com/studio/publish/app-signing#app-signing-google-play. Retrieved on 11/15/19.

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

[31] GrammaTech. CodeSonar. https://www.grammatech.com/products/codesonar, 2019.

[32] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek. Flow permissions for android. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 652–657, 2013.

[33] Federico Iosue. Omni-Notes open source note-taking application for android. https://github.com/federicoiosue/Omni-Notes, 2020.

[34] skylot/jadx: Dex to Java decompiler. https://github.com/skylot/jadx, 2019.

[35] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM.

[36] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. Why are they collecting my data?: Inferring the purposes of network traffic in mobile apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(4):173:1–173:27, December 2018.

[37] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, page 3393–3402, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Yong-Fong Lee and Barbara G. Ryder. A comprehensive approach to parallel data flow analysis. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 236–247, New York, NY, USA, 1992. ACM.

[39] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[40] Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. Sarre: Semantics-aware rule recommendation and enforcement for event paths on android. *Trans. Info. For. Sec.*, 11(12):2748–2762, December 2016.

[41] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 501–510, New York, NY, USA, 2012. ACM.

[42] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] MoPub. Mopub. https://www.mopub.com, 2019.

[44] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical difc enforcement on android. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 1119–1136, Berkeley, CA, USA, 2016. USENIX Association.

[45] New York Times. Google's Sundar Pichai: Privacy should not be a luxury good. https://www.nytimes.com/2019/05/07/opinion/google-sundar-pichai-privacy.html, 2019.

[46] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Pub. Co., 1971.

[47] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 469–484, New York, NY, USA, 2016. ACM.

[48] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[49] Office of the Privacy Commissioner of Canada. The personal information protection and electronic documents act (pipeda). https://www.priv.gc.ca/en/privacy-topics/privacy-laws-in-canada/the-personal-information-protection-and-electronic-documents-act-pipeda/, 2019.

[50] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[51] Xiang Pan, Yinzhi Cao, Xuechao Du, Boyuan He, Gan Fang, Rui Shao, and Yan Chen. Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1669–1685, Baltimore, MD, 2018. USENIX Association.

[52] The Washington Post. The U.S. government fined the app now known as TikTok $5.7 million for illegally collecting children's data. https://www.washingtonpost.com/technology/2019/02/27/us-government-fined-app-now-known-tiktok-million-illegally-collecting-childrens-data/, 2019.

[53] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[54] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. Haystack: A multi-purpose mobile vantage point in user space, 2015.

[55] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 603–620, Santa Clara, CA, August 2019. USENIX Association.

[56] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 361–374, New York, NY, USA, 2016. Association for Computing Machinery.

[57] Alireza Sadeghi, Reyhaneh Jabbarvand, Negar Ghorbani, Hamid Bagheri, and Sam Malek. A temporal permission analysis and enforcement framework for android. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 846–857, New York, NY, USA, 2018. ACM.

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

[59] Julian Schütte, Dennis Titze, and J. M. De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '14, page 370–379, USA, 2014. IEEE Computer Society.

[60] Yihang Song and Urs Hengartner. Privacyguard: A vpn-based platform to detect information leakage on android devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, pages 15–26, New York, NY, USA, 2015. ACM.

[61] Gaurav Srivastava, Kunal Bhuwalka, Swarup Kumar Sahoo, Saksham Chitkara, Kevin Ku, Matt Fredrikson, Jason Hong, and Yuvraj Agarwal. Privacyproxy: Leveraging crowdsourcing and in situ traffic analysis to detect and mitigate information leakage, 2017.

[62] State of California. California consumer privacy act of 2018. https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375, 2018.

[63] Peter Story, Sebastian Zimmeck, Abhilasha Ravichander, Daniel Smullen, Ziqi Wang, Joel Reidenberg, N. Cameron Russell, and Norman Sadeh. Natural language processing for mobile app privacy compliance. *AAAI Spring Symposium on Privacy-Enhancing Artificial Intelligence and Language Technologies*, March 2019.

[64] Mingshen Sun, Tao Wei, and John C.S. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 331–342, New York, NY, USA, 2016. ACM.

[65] TechCrunch. Apple will require all apps to have a privacy policy as of October 3. https://techcrunch.com/2018/08/31/apple-will-require-all-apps-to-have-a-privacy-policy-as-of-october-3/, 2018.

[66] Transparent clock & weather - apps on google play. https://play.google.com/store/apps/details?id=com.droid27.transparentclockweather, 2019.

[67] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[68] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I. Hong. Understanding the purpose of permission use in mobile apps. *ACM Trans. Inf. Syst.*, 35(4):43:1–43:40, July 2017.

[69] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.*, 21(3):14:1–14:32, April 2018.

[70] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, Washington, D.C., August 2015. USENIX Association.

[71] Shomir Wilson, Florian Schaub, Frederick Liu, Kanthashree Mysore Sathyendra, Daniel Smullen, Sebastian Zimmeck, Rohan Ramanath, Peter Story, Fei Liu, Norman Sadeh, and Noah A. Smith. Analyzing privacy policies at scale: From crowdsourcing to automated annotations. *ACM Trans. Web*, 13(1):1:1–1:29, December 2018.

[72] Wired. FTC reportedly hits facebook with record $5 billion settlement. https://www.wired.com/story/facebook-ftc-fine-five-billion/, 2019.

[73] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 899–914, Washington, DC, USA, 2015. IEEE Computer Society.

[74] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 27–27, Berkeley, CA, USA,

2012. USENIX Association.

[75] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards on-device non-invasive mobile malware analysis for art. In *Proceedings of The 26th USENIX Security Symposium (Security'17)*, August 2017.

[76] K. Yang, K. Zhang, J. Ren, and X. Shen. Security and privacy in mobile crowdsourcing networks: challenges and opportunities. *IEEE Communications Magazine*, 53(8):75–81, 2015.

[77] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 303–313, Piscataway, NJ, USA, 2015. IEEE Press.

[78] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security-centric descriptions for android apps. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 518–529, New York, NY, USA, 2015. ACM.

[79] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1105–1116, New York, NY, USA, 2014. ACM.

Appendix

A Apps for Generalize Behaviors

We analyzed a list of open and closed source apps to generalize behavior specifications. Table 3 shows the list of closed-source app binaries we used to decompile and manual analysis. In our 184k Google Play app dataset, we have app metadata for the number of user-provided ratings on Google Play store. We retrieve the latest install count from their current Google Play store app listing. However, several apps are no longer actively listed, so their current install count is not applicable. We only list their rating count from the metadata, as shown in our dataset, from which we can extrapolate the range of their install count. Table 4 lists open-source apps we analyzed. We selected Android apps from Github, sorted their popularity by the number of stars they received. After selecting these 4 apps, we retrieve their install counts from the Google Play store to illustrate their popularity with our closed-source app binary dataset.

Package Name	Google Play Rating Count	Google Play Installs
tr.gov.mgm.meteorolojihavadurumu	98,189	1,000,000+
net.darksky.darksky	23,895	1,000,000+
ru.rp5.rp5weather	20,429	1,000,000+
weather.widget.weatherforecast	4,102	N/A
com.boom.meteo.views	2,924	500,000+
com.sam.meteo.views	2,091	500,000+
net.dmdigital.meteoindiretta	1,054	100,000+
com.california.cowboy.studios.wifi.finder	190	N/A
com.project.project2	167	10,000+
br.com.s2it.clubeuol	161	10,000+
com.project.e_services	106	N/A
com.rain.nearme	77	10,000+

Table 3. List of closed source apps used in behavior generalization. We decompile every app and analyze how they use location data for personalization purposes. Section 5.4 explains the app selection process.

App Name	GitHub Stars	Google Play Installs
Telegram	14k	500,000,000+
Signal	13.1k	10,000,000+
TutaNota	3.3k	100,000+
Omni Notes	2.1k	100,000+

Table 4. List of open source apps used in behavior generalization. We select the most starred Android projects on GitHub and analyze how they use various private user data (location and other types). See Section 5.4 for more details.