ABSTRACT
A good graphical user interface (GUI) is crucial for an application’s usability, so vendors and regulatory agencies increasingly place restrictions on how GUI elements should appear to and interact with users. Motivated by this concern, this paper presents a new technique (based on static analysis) for checking conformance between (Android) applications and GUI policies expressed in a formal specification language. In particular, this paper (1) describes a specification language for formalizing GUI policies, (2) proposes a new program abstraction called an event-driven layout forest, and (3) describes a static analysis for constructing this abstraction and checking it against a GUI policy. We have implemented the proposed approach in a tool called Venus, and we evaluate it on 2361 Android applications and 17 policies. Our evaluation shows that Venus can uncover malicious applications that perform ad fraud and identify violations of GUI design guidelines and GDPR laws.

CCS CONCEPTS
• Software and its engineering → Software verification and validation;
• Security and privacy → Human and societal aspects of security and privacy;
• Software security engineering.

KEYWORDS
ad fraud, static analysis, user interface, Android, mobile app

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
https://doi.org/10.1145/3468264.3468561
In practice, checking conformance between an app and a GUI policy turns out to be challenging for two key reasons. First, Android applications consist of several interacting activities, all of which provide a different and dynamically changing interface. Thus, checking adherence to a GUI policy requires exploring the (possibly infinite) ways that a user can interact with the app. Second, by studying existing GUI policies, we found that many of them concern not only the static appearance of the app, but also how the interface needs to dynamically evolve as users interact with it. Thus, verifying an app against a GUI policy requires reasoning about the dynamic behavior of the app in relation to the GUI elements it provides.

In this paper, we address these challenges through an end-to-end solution that statically reasons about an app’s GUI-related behaviors. Our solution consists of three ingredients that make it possible to specify and check such properties:

1. **Policy language**: We present a formal policy language called **Vesper** for expressing realistic GUI design guidelines. **Vesper** allows specifying both spatial relations between GUI elements as well as their behavioral properties, such as how a button should react to a click event.

2. **ELF abstraction**: We propose a new program abstraction called **Event-driven Layout Forest (Elf)** that summarizes spatial and behavioral properties of GUI elements. While Elf bears resemblance to other Android abstractions like window transition graph [46] and ICCG [11], it differs from them in that nones correspond to individual GUI elements (rather than activities) and node labels (computed using numeric abstract domains and pointer analysis) track GUI-related properties.

3. **Conformance checking**: To check whether an Android app corresponds to a **Vesper** specification, **Venus** needs to decide whether a given Elf abstraction is a model of the input **Vesper** specification. **Venus** achieves this task by encoding both the Elf abstraction and the **Vesper** policy as logical formulas and reduces conformance checking to a satisfiability query.

To evaluate the effectiveness of our proposed approach, we performed an extensive experimental evaluation on 2361 Android applications. Specifically, we formalized existing GUI policies as **Vesper** specifications and then used **Venus** to check each Android application against these policies. Our evaluation shows that **Venus** is able to accurately pinpoint violations of GUI policies with a low false positive rate (around 6.9%). Furthermore, **Venus** can identify previously unknown ad fraud instances and detect violations of a subset of GDPR (General Data Protection Regulation) regulations.

In short, this paper makes the following key contributions:

- We propose a policy language called **Vesper** for describing GUI policies (Section 4).
- We introduce a new program abstraction called **event-driven layout forest** that is suitable for checking such GUI policies and present a static analysis technique for automatically constructing the proposed Elf abstraction (Section 5).
- We implement **Venus**, the first tool for statically checking conformance between Android applications and GUI specifications, and we extensively evaluate **Venus** by checking conformance between 2361 Android applications and several existing GUI policies (Section 6).

Figure 2: Example demonstrating a typical event-driven flow in Android apps. Listing 1 defines the layout shown in (a) and (b). The transition from (b) to (c) is defined in listing 2.

```
1 <ConstraintLayout>
2   ...
3   <TextView android:id="@+id/demo_title" android: text="Default title" ... />
4   <ImageView app:layout_constraintTop_toBottomOf="@+id/demo_title" ... />
5   <Button android:id="@+id/continue_button" android: text="CONTINUE" ... />
6 </ConstraintLayout>
```

Listing 1: Activity layout for the app shown in figure 2(b).

```
1  class MainActivity extends Activity {
2   void onCreate(...) {
3     ...
4     setContentView(R.layout.activity_main);
5     TextView demoTitle = findViewById(R.id.demo_title);
6     demoTitle.setText("Venus Demo");
7     Button continueButton = findViewById(R.id.
8       continue_button);
9     continueButton.setOnClickListener(new View.
10        OnClickListener() {
11          void onClick(View v) {
12            AlertDialog d = new Dialog(...) .create();
13            d.setButton(DialogInterface.BUTTON_POSITIVE, "YES", DialogInterface.OnClickListener () { }
14            void onClick(...) { d.dismiss(); }
15          });
16          d.show();
17        });
18   }
```

Listing 2: onCreate source code for activity from figure 2(b).

2 BACKGROUND ON ANDROID GUI

In Android, the basis of an app’s user interface is an activity, which always has a window associated with it. Activities can start other activities by a message-passing system known as inter-component communication (ICC). An Android ICC message is an Intent, which can be thought of as a description of what the launched component should do. An Intent object specifies both the action to perform (e.g., view, edit, etc.) and provides the relevant data.

The Android framework provides two types of basic GUI elements, namely Views and ViewGroup. A View element is a widget, such as a button or progress bar, that the user can see and interact with. A ViewGroup is an invisible container that stitches together Views and ViewGroup. Android provides different types of ViewGroup, such as a Linear Layout, for arranging GUI elements horizontally or vertically. The user interface of a GUI activity corresponds to a tree data structure (figure 2(a)), where internal nodes are
Declaring and manipulating GUI elements. In Android, there are two ways to declare GUI elements. The first option is to specify the layout through an XML file. In addition to defining the hierarchical user interface of an activity, the XML file can also specify the attribute values of each GUI element, such as the text attribute “CONTINUE” of a button on line 5 of listing 1. During compilation, the XML file is translated into a so-called layout resource that can be loaded in the application’s source code by calling setContentView (R.layout.layout_name) (e.g., line 4 of listing 2). An alternative way to create a layout is to do so programmatically by calling methods provided by the Android framework. For instance, rather than statically declaring the text attribute in the XML file, a program can do this at run time by calling the setText method.

In practice, programmers often combine XML-based declaration of GUI elements with programmatic modifications at run time. For example, line 4 of listing 2 loads the layout declared in the XML file, but the two subsequent lines modify the title of the nested TextView element to “Venus Demo” from its original name (“Default title”) declared in line 3 of listing 1. Hence, understanding an application’s user interface requires analysis of both XML files and source code.

Interacting with GUI elements. To facilitate interaction with users, GUI elements register callbacks that get invoked upon specific types of user events (e.g., click, hover, etc.). In particular, Android GUI elements can respond to events of type X by registering an OnXListener object whose OnX method gets executed when event X occurs. For instance, lines 8–15 in listing 2 cause the widget to pop up a dialog box when the user clicks “CONTINUE.” This behavior is illustrated in the transition from figure 2(b) to figure 2(c).

3 OVERVIEW

This section gives an overview of the VENUS framework through a simple but realistic motivating example.

3.1 Example GUI Policy for AdFraud Detection

Fig. 3 shows the screenshot of an ad fraud application called “Super Cleaner” that was recently submitted to the Google Play Store. This app does not conform to a Google AdMob policy [17], which states that transparent backgrounds should not display ads upon a click event. However, as shown in parts (b) and (c) of figure 3, the Super Cleaner application blatantly violates this policy.

In order to use VENUS to check conformance between this app and the AdMob policies, the user first needs to formalize the policy in VENUS’s specification language. In particular, figure 5 shows a formalization for the policy “transparent backgrounds should not display ads upon click events” in our policy language called VESPER. Here, the first line declares a View element called bg. Next, the assumes statement stipulates that bg is the background of some other View element. Then, on line 3, the let binding defines a custom predicate called popAd?, which evaluates to true if clicking on v shows a new window v’ that corresponds to an adView GUI element. Finally, the assertion specifies the desired property. Section 4 will present more about VESPER.

Figure 3: Clicking on the white space will (surprisingly) trigger the display of an untrusted website.

<table>
<thead>
<tr>
<th>Listing 3: Main activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class MainActivity extends Activity {</td>
</tr>
<tr>
<td>private Button saverBtn;</td>
</tr>
<tr>
<td>protected void onCreate(Bundle bundle) {</td>
</tr>
<tr>
<td>setContentView(R.layout.activity_main);</td>
</tr>
<tr>
<td>saverBtn = findViewById(R.id.btn_save);</td>
</tr>
<tr>
<td>saverBtn.setOnClickListener(this);</td>
</tr>
<tr>
<td>public void onClick(View view) {</td>
</tr>
<tr>
<td>Intent intent = new Intent(this, BatterySaver.class);</td>
</tr>
<tr>
<td>startActivity(intent);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>class NativeAdViewBuild {</td>
</tr>
<tr>
<td>public View f() {</td>
</tr>
<tr>
<td>View adView = new UnifiedNativeAdView();</td>
</tr>
<tr>
<td>View bgView = findViewById(R.id.bg_view);</td>
</tr>
<tr>
<td>// set a transparent background</td>
</tr>
<tr>
<td>bgView.setOpacity(0);</td>
</tr>
<tr>
<td>bgView.setOnClickListener(this);</td>
</tr>
<tr>
<td>return adView;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>public void onClick(View arg0) {</td>
</tr>
<tr>
<td>loadUrl(&quot;<a href="http://funtest.amatwallet.com">http://funtest.amatwallet.com</a>&quot;); // suspicious URL</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 4: Battery saver activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class BatterySaver extends Activity {</td>
</tr>
<tr>
<td>public void onCreate(Bundle bundle) {</td>
</tr>
<tr>
<td>setContentView(R.layout.battery_saver_ad);</td>
</tr>
<tr>
<td>FrameLayout frameLayout = (FrameLayout) findViewById(R.id.content);</td>
</tr>
<tr>
<td>View a = new NativeAdViewBuild().f();</td>
</tr>
<tr>
<td>frameLayout.addView(a);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Figure 4: Simplified source code for the Super Cleaner app.

Given this VESPER policy and the source code of the Super Cleaner application (shown in Figure 4), we next explain how VENUS automatically identifies this policy violation.

3.2 ELF Generation via Static Analysis

As mentioned earlier, VENUS uses static analysis to construct an even-driven layout forest (ELF) abstraction of the application. At a
1. View bg
2. assume (∃v. (View(v) ∧ background(bg,v)))
3. let popAd(v) = ∃v’. (showWindow(v, c1lick, v’) ∧ AdView(v’))
4. assert (transparent(bg) → ¬popAd(bg))

Figure 5: Vesper specification for the policy “Transparent backgrounds should not be clickable”.

high-level, this abstraction captures all relevant behavior of the app with respect to the Vesper policy language. For example, figure 6 shows the Elf abstraction for the Super Cleaner application. Here, each node corresponds to a GUI element; node labels (e.g., for bgView) indicate attribute values (e.g., opacity, width); and there are two types of edges: (1) a spatial (solid) edge from node n to n’ indicates that GUI element n’ is nested inside n, and (2) a behavioral (dashed) edge from n to n’ labeled with e indicates that GUI element n launches another GUI element n’ upon event e. For example, in figure 6, there is a spatial (solid) edge from MainActWindow to saverBtn since the latter is spatially nested within the window of the main activity (see figure 3). On the other hand, there is a behavioral (dashed) edge from saverBtn to BatterySaverWindow labeled with showWindow(c1lick) because clicking on the saverBtn results in opening the window of the BatterySaver activity (see code 3).

In practice, constructing a sufficiently precise Elf abstraction of the application requires non-trivial static analysis. For example, the construction of behavioral edges between GUI elements requires reasoning about heap objects and callbacks as well as analysis of inter-component communication (ICC). On the other hand, reasoning about GUI element attributes (e.g., height, width) requires reasoning about numeric values.

3.3 Checking Conformance

Our method uses the computed Elf abstraction to check conformance against any Vesper policy. At a high-level, we can think of the Elf abstraction as defining a conjunction of ground predicates in Vesper. Thus, checking conformance between the app and policy boils down to determining whether the formula defined by the Elf abstraction implies the specification. Going back to our example, we can determine that Super Cleaner violates the Vesper policy from Figure 5 using the following chain of inferences:

- First, since bgView is nested inside nativeAdView and has the same width/height of its parent (figure 6), we determine that bgView is the background of nativeAdView. Thus, bgView satisfies the assumption from line (2) of Figure 5.
- Next, because the opacity attribute of bgView is 0 (see figure 6), transparent evaluates to true for bgView.
- In addition, bgView satisfies the popAd predicate because figure 6 contains a behavioral edge from bgView to adViewWindow labeled with c1lick.
- Finally, because bgView satisfies both the assumption at line (2) as well as the transparent and popAd predicates, the assertion at line (4) of Figure 5 is violated.

Therefore, Venus reports that the Super Cleaner app does not conform to the Vesper policy from Figure 5.

4 VESPER SPECIFICATIONS

As shown in figure 7, a specification in Vesper starts with a set of declarations, followed by a sequence of statements (i.e., definitions and assumptions), and ends in a set of assertions. While Vesper provides built-in predicates relevant to the spatial and behavioral properties of GUI elements (figure 8), the user can also define custom predicates through let bindings. For instance, in figure 5, showWindow is an example of a built-in predicate, whereas popAd is a custom predicate defined by the user. Vesper also provides a way to define a set of GUI elements through the set comprehension syntax { v | Φ }.

Expressions. In Vesper, the most basic expressions are variables v, integer constants c, and predefined Android events e such as click or touch. Vesper allows performing arithmetic operations over integers as well as aggregation over sets. For instance, the expression count(v) returns the number of elements in set v.

Built-in predicates. Vesper provides a core set of built-in predicates that constrain spatial and behavioral properties of GUI elements. Figure 8 shows examples of these predicates, which are classified into three categories:

- **Element type predicates** describe the type of a GUI element (e.g., button, dialog). Note that, unlike the actual Android API, Vesper does not differentiate between views and view groups, and every GUI element is considered to be a view. Thus, views can contain nested views under Vesper’s semantics.
- **Spatial predicates** refer to visual properties of GUI elements (e.g., height, width) as well as spatial relationships between different GUI elements (e.g., containment).
- **Behavioral predicates** constrain how GUI elements react to user events (e.g., what methods they can invoke, which other GUI elements they can display, etc.).

**Example 1.** Consider the following Vesper specification:

View w;
let LView(v) = ∃x,y. (width(x,y) ∧ x > 100 ∧ height(x,y) ∧ y > 100)
let LAds = { v | AdView(v) ∧ contains(w, v) ∧ LView(v) } assert count(LAds) ≤ 1

This specification requires that every window contains at most one “large” ad, meaning that the width and height of the ad is above a certain threshold. Here, the combination of set comprehension syntax and the count function allows constraining the number of GUI elements with a certain property.

We present the formal semantics of Vesper policies in Appendix A. At a high level, the semantics of Vesper policies are defined over execution traces, and we consider a predicate p(t) to be true in an execution ω if it holds on objects t at any time during ω. For example, the predicate startBrowser(e, v) evaluates to true in execution ω if e starts the browser at some point during ω. Given the truth value of built-in predicates under ω, evaluation of the full policy under ω largely follows the standard semantics of first-order logic, with some modifications to handle set comprehension (see
Finally, we say that an app

\[ \text{Vesper} \]

\[ \text{Elf} \]

\[ \text{Spatial predicates:} \]

- `height(v, h)`
- `width(v, w)`
- `textSize(v, s)`
- `transparent(v)`
- `contains(u, v)`
- `background(u, v)`

\[ \text{Behavioral predicates:} \]

- `entryView(v)`
- `invoke(u, e, m)`
- `showWindow(u, e, o)`
- `launchDialog(w, e, o)`
- `startMarketplace(e, o)`
- `startBrowser(e, o)`

\[ \text{Figure 7: The Vesper policy language} \]

\[ \text{Figure 6: Simplified Elf for motivating example of Sec.3.1. Solid (resp. dashed) lines represent spatial (resp. behavioral) edges.} \]

\[ \text{Example 2. Consider the following layout XML:} \]

```xml
<LinearLayout id="lin" orientation="vertical">
    <TextView id="txt" width=100 height=200>
        "Hello, I am a TextView" />
</LinearLayout>
```
[...]

Figure 9: Layout Schema Definition

We represent this as the following layout schema:

\[
\Psi(l) = \{\text{orientation} \mapsto (\text{string}, "vertical"), \text{subview} \mapsto (\text{TextView, DefaultVal(txt1)})\}
\]

\[
\Psi(txt1) = \{\text{width} \mapsto (\text{Int, 100}), \text{height} \mapsto (\text{Int, 200}), \ldots\}
\]

5.3 Static Analysis

In this section, we describe our static analysis for computing the Elf abstraction using Datalog-style inference rules. Note that the event-driven layout forest is a global abstraction of the entire application; however, our static analysis for computing is both flow- and context-sensitive. Our analysis leverages the layout schema extracted from the XML file (Section 5.2) as well as the results of standard techniques like pointer analysis.

We formalize our static analysis using three different types of predicates (summarized in Table 1):

- **Source code predicates** refer to statements in the source code. For instance, `addView(l, m, v1, v2)` indicates that there is an API call of the form `addView(v2)` at location `l` of method `m`.

- **Pre-analysis predicates** refer to program facts computed by off-the-shelf static analyzers. For example, `pointsTo(c, l, v)` indicates that variable `v` points to heap object `o` at program location `l` and calling context `c`. Similarly, `val(c, l, o, v)` indicates that variable `v` has abstract value `a` at location `l` in calling context `c`.

- **Output predicates** collectively define our Elf abstraction. For example, the predicate `sAttrib(o, a, val)` indicates that the abstract value for spatial attribute `a` of object `o` is `val`, and `bEdge(o, e, o')` indicates that there is a behavior edge between `o` and `o'` labeled `e`.

As mentioned earlier, we present our static analysis (see Figure 10) using Datalog-style rules of the form:

\[
H(x_1, \ldots, x_n) \leftarrow B_1(\ldots), \ldots, B_k(\ldots).
\]

The meaning of such a rule is that the predicate `H(x_1, \ldots, x_n)` is true if all the of the predicates `B_1, \ldots, B_k` in the rule body are satisfied. We refer to `H` as the head predicate and the `B_i`’s as body predicates. In our case, the head predicates are either auxiliary or output predicates computed by our analysis, whereas body predicates also involve source code and pre-analysis predicates. If an argument to a predicate does not matter, we use the symbol "_" to indicate that it matches anything.

We now explain the rules from Figure 10 in more detail.

**Nodes.** According to the first rule in Figure 10, any (abstract) heap object that corresponds to a GUI element (i.e., is subtype of `View`) is a node in the event-driven layout forest abstraction.

**Root view.** The second rule computes a predicate `rootView(o, o')` indicating that Activity `o` sets its main window to be GUI element `o'`. Since root views are set via an API call `v.setContentView(v')`, this rule looks up the heap objects pointed to by variables `v, v'` at the program location `l` (in method `m`) where the API call occurs. Note that our analysis is context-sensitive in that we look up the points to sets of `v, v'` in feasible calling contexts of `m`.

**Entry view.** The next rule marks the initial nodes of the Elf abstraction. To determine the initial nodes, we first identify all heap objects `o` that are of instance of type `A`, where `A` is the main activity of the application. We then mark all root views of `o` as initial nodes using the auxiliary `rootView` predicate from rule (2).

**Behavioral attributes.** The next rule, (4), describes how we compute behavioral attributes of each node. In particular, behavioral attributes map each GUI event to a set of methods that can be used to handle that event. Since event handlers are registered via `setListener` methods, this rule uses the `setEventListener(l, m, v')` source code predicate, which indicates that method `m'` is registered as the listener for event `X` for variable `v`, and `l, m` correspond to the program location and method where the registration occurs respectively. If `v` points to a heap object `o` that is a node in the Elf abstraction, behavioral attribute `X` is mapped to method `m'`. Note that, in general, there may be multiple methods `m_1, \ldots, m_k` that are used to handle event `X`. In this case, our analysis computes multiple facts of the form `bAttrib(o, X, m_i)`, \ldots, `bAttrib(o, X, m_k)` meaning that behavioral attribute `X` is mapped to the set `{m_1, \ldots, m_k}`.

**Spatial attributes.** The next three rules, (5)–(7), describe the computation of spatial attributes. Unlike behavioral attributes that have a finite domain (i.e., a set of methods), spatial attributes like height have an infinite domain (i.e., all integers). Thus, our method uses abstract interpretation to reason about such attributes. In particular, rule (5) initializes all spatial attributes to `_`, as standard.

The next two rules essentially describe a fixed point computation where we take the join of existing values with a new value. Specifically, in rule (6), we deal with API calls that load a view from the XML file. In particular, suppose that we have determined that...
Table 2: GUI policies that we formalized as Vesper specifications.

<table>
<thead>
<tr>
<th>Category</th>
<th>Total</th>
<th>Description &amp; Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraudulent</td>
<td>8</td>
<td>Violation of policy often indicates ad fraud e.g. the size ratio between the ad and the screen is required to be greater than a minimum threshold (0.2) [8]</td>
</tr>
<tr>
<td>Unwanted</td>
<td>3</td>
<td>Violation of policy considered annoying/aggressive e.g. activities that display full-screen ads should call the preload function of the ad when they are created. [16]</td>
</tr>
<tr>
<td>Appearance</td>
<td>4</td>
<td>Guidelines about the appearance / spacing of GUI elements e.g. the smallest recommended font size is 10sp [21]</td>
</tr>
<tr>
<td>GDPR Consent</td>
<td>2</td>
<td>GDPR laws about acquiring user consent e.g. applications that display personalized ads should get user consent when they are started [20]</td>
</tr>
</tbody>
</table>

```
node(o, r) ⇐ pointsTo(_, _, o, o), hasType(o, r), r <: View.
rootView(o, o') ⇐ setContentView(I, m, o', i, Nx), inCtx(m, c),
pointsTo(c, l, u, o), pointsTo(c, l, o', o').
entryView(o) ⇐ mainAct(A), instanceof(o, A),
rootView(o', o).

bAttrib(o, X, m) ⇐ node(o, _), setOnClickListener(I, u, m),
inCtx(m, c), pointsTo(I, l, o, o).
sAttrib(o, a, val) ⇐ loadView(I, m, o, N),
inCtx(m, c), pointsTo(c, l, a, o), a = Dom(Ψ(N)),
inCtx(o, a, a'), val' = val ∪ a (vala).
sAttrib(o, a, val') ⇐ setAttrib(I, m, o, a', i, cm, c),
pointsTo(c, l, o, o), val' = val ∪ a (vala).
sEdge(o, o') ⇐ loadView(I, m, o, N),
inCtx(m, c), pointsTo(c, l, o, o'), o' ∈ Ψ(N) (subview).
sEdge(o1, o2) ⇐ addView(I, m, o1, o2), inCtx(m, c),
pointsTo(c, l, o1, o2), pointsTo(c, l, o2, o2).
bEdge(o1, X, o2) ⇐ bAttrib(o1, X, m), inCtx(m, c), call'(c, m, m'),
inCtx(m', c'), showWindow(I, m', o),
pointsTo(c', l, o2).
bEdge(o1, X, o2) ⇐ bAttrib(o1, X, m), inCtx(m, c), call'(c', m, m'),
inCtx(m', c), inc(I, m', l), pointsTo(c, l, o).
```

Figure 10: Datalog-style inference rules describing Elf construction. Here, ⊔ is an abstraction function for the underlying abstract domain, and ⊔ is the corresponding join operator. Ψ refers to the layout schema from Section 5.2.

```
node(o, r) ⇐ pointsTo(_, _, o, o), hasType(o, r), r <: View.
rootView(o, o') ⇐ setContentView(I, m, o', i, Nx), inCtx(m, c),
pointsTo(c, l, u, o), pointsTo(c, l, o', o').
entryView(o) ⇐ mainAct(A), instanceof(o, A),
rootView(o', o).

bAttrib(o, X, m) ⇐ node(o, _), setOnClickListener(I, u, m),
inCtx(m, c), pointsTo(I, l, o, o).
sAttrib(o, a, val) ⇐ loadView(I, m, o, N),
inCtx(m, c), pointsTo(c, l, a, o), a = Dom(Ψ(N)),
inCtx(o, a, a'), val' = val ∪ a (vala).
sAttrib(o, a, val') ⇐ setAttrib(I, m, o, a', i, cm, c),
pointsTo(c, l, o, o), val' = val ∪ a (vala).
sEdge(o, o') ⇐ loadView(I, m, o, N),
inCtx(m, c), pointsTo(c, l, o, o'), o' ∈ Ψ(N) (subview).
sEdge(o1, o2) ⇐ addView(I, m, o1, o2), inCtx(m, c),
pointsTo(c, l, o1, o2), pointsTo(c, l, o2, o2).
bEdge(o1, X, o2) ⇐ bAttrib(o1, X, m), inCtx(m, c), call'(c, m, m'),
inCtx(m', c'), showWindow(I, m', o),
pointsTo(c', l, o2).
bEdge(o1, X, o2) ⇐ bAttrib(o1, X, m), inCtx(m, c), call'(c', m, m'),
inCtx(m', c), inc(I, m', l), pointsTo(c, l, o).
```

attribute a of layout name N can have default value c according to the analysis from Section 5.2. Now, if we encounter an API call that loads layout N into variable v, we first look-up the points-to target o of v and add c to the set of possible values of o.a by taking the join with the old abstract value with c.

Next, rule (7) deals with spatial attributes that are set programatically via an API call. We represent such API calls using the source code predicate setAttrib(l, m, o, a, a') indicating that attribute a of variable v is set to variable a' at program location l inside method m. To update the Elf abstraction, we first look up the abstract value a of variable a' at program location l in some calling context c of method m. If o points to heap object o at the same program location l and calling context c, we then update o.a to be the join of a and o.a's old abstract value. Our implementation uses the interval abstract domain for numeric attributes and the so-called bounded set abstraction for strings [7, 31]

Spatial edges. The next two rules, (8) and (9), describe the introduction of spatial edges due to loading views from the XML file and programmatically adding sub-views respectively. Since these rules are very similar, we only focus on (9). Consider an API call for adding view v2 as a sub-view of v1 at program point l in method m. If o, o' point to heap objects o, o' at program location l in the same calling context c of method m, we introduce a spatial edge from o to o' in the Elf abstraction. In general, o, o' can have multiple points-to targets; thus, this rule can end up introducing multiple spatial edges for the same source code statement.

Behavioral edges. The last two rules, (10) and (11), deal with the introduction of behavioral edges. Recall that a behavioral edge indicates that GUI element e launches GUI element e' upon event X. In general, o can launch o' in one of two ways: The handler of event X (transitive) calls a method that (a) either directly displays o' by calling an API (e.g., showWindow) or (b) indirectly displays o' by performing inter-component communication via an intent object whose target has root view o'. In figure 10, rule (10) deals with case (a), and rule (11) deals with case (b). Since both of these rules rely on knowing the handler method for event X, the body of the rule matches the bAttrib predicate computed by the other rules.

5.4 Checking Conformance

Once Venus generates the Elf abstraction, it translates attributes and edges in the Elf abstraction to ground built-in predicates in the Vesper specification language in the expected way. For instance, the spatial edge (o, o') in the Elf corresponds to the predicate contains(o, o') in the Vesper DSL. Similarly, a behavioral edge (o, e, o') corresponds to the Vesper predicate showWindow(o, e, o') if o' is another window and, for instance, to startBrowser(e, o) if o' is the browser. Thus, Venus can directly convert the Elf abstraction to a formula F that is a conjunction of ground predicates.

Next, to decide whether the input program P entails specification ψ, Venus checks whether F implies ψ. To do so, Venus first converts ψ to a logical formula ψ using the [ ] function defined in Appendix A and then checks the satisfiability of the formula
\( F \land \neg \phi \) using a Datalog solver. If this formula is satisfiable, the specification is violated under the computed ElF abstraction, and \textsc{Venus} produces a model of \( F \land \neg \phi \) as a potential counterexample. On the other hand, the unsatisfiability of \( F \land \neg \phi \) constitutes a proof of conformance since \( F \) over-approximates the app’s relevant behavior with respect to the \textsc{Vesper} specification language.

6 IMPLEMENTATION AND EVALUATION

We implemented our core static analysis on top of the Soot framework [42] and the IC3 tool for Android [34]. We use the SPARK framework [27] provided by Soot to perform pointer analysis and construct a call-graph. Our implementation uses the interval abstract domain for reasoning about numeric attributes and the bounded-set abstraction for strings. \textsc{Venus} also leverages the Souflé [41] Datalog solver for checking conformance between the ELF abstraction and the \textsc{Vesper} specification. As described in Section 5, our analysis is context-sensitive and uses the call site representation proposed in [35]. \textsc{Venus} is openly available on Github. \(^1\)

Experimental set-up. All of our experiments are conducted on a shared 48-core server with Intel Xeon E7-8850 CPU and 500G memory, running the CentOS 7.6 operating system.

6.1 Benchmarks

To evaluate \textsc{Venus}, we collected 2361 Android applications from three different sources:

- **Google Play**: We collected 1488 popular applications that were available on the Google Play Store in Jan 2019.

- **GPP benchmarks**: The Google Play Protect (GPP) team provided us with a labeled data set consisting of 773 Android apps and their label (benign or type of malware). All of these applications were flagged as potential malware by Google’s internal tools and manually audited by Google security analysts.

- **AdFraudBench**: We also evaluate our approach on a dataset taken for detecting ad fraud [8]. This dataset includes 57 ad fraud samples and 43 benign applications.

6.2 Properties

To evaluate \textsc{Venus}, we collected a total of 49 representative GUI policies from Google Play Ads Policy [19], AdMob Help [14], Material Design [18], and EU General Data Protection Regulation [9]. Among those 49 policies, 25 are too vague to formalize (e.g., “Ensure that none of the ad attributes look like navigation features within the app.”). Among the remaining 24, seven of them cannot be expressed in \textsc{Vesper} (e.g., require temporal logic). This leaves us with a total of 17 policies that we formalized in \textsc{Vesper}. To give the reader some intuition, table 2 shows a categorization of these policies and provides some examples of the types of policies we formalized. (See the Appendix for their \textsc{Vesper} formalizations.)

6.3 Results on Google Play Dataset

We evaluated \textsc{Venus} on the 1488 Google Play apps by checking conformance against all 17 policies summarized in table 2. As shown in the first row of table 3, \textsc{Venus} reports a total of 1645 violations across 711 apps, with an average running time of 465.3 seconds per app. Among the 1645 reports, 1258 reports pertain to violations of ad-related policies, 127 reports concern GDPR regulations, and the remaining 260 reports pertain to Material design guidelines.

Manual inspection. Since there is no ground truth label for the apps in the Google Play dataset, we manually inspected 50 of the 711 apps for which \textsc{Venus} reports at least one violation. For these 50 apps, \textsc{Venus} reports a total of 195 warnings. We now report on the findings from our manual inspection.

- **GDPR violations**: Among the 50 apps we inspected, \textsc{Venus} reports a total of 18 GDPR violations, and we manually confirmed that 16 of them indeed access private user information without ever displaying a user consent form.

- **Ad-fraud**: Across the 50 manually inspected apps, \textsc{Venus} reports 40 of them to violate an ad-related property. In particular, 37 of these are true positives, and 11 are previously unknown ad fraud instances (confirmed by Google security auditors).

- **Design guidelines**: \textsc{Venus} reports 24 of the 50 apps to violate a Material design guideline-related property, and 18 of these indeed violate the design guidelines we encoded.

False positive analysis. Among all 50 sampled apps, \textsc{Venus} reported 195 violations, of which 174 are true positives. Based on our manual inspection, most of the false positives are due to imprecision in the pointer analysis. Using the estimation of proportion method [24], we conclude that it is 95% likely that the false positive rate for the whole dataset is between 4% and 18%.

**Result #1**: Among the 50 apps we manually inspected, \textsc{Venus} identified 11 previously unknown ad fraud instances (confirmed) and 16 Google Play apps that violate GDPR regulations. Furthermore, \textsc{Venus}'s false positive rate for the inspected apps is around 10%.

6.4 Results on GPP Dataset

The GPP dataset consists of 773 apps where each app is either labeled as benign or malicious. If the app is malicious, the label also indicates the type of malware (e.g., ad fraud, spyware). For this dataset, we used \textsc{Venus} to detect ad fraud instances by checking...
Comparison against VirusTotal. To put these results in context, we compare Venus’s results with those of VirusTotal [43], which is a widely-used service for detecting several types of malware. VirusTotal uses more than sixty state-of-the-art malware detection engines to analyze an app and shows the aggregate results.

Since VirusTotal does not report a single result and covers a broader class of malware than just ad fraud, there is no “obviously right” way to compare against it for the purposes of our evaluation. Thus, we consider two different, but equally plausible, ways of interpreting VirusTotal’s results:

- **VirusTotal-a**: As in prior work on ad fraud detection [8], we consider VirusTotal to classify an app as ad fraud if at least two of its underlying malware detection engines label it as ad fraud.
- **VirusTotal-b**: Since the security community typically uses VirusTotal as a binary classifier [5], we consider an app to be ad fraud if at least two of the underlying malware detectors label the app as not benign.

The results of our comparison are shown in figure 11. Here, blue bars (with “\"" pattern) show recall, whereas dark magenta bars (with ”" pattern) indicate precision. As we can see from this bar chart, both variants of VirusTotal yield much lower recall and precision compared to Venus.

Analysis of false positives and negatives. We manually inspected the apps that are incorrectly classified by Venus to better understand the root causes of false positives and false negatives. Most of the false positives are caused by imprecision in the pointer analysis (e.g., additional spurious methods are identified as event handlers). On the other hand, false negatives are mainly caused by foreign binary code that our static analyzer cannot reason about. For instance, the “Casino Classic” app from the GPP dataset employs the Unity framework that contains code in the Common Intermediate Language (CIL) binary format. Since our tool cannot analyze CIL binary, it fails to understand some ad-related functionality, and this leads to false negatives.

<table>
<thead>
<tr>
<th></th>
<th># apps</th>
<th># violating apps</th>
<th># violations</th>
<th>recall</th>
<th>precision</th>
<th>avg. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Play</td>
<td>1488</td>
<td>711</td>
<td>1645</td>
<td>N/A</td>
<td>89.2%</td>
<td>465.3</td>
</tr>
<tr>
<td>GPP</td>
<td>773</td>
<td>243</td>
<td>391</td>
<td>86.8%</td>
<td>94.7%</td>
<td>464.7</td>
</tr>
<tr>
<td>AdFraudBench</td>
<td>100</td>
<td>54</td>
<td>90</td>
<td>91.2%</td>
<td>96.3%</td>
<td>302.1</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td><strong>2361</strong></td>
<td><strong>1008</strong></td>
<td><strong>2126</strong></td>
<td>N/A</td>
<td><strong>91.3%</strong></td>
<td><strong>458.2</strong></td>
</tr>
</tbody>
</table>

Table 4: Results on AdFraudBench

<table>
<thead>
<tr>
<th></th>
<th>Venus</th>
<th>FraudDroid</th>
<th>VirusTotal-a</th>
<th>VirusTotal-b</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>precision</strong></td>
<td>96.3%</td>
<td>91.8%</td>
<td>79.6%</td>
<td>75.0%</td>
</tr>
<tr>
<td><strong>recall</strong></td>
<td>91.2%</td>
<td>78.9%</td>
<td>75.4%</td>
<td>89.5%</td>
</tr>
</tbody>
</table>

6.5 Results on AdFraudBench Dataset

In our next experiment, we evaluate Venus on the AdFraudBench dataset used in prior work [8]. Since this data set is specifically targeted for ad fraud detection, we check these apps against the eight ad-fraud-related policies formalized in Vesper. As shown in table 3, Venus has a precision of 96.3% and recall 91.2% on this dataset, and its average running time per app is 302.1 seconds.

To put these results in context, we also compare Venus’s results against those of VirusTotal as well as FraudDroid, which is a dynamic analysis tool specifically for detecting ad fraud [8].

The results of this comparison are shown in table 4, which shows that Venus outperforms VirusTotal and FraudDroid both in terms of precision and recall.

**Result #3:** Venus outperforms FraudDroid (a dynamic analysis tool for ad fraud detection) significantly in terms of recall, while also attaining better precision.

6.6 Evaluation of the Elf Abstraction

In our final experiment, we evaluate the benefits of our proposed Elf abstraction by performing ablation studies and comparing it against the window transition graph (WTG) abstraction proposed in prior work [46].

*WTG abstraction.* As mentioned earlier, the WTG abstraction from the Gator tool [39] is somewhat similar to Elf in that it is a graph abstraction of Android applications where nodes are windows, and edges (annotated with events) represent communication between them. However, WTG differs from our proposed Elf abstraction in two important ways: First, nodes in a WTG correspond to main windows of activities, so it does not contain nodes for any nested GUI elements. Second, a WTG does not contain any information about spatial attributes of windows. To use the WTG abstraction to check Vesper specifications, we use the following

---

1 Recall that all applications in these datasets are either benign or ad fraud.
2 Recall that all applications in these datasets are either benign or ad fraud.
methodology: First, since WTG only contains main windows of activities, we consider any GUI element mentioned in the VESPER specification but not in the WTG as being non-existent in the app. However, this may result in GATOR reporting false negatives. Second, since a WTG does not contain any information about spatial attributes, we consider the abstract value of any spatial attribute to be $\top$, which can result in false positives. Thus, in principle, using GATOR to check for VESPER specifications can suffer from both false positives as well as false negatives.

**Ablations of ELF.** In this evaluation, we also compare our proposed ELF abstraction against two of its own ablations. Since one of our claims is that many GUI policies require reasoning about both spatial and behavioral properties in practice, we consider the following two ablations of ELF:

- **Venus $^{-S}$:** This is a variant of Venus that does not contain spatial attributes. In other words, we do not perform abstract interpretation to reason about values of spatial attributes such as height, size etc., and simply map all of them to $\top$.

- **Venus $^{-B}$:** This is a variant of Venus that does not contain any behavioral edges or attributes. In particular, we do not reason about event handlers of GUI elements (i.e., behavioral attributes), and we also do not reason about communication between different GUI elements (i.e., behavioral edges).

At first glance, it might seem that Venus $^{-S}$ should have only false positives whereas Venus $^{-B}$ would suffer from only false negatives. However, since VESPER predicates may appear negated in the specification, in principle, Venus $^{-S}$ and Venus $^{-B}$ can have both false negatives and false positives.

Table 5 presents the results of our evaluation of the ELF abstraction by comparing it against WTG, Venus $^{-S}$, and Venus $^{-B}$ on both the GPP and AdFraudBench datasets for which we know the ground truth. Our first observation is that GATOR has high precision but very poor recall. While the poor recall is perhaps expected, the high precision is surprising since we treat spatial attributes as $\top$ when using the WTG abstraction to check VESPER policies. However, the reason for this is that GATOR reports a grand total of 3 violations (among the 258 actual violations) in the GPP dataset, and all of these three reports turn out to be real violations. However, the recall is extremely poor, resulting in F1-scores of 0.024 and 0.387 on the GPP and AdFraudBench datasets compared to that of 0.906 and 0.937 of Venus.

Next, we compare Venus against its two ablations. While the recall of both ablations are significantly higher than the WTG abstraction, the overall F1-scores of substantially worse than Venus.

<table>
<thead>
<tr>
<th>Tool</th>
<th>GPP</th>
<th></th>
<th></th>
<th>AdFraudBench</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec.</td>
<td>Recall</td>
<td>F-1</td>
<td>Prec.</td>
<td>Recall</td>
<td>F-1</td>
</tr>
<tr>
<td>GATOR</td>
<td>100.0%</td>
<td>1.2%</td>
<td>0.024</td>
<td>92.3%</td>
<td>24.5%</td>
<td>0.387</td>
</tr>
<tr>
<td>Venus $^{-S}$</td>
<td>53.8%</td>
<td>85.2%</td>
<td>0.660</td>
<td>93.8%</td>
<td>84.6%</td>
<td>0.727</td>
</tr>
<tr>
<td>Venus $^{-B}$</td>
<td>69.0%</td>
<td>80.9%</td>
<td>0.745</td>
<td>79.6%</td>
<td>75.0%</td>
<td>0.772</td>
</tr>
<tr>
<td>Venus</td>
<td>94.7%</td>
<td>86.8%</td>
<td>0.906</td>
<td>96.3%</td>
<td>91.2%</td>
<td>0.937</td>
</tr>
</tbody>
</table>

These results indicate that our proposed ELF abstraction is highly beneficial for checking apps against GUI policies.

**Result #4:** Our proposed ELF abstraction significantly outperforms the WTG abstraction in terms of recall, and it also outperforms its own ablations in terms of F1-score.

7 RELATED WORK

7.1 Program Analysis for User Interfaces

**GUI analysis for mobile apps.** In the space of GUI analysis tools of mobile apps, the most related one is GATOR [39], which statically analyzes Android applications to build models of their GUI-related behavior. These models include so-called constraint graphs [40] and (more related to this work) window transition graphs [46]. However, as shown in Section 6.6, the models produced by GATOR do not provide sufficient information to check an app against VESPER specifications. Another static analyzer that is related to this work is the BackStage tool [25] for identifying which sensitive API functions can be invoked through which UI elements. BackStage checks for specific unintended behaviors of GUI elements, such as leaking a user’s location when she clicks the “upload picture” button. In contrast to BackStage, Venus supports a general class of policies expressed in the VESPER policy language and also reasons about spatial properties of GUI elements as well as communication patterns between them.

There are also some GUI-related analysis tools based on dynamic techniques. For instance, Corridroid [29] tests an application against a set of UI constraints given by the user. As another example, GVT [32] dynamically checks whether the user interface of a mobile app is implemented according to its design mock-up by monitoring its visual appearance. Similarly, REMAUI [33] can automatically identify certain types of UI elements (e.g., images and text) using optical character recognition (OCR) and computer vision techniques. Compared to these dynamic techniques, static techniques like Venus provide complementary advantages such as higher coverage for behaviors that are hard to trigger at run-time.

**GUI analysis for web applications.** Beyond mobile applications, GUI analysis has also attracted some interest in the context of web applications. For example, Cilla [30] finds unused CSS selectors by dynamically monitoring the relationship between CSS rules and webpage elements selected by those rules. Another related work in this space is the Cassius framework [36, 37] for building semantics-aware CSS tools. Specifically, Cassius formalizes the semantics of CSS in first-order logic and can be used to check spatial properties of GUI elements displayed on a webpage. However, since the user interface of web applications is rendered exclusively based on declarative HTML and CSS code, Cassius does not need to analyze JavaScript programs. In contrast, checking an Android application against a VESPER specification requires both precise reasoning about Java code as well as the declarative layout definitions provided in XML files. Besides Cassius, there are other tools specifically built for addressing accessibility problems in web pages [38, 44]. Compared to these tools that are typically based on dynamic testing, Venus has the potential to cover code that is hard to reach by dynamic
analysis. Furthermore, accessibility tools can only check spatial properties of GUI elements while VENUS reasons about both spatial and behavioral properties.

7.2 Static Analysis of Android Applications

Due to the popularity and security-critical nature of Android applications, there is a rich literature of program analysis techniques for the Android framework [4, 6, 11, 22, 23, 28, 34, 47]. A key challenge in statically analyzing Android applications is reasoning about dependencies between different components, such as activities and services. Thus, several papers focus on inter-component communication (ICC) analysis for Android [11, 34]. In this work, we leverage the ICC analysis techniques proposed in prior research.

Among techniques for analyzing Android applications, a particularly relevant work is the Apposcopy system for malware detection [11]. Similar to VENUS, Apposcopy provides a specification language for describing semantic behaviors of Android apps and allows statically checking an app against such a specification. However, the specification language of Apposcopy is tailored for spyware detection and does not allow referring to GUI elements. Thus, beyond ICC analysis, the underlying static analyses performed by Apposcopy and VENUS are quite different.

7.3 Android Malware Detection

Since one of the use cases for VENUS is to detect ad fraud, VENUS is also related to a long line of work on Android malware detection [3, 8, 11, 12, 26, 45]. Most malware detection tools in this space focus on information leakage [11, 28], rather than GUI-related behavior and are therefore not suitable for accurately detecting ad fraud applications. As mentioned earlier, the most relevant work in this space is the FraudDroid tool [8] for detecting malware in the ad fraud category. However, unlike VENUS, FraudDroid is based on dynamic analysis, and, as demonstrated in section 6.5, it has significantly worse recall compared to VENUS.

8 CONCLUSION

We introduced a new framework called VENUS for checking conformance between Android apps and GUI policies expressed in a policy language called VESPER. We manually studied GUI policies from multiple different sources and, among English policies that are precise enough to be formalized, we showed that around 70% are expressible in the VESPER policy language. We used VENUS to check conformance between these policies and over 2000 Android applications and showed that VENUS can uncover previously unknown ad fraud instances as well as violations of GDPR regulations. Our comparison against VirusTotal and FraudDroid indicates that VENUS advances the state-of-the-art in ad fraud detection in terms of both precision and recall. Finally, our comparison against GATOR as well as the two ablation studies highlight the benefits of our proposed ELF abstraction.

9 ACKNOWLEDGEMENTS

This work was partially supported by NSF Grants #1908494, #1908304, CCF-#2005889, and CNS-#1822251 as well as a Google Faculty Research Award.

REFERENCES


[22] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gao, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications, there is a rich literature of program analysis techniques for the Android framework [4, 6, 11, 22, 23, 28, 34, 47]. A key challenge in statically analyzing Android applications is reasoning about dependencies between different components, such as activities and services. Thus, several papers focus on inter-component communication (ICC) analysis for Android [11, 34]. In this work, we leverage the ICC analysis techniques proposed in prior research. Among techniques for analyzing Android applications, a particularly relevant work is the Apposcopy system for malware detection [11]. Similar to VENUS, Apposcopy provides a specification language for describing semantic behaviors of Android apps and allows statically checking an app against such a specification. However, the specification language of Apposcopy is tailored for spyware detection and does not allow referring to GUI elements. Thus, beyond ICC analysis, the underlying static analyses performed by Apposcopy and VENUS are quite different. Android Malware Detection Since one of the use cases for VENUS is to detect ad fraud, VENUS is also related to a long line of work on Android malware detection [3, 8, 11, 12, 26, 45]. Most malware detection tools in this space focus on information leakage [11, 28], rather than GUI-related behavior and are therefore not suitable for accurately detecting ad fraud applications. As mentioned earlier, the most relevant work in this space is the FraudDroid tool [8] for detecting malware in the ad fraud category. However, unlike VENUS, FraudDroid is based on dynamic analysis, and, as demonstrated in section 6.5, it has significantly worse recall compared to VENUS. 8 CONCLUSION We introduced a new framework called VENUS for checking conformance between Android apps and GUI policies expressed in a policy language called VESPER. We manually studied GUI policies from multiple different sources and, among English policies that are precise enough to be formalized, we showed that around 70% are expressible in the VESPER policy language. We used VENUS to check conformance between these policies and over 2000 Android applications and showed that VENUS can uncover previously unknown ad fraud instances as well as violations of GDPR regulations. Our comparison against VirusTotal and FraudDroid indicates that VENUS advances the state-of-the-art in ad fraud detection in terms of both precision and recall. Finally, our comparison against GATOR as well as the two ablation studies highlight the benefits of our proposed ELF abstraction. 9 ACKNOWLEDGEMENTS This work was partially supported by NSF Grants #1908494, #1908304, CCF-#2005889, and CNS-#1822251 as well as a Google Faculty Research Award.