Checking Conformance of Applications against GUI Policies

Zhen Zhang
University of Washington
Seattle, USA
zgzhen@cs.washington.edu

Yu Feng
University of California, Santa Barbara
Santa Barbara, USA
yufeng@cs.ucsb.edu

Michael D. Ernst
University of Washington
Seattle, USA
mernst@cs.washington.edu

Sebastian Porst
Google
Mountain View, USA
sporst@google.com

Isil Dillig
University of Texas Austin
Austin, USA
isil@cs.utexas.edu

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

ACM Reference Format:
Zhen Zhang, Yu Feng, Michael D. Ernst, Sebastian Porst, and Isil Dillig. 2021. Checking Conformance of Applications against GUI Policies. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468561

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468561

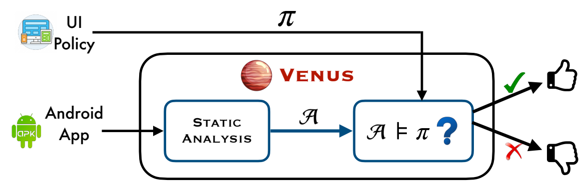

Figure 1: Architecture of Venus. $\mathcal{A}$ **is an event-driven layout forest** (Elf) **(defined in section 3.2) and** π **is a GUI policy written in** Vesper **language (Section 4).**

1 INTRODUCTION

Good graphical user interfaces (GUIs) are essential for the success and popularity of mobile applications. A bad user interface can significantly degrade the user's overall experience, causing the app to become unpopular even if it provides otherwise useful functionality. Beyond leading to poor user experience, bad GUI designs can indicate malicious intent — for example, many ad fraud applications provide a misleading user interface to trick their users into clicking on unwanted links. Such behavior violates one of the advertisement policies published by mobile platforms [1, 14, 19], and according to a recent report [13], click fraud (a major type of ad policy violation) accounts for more than 50% of all potentially harmful applications. Furthermore, several companies and governmental agencies have others types of policies concerning the user interface of mobile apps. For instance, both Google and Apple publish UI design guidelines [2, 15], and the European Union's General Data Protection Regulation (GDPR) laws [10, 20] impose restrictions on how mobile apps may interact with users via their user interfaces.

Despite the increasing importance of ensuring compliance between GUI policies and mobile applications, there are no existing techniques that can be used to check whether an app conforms to such GUI policies. This work aims to address this problem by proposing a new technique, and its implementation in a tool called Venus (figure 1), for checking conformance between a mobile application written in the Android framework and a GUI policy. We envision such a tool being utilized in two different ways: First, Venus can be used by developers to ensure that their user interface is consistent with existing policies, thereby improving overall user experience and ensuring compliance with applicable laws. Second, Venus can be used by security analysts to detect ad fraud applications that trick users through a misleading user interface.

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 nodes 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).

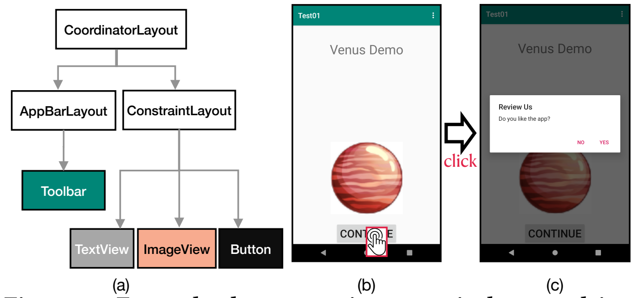

Figure 2: Example demonstrating a typical event-driven flow in Android apps. Listing 1 defines the layout shown in (a) and (b). The transition from (b) to (c) is defined in listing 2.

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

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

```
class MainActivity extends Activity {
  void onCreate(...) {
    ...
    setContentView(R.layout.activity_main);
    TextView demoTitle = findViewById(R.id.demo_title);
    demoTitle.setText("Venus Demo");
    Button continueButton = findViewById(R.id.continue_button);
    continueButton.setOnClickListener(new View.OnClickListener() {
      void onClick(View v) {
        AlertDialog d = new d.Builder(...).create(); ...
        d.setButton(DialogInterface.BUTTON_POSITIVE,
            "YES", new DialogInterface.OnClickListener() {
          void onClick(...) { d.dismiss(); }
        });
        d.show(); } });
  } ...
```

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 `ViewGroups`. 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 `ViewGroups`. Android provides different types of `ViewGroups`, such a `LinearLayout` 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 `ViewGroup` elements and all leaves are `View` objects. Each Android GUI element also has a set of attributes that define its properties, including height, width, alignment, position etc.

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 4 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 `assume` 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(v), which evaluates to true if clicking on v shows a new window v' that corresponds to an `adView` GUI element.

(a) Main activity (b) Battery saver activity (c) Untrusted website

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

```
public class MainActivity extends Activity implements
    OnClickListener {
  private Button saverBtn;
  protected void onCreate(...) {
    setContentView(R.layout.activity_main);
    saverBtn = findViewById(R.id.btn_save);
    saverBtn.setOnClickListener(this);
  }
  public void onClick(View view) {
    Intent intent = new Intent(
      this, BatterySaver.class);
    startActivity(intent);
  }
}
```

Listing 3: Main activity

```
public class BatterySaver extends Activity {
  public void onCreate(Bundle bundle) {
    setContentView(R.layout.battery_saver_ad);
    FrameLayout frameLayout = (FrameLayout)
        findViewById(R.id.content);
    View a = new NativeAdViewBuild().f();
    frameLayout.addView(a);
  }
}

class NativeAdViewBuild {
  public View f() {
    View adView = new UnifiedNativeAdView();
    View bgView = findViewById(R.id.bg_view);
    // set a transparent background
    bgView.setOpacity(0);
    bgView.setOnClickListener(this);
    return adView;
  }

  public void onClick(View arg0) {
    // suspicious URL
    loadURL("http://funtest.afatwallet.com");
  }
}
```

Listing 4: Battery saver activity

Finally, the assertion specifies the desired property. Section 4 will present more about Vesper.

Given this Vesper policy and the source code of the Super Cleaner application (shown in listing 3 and listing 4), we next explain how Venus automatically identifies this policy violation.

1. View bg
2. `assume` $(\exists v.(\text{View}(v) \wedge \text{background}(bg, v)))$
3. `let` $\text{popAd}(v) = \exists v'. (\text{showWindow}(v, \texttt{click}, v') \wedge \text{AdView}(v'))$
4. `assert` $(\text{transparent}(bg) \rightarrow \neg\text{popAd}(bg))$

Figure 4: VESPER specification for the policy "Transparent backgrounds should not be clickable".

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 high-level, this abstraction captures all relevant behavior of the app with respect to the VESPER policy language. For example, figure 5 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 5, 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(click)` 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 4 using the following chain of inferences:

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

Therefore, VENUS reports that the Super Cleaner app does not conform to the VESPER policy from Figure 4.

4 VESPER SPECIFICATIONS

As shown in figure 6, a specification in VESPER starts with a set of declarations, is 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 7), the user can also define *custom predicates* through let bindings. For instance, in figure 4, 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 $\{v \mid \Phi\}$.

Expressions. In VESPER, the most basic expressions are variables v, integer constants c, and pre-defined Android events ε such as click or touch. VESPER allows performing arithmetic operations over integers as well as aggregation over sets. For instance, the expression $\text{count}(v)$ 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 7 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 $\text{LView}(v) = \exists x, y.\ (\text{width}(v, x) \wedge x > 100 \wedge \text{height}(v, y) \wedge y > 100)$
let $\text{LAds} = \{v \mid \text{AdView}(v) \wedge \text{contains}(w, v) \wedge \text{LView}(v)\}$
assert $\text{count}(\text{LAds}) \leq 1$

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 $p(\overline{o})$ to be true in an execution ω if it holds on objects $\overline{o}$ at any time during ω. For example, the predicate $\text{startBrowser}(e, v)$ evaluates to true in execution ω if v starts the browser at some point during ω. Given

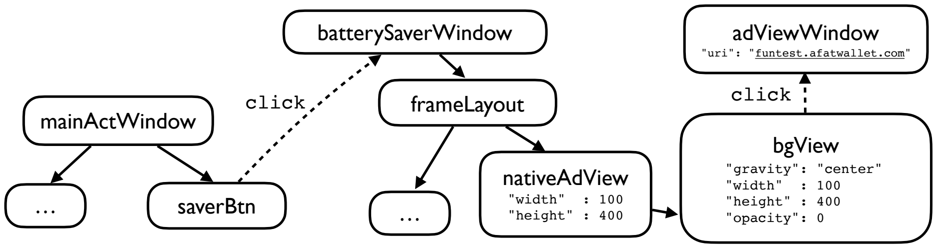

Figure 5: Simplified Elf for motivating example of Sec.3.1. Solid (resp. dashed) lines represent spatial (resp. behavioral) edges.

$$
\begin{array}{rcl}
\text{Policy } \psi & \rightarrow & D;\ S;\ A \\
\text{Decl } D & \rightarrow & \tau\ v \mid D; D \\
\text{Stmt } S & \rightarrow & \texttt{let } v = \{v' \mid \Phi\} \\
 & & \mid \texttt{let } p(\overline{v}) = \Phi \mid \texttt{assume } \Phi \mid S; S \\
\text{Assert } A & \rightarrow & \texttt{assert } \Phi \mid A; A \\
\text{Expr } e & \rightarrow & v \mid \varepsilon \mid c \mid f(e_1, ..., e_n) \\
\text{Pred } \Phi & \rightarrow & p(\overline{e}) \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \\
 & & \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \forall v.\ \Phi \mid \exists v.\ \Phi \\
\text{Event } \varepsilon & \rightarrow & \text{click} \mid \text{longClick} \mid \ldots \mid \text{touch} \\
\text{Type } \tau & \rightarrow & \text{View} \mid \text{Dialog} \mid ... \mid \text{Button} \\
a & \in & \text{Attributes}, \quad c \in \text{Int} \quad f \in \text{Built-in fns} \\
p & \in & \text{Built-in predicates} \cup \text{User-defined predicates}
\end{array}
$$

Figure 6: The Vesper policy language

Element type predicates	
Button(v), Dialog(v), ImageView(v), AdView(v) ...	
Spatial predicates:	
$\text{height}(v, h)$	View v has height h
$\text{width}(v, w)$	View v has width w
$\text{textSize}(v, s)$	Text view v has text size s
$\text{transparent}(v)$	View v is transparent
$\text{contains}(u, v)$	u contains v as a sub-view
$\text{background}(u, v)$	u is the background container of v
Behavioral predicates:	
$\text{entryView}(v)$	v is the top-level window that is displayed when the app starts
$\text{invoke}(u, e, m)$	User event e on GUI element u directly causes invocation of method m
$\text{showWindow}(u, e, v)$	Event e on u results immediately in display of element v
$\text{launchDialog}(w, e, v)$	Window w's event e causes new dialog v to be immediately displayed
$\text{startMarketplace}(e, v)$	Event e on v results immediately in starting a new marketplace window
$\text{startBrowser}(e, v)$	Event e on v results immediately in starting a new browser window

Figure 7: Examples of built-in predicates provided by Vesper

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 Appendix A). Finally, we say that an app A conforms to a Vesper policy P if P evaluates to true in all executions of A.

5 STATIC CONFORMANCE CHECKING

In this section, we introduce the Elf abstraction, describe our static analysis for computing it, and then discuss how to use the Elf abstraction to check conformance against Vesper policies.

5.1 The Elf Abstraction

An event-driven layout forest (Elf) is a tuple $\mathcal{G} = (N, N_0, E, L)$ where:

- N is a set of nodes where each node is a pair $\langle o, \tau \rangle$ representing an abstract heap object o of GUI element type τ.
- Nodes $N_0 \subseteq N$ are *initial nodes* that may correspond to the main window of the application.
- Edges $E = E_S \uplus E_B$ encode relationships between GUI elements. We refer to edges $(n, n') \in E_S$ as *spatial edges* and $(n, \varepsilon, n') \in E_B$ as *behavioral edges*.
- Labeling function $L : N \times \text{Attrib} \rightarrow \text{AbstractVal}$ maps attributes of GUI elements to their abstract values.

As mentioned in section 3, a spatial edge (n, n') encodes that GUI element n is nested within n', whereas a behavioral edge (n, ε, n') indicates that user/system event ε on GUI element n directly results in the display of element of n'. The labeling function L can refer to both spatial and behavioral properties of GUI elements. For example, the height attribute refers to a spatial property of the node, whereas click is a behavioral property that identifies which methods may be invoked upon a click event. In general, since Venus cannot exactly determine the values of node attributes using static analysis, the labeling function L maps these attributes to *abstract* rather than concrete values; however, the choice of abstract domain depends on the type of the attribute (see Section 6).

5.2 XML Analysis for Layout Schema

As mentioned in section 2, GUI elements in Android are typically declared via an XML file and then loaded by the application code at run time. Thus, to facilitate static analysis, Venus encodes the GUI-related information declared in the XML file as a so-called *layout schema*. As shown in figure 8, a layout schema Ψ maps each layout name to its structure, represented as a multi-map from attributes to their type T and *default value* c. Given a layout name N and its definition $\Psi(N) = [a_1 \mapsto (T_1, c_1), a_n \mapsto (T_n, c_n)]$, we write $\text{DefaultVal}(N)$ to indicate an object with fields $a_1, \ldots, a_n$ where each field a_i initialized to c_i.

Example 2. *Consider the following layout XML:*

```
<LinearLayout id="lin" orientation="vertical">
    <TextView id="txt1" width=100 height=200
```

Schema Ψ	:=	LayoutName $\rightarrow \Delta$
Layout Δ	:=	Attrib $\rightarrow (T, c)$
Type T	:=	Int | String | Float | Builtin | LayoutName
Constant c	$\in$	Int $\cup$ String $\cup$ Float $\cup$ DefaultVal(LayoutName) $\cup$ DefaultVal(Builtin)
Builtin	$\in$	{Button, TextView, ...}
Attrib	$\in$	{orientation, subview, ...}

Figure 8: Layout Schema Definition

```
        text="Hello, I am a TextView" />
</LinearLayout>
```

We represent this as the following layout schema:

$$\Psi(\text{lin}) = \{\text{orientation} \mapsto (\text{string}, \text{"vertical"}),$$
$$\text{subview} \mapsto (\text{TextView}, \text{DefaultVal}(\text{txt1}))\}$$
$$\Psi(\text{txt1}) = \{\text{width} \mapsto (\text{Int}, 100), \text{height} \mapsto (\text{Int}, 200), \cdots\}$$

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, v_1, v_2) indicates that there is an API call of the form v_1.addView(v_2) 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, o) indicates that variable v points to heap object o at program location l in calling context c. Similarly, aval(c, l, v, a) 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 o is val, and bEdge(o, ε, o') indicates that there is a behavior edge between o and o' labeled ε.

As mentioned earlier, we present our static analysis (see figure 9) 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 9 in more detail.

Nodes. According to the first rule in figure 9, 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.

Table 1: Predicates used or computed by our analysis

Source code predicates	
loadView(l, m, v, N)	load layout N to v at location l of method m
addView(l, m, v_1, v_2)	v_2 is added as sub-view of v_1 at l in method m
setContentView(l, m, v_1, v_2)	add v_2 as content view of v_1 at l in method m
setAttrib(l, m, v, a, v')	attribute a of v is set to v' at l in method m
set$X$Listener(l, m, v, m')	Method m' is set as v's X listener
showWindow(l, m, v)	Location l has a call to display window v
icc$(l, m, intent)$	Perform ICC using $intent$ at l of method m
mainAct(A)	A is the app's main activity
Pre-analysis predicates	
inCtx(m, c)	c is a calling context of method m
aval(c, l, v, a)	v has abstract value a at location l in context c
pointsTo(c, l, v, o)	v points to object o at location l in context c
pointsTo(c, l, o, f, o')	The f field of o points to o' at l in context c
call$^*(c, m, m')$	m directly or transitively calls m' in context c
hasType(o, τ)	Heap object o has type τ
...	...
Output predicates	
node(o, τ)	o is a GUI element node of type τ in Elf
sAttrib(o, a_s, val)	node o has spatial attribute a_s with value val
bAttrib(o, a_b, val)	node o has behavioral attribute a_b with val
entryView(v)	v is a window shown on app startup
sEdge(o, o')	view o contains view o'
bEdge(o, ε, o')	view o leads to view o' under event ε
rootView(o_1, o_2)	o_1 has root view o_2

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 $setXListener(l, m, v, m')$ 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$(o, X, m_1), \ldots,$ bAttrib(o, X, m_1) 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 $\perp$, as standard.

The next two rules essentially describe a fixed point computation where we take the join of existing values with a new value.

Table 2: GUI policies that we formalized as VESPER specifications.

	Category	Total	Description & Example
Ad-related	Fraudulent	8	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]
	Unwanted	3	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]
Non-Ad	Appearance	4	Guidelines about the appearance / spacing of GUI elements e.g. the smallest recommended font size is 10sp [21]
	GDPR Consent	2	GDPR laws about acquiring user consent e.g. applications that display personalized ads should get user consent when they are started [20]

$$
\begin{aligned}
\mathrm{node}(o,\tau) &\Leftarrow \mathrm{pointsTo}(\_,\_,v,o),\\
&\quad \mathrm{hasType}(o,\tau),\ \tau <: \mathsf{View}. && (1)\\
\mathrm{rootView}(o,o') &\Leftarrow \mathrm{setContentView}(l,m,v,v'),\ \mathrm{inCtx}(m,c)\\
&\quad \mathrm{pointsTo}(c,l,v,o),\ \mathrm{pointsTo}(c,l,v',o'). && (2)\\
\mathrm{entryView}(o) &\Leftarrow \mathrm{mainAct}(A),\ \mathrm{instanceOf}(o,A),\\
&\quad \mathrm{rootView}(o',o). && (3)\\
\mathrm{bAttrib}(o,X,m) &\Leftarrow \mathrm{node}(o,\_),\ \mathrm{set}X\mathrm{Listener}(l,v,m),\\
&\quad \mathrm{inCtx}(m,c),\ \mathrm{pointsTo}(c,l,v,o). && (4)\\
\mathrm{sAttrib}(o,a,\bot) &\Leftarrow \mathrm{node}(o,\mathsf{View}),\ a \in \mathsf{Attribs}(\Psi). && (5)\\
\mathrm{sAttrib}(o,a,val') &\Leftarrow \mathrm{loadView}(l,m,v,N),\ \mathrm{inCtx}(m,c),\\
&\quad \mathrm{pointsTo}(c,l,v,o).\ a \in \mathsf{Dom}(\Psi(N)),\\
&\quad a \neq \mathsf{subview},\ \Psi(N)(a) = (T, val_0)\\
&\quad \mathrm{sAttrib}(o,a,val),\ val' = val \sqcup \alpha(val_0) && (6)\\
\mathrm{sAttrib}(o,a,val'') &\Leftarrow \mathrm{setAttrib}(l,m,v,a,v'),\ \mathrm{inCtx}(m,c),\\
&\quad \mathrm{pointsTo}(c,l,v,o),\ \mathrm{aval}(c,l,v',val'),\\
&\quad \mathrm{sAttrib}(o,a,val),\ val'' = val \sqcup \alpha(val'). && (7)\\
\mathrm{sEdge}(o,o') &\Leftarrow \mathrm{loadView}(l,m,v,N),\ \mathrm{inCtx}(m,c),\\
&\quad \mathrm{pointsTo}(c,l,v,o),\ o' \in \Psi(N)(\mathsf{subview}). && (8)\\
\mathrm{sEdge}(o_1,o_2) &\Leftarrow \mathrm{addView}(l,m,v_1,v_2),\ \mathrm{inCtx}(m,c),\\
&\quad \mathrm{pointsTo}(c,l,v_1,o_1),\ \mathrm{pointsTo}(c,l,v_2,o_2). && (9)\\
\mathrm{bEdge}(o_1,X,o_2) &\Leftarrow \mathrm{bAttrib}(o_1,X,m),\ \mathrm{inCtx}(m,c),\ \mathrm{call}^*(c,m,m'),\\
&\quad \mathrm{inCtx}(m',c'),\ \mathrm{showWindow}(l,m',v),\\
&\quad \mathrm{pointsTo}(c',l,v,o_2). && (10)\\
\mathrm{bEdge}(o_1,X,o_2) &\Leftarrow \mathrm{bAttrib}(o_1,X,m),\ \mathrm{inCtx}(m,c'),\ \mathrm{call}^*(c',m,m'),\\
&\quad \mathrm{inCtx}(m',c),\ \mathrm{icc}(l,m',i),\ \mathrm{pointsTo}(c,l,i,o)\\
&\quad \mathrm{pointsTo}(c,l,o,\text{“tgt”},o'),\ \mathrm{rootView}(o',o_2). && (11)
\end{aligned}
$$

Figure 9: Datalog-style inference rules describing ELF construction. Here, α is an abstraction function for the underlying abstract domain, and $\sqcup$ is the corresponding join operator. Ψ refers to the layout schema from section 5.2.

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 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 programmatically via an API call. We represent such API calls using the source code predicate $setAttrib(l, m, v, a, v')$ indicating that attribute a of variable v is set to variable v' at program location l inside method m. To update the ELF abstraction, we first look up the abstract value a of variable v' at program location l in some calling context c of method m. If v 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 v_2 as a sub-view of v_1 at program point l in method m. If v, v' 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, v, v' 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 o launches GUI element o' upon event X. In general, o can launch o' in one of two ways: The handler of event X (transitively) 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 9, 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 $\mathsf{contains}(o, o')$ in the VESPER DSL. Similarly, a behavioral edge (o, ε, o') corresponds to the VESPER predicate $\mathsf{showWindow}(o, \varepsilon, o')$ if o' is another window and, for instance, to $\mathsf{startBrowser}(\varepsilon, o)$ if o' is the browser. Thus, VENUS can directly convert the ELF abstraction to a formula $\mathcal{F}$ that is a conjunction of ground predicates.

Next, to decide whether the input program $\mathcal{P}$ entails specification ψ, VENUS checks whether $\mathcal{F}$ implies ψ. To do so, VENUS first converts ψ to a logical formula ϕ using the $[\![\cdot]\!]$ function defined in Appendix A and then checks the satisfiability of the formula $\mathcal{F} \wedge \neg\phi$ using a Datalog solver.If this formula is satisfiable, the specification is violated under the computed Elf abstraction, and Venus produces a model of $\mathcal{F} \wedge \neg\phi$ as a potential counterexample. On the other hand, the unsatisfiability of $\mathcal{F} \wedge \neg\phi$ constitutes a proof of conformance since $\mathcal{F}$ over-approximates the app's relevant behavior with respect to the 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. Venus also leverages the Soufflé [41] Datalog solver for checking conformance between the ELF abstraction and the Vesper specification. As described in Section 5, our analysis is context-sensitive and uses the call site representation proposed in [35]. 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 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 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 Vesper (e.g., require temporal logic). This leaves us with a total of 17 policies that we formalized in 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 Vesper formalizations.)

6.3 Results on Google Play Dataset

We evaluated 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, 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.

[1]Details are hidden for double-blind review purpose. The experimental artifact including the tool will be submitted after acceptance.

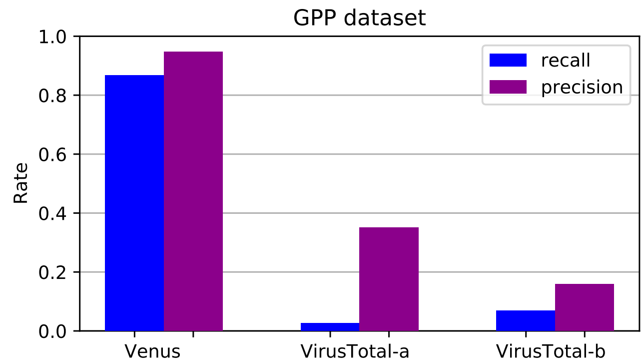

Figure 10: Results on the GPP dataset

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 Venus reports *at least one* violation. For these 50 apps, 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, 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, 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:** 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, 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, Venus identified 11 previously unknown ad fraud instances (confirmed) and 16 Google Play apps that violate GDPR regulations. Furthermore, 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 Venus to detect ad fraud instances by checking

Table 3: Summary of VENUS results across all three datasets

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

conformance between each app and the eight ad-fraud-related policies that we formalized in VESPER. As summarized in the second row of table 3, the recall of VENUS on this dataset is 86.8% and the precision is 94.7%. The average running time is 464.7 seconds.

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 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. [2]

The results of our comparison are shown in figure 10. 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.

> **Result #2:** On 773 apps flagged as potentially malicious by Google's internal tools and manually labeled by security analysts, VENUS has a precision of 94.7% and recall of 86.8%. Furthermore, VENUS outperforms VirusTotal by a factor of 2.7 in terms of precision and by a factor of 12.8 in terms of recall.

[2]Recall that all applications in these datasets are either benign or ad fraud.

Table 4: Results on AdFraudBench

	VENUS	FraudDroid	VirusTotal-a	VirusTotal-b
precision	**96.3%**	91.8%	79.6%	75.0%
recall	**91.2%**	78.9%	75.4%	89.5%

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]. [3] 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. [4]

> **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 *windown 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

[3]FraudDroid is not available, so we cannot evaluate it on GPP apps.

[4]Dong et al. ([8]) report 92% recall on 100 apps from the AdFraudBench instead of the 12000 apps dataset. After we manually inspected the ground truth for those 100 apps, we noticed that FraudDroid actually mislabeled 7 malicious apps as benign. To resolve this discrepancy, we further confirmed our results by uploading those 7 apps to VirusTotal, which also marked those apps as malware. We further contacted the co-authors of FraudDroid and they also agreed that those 7 apps should all be ad fraud. That is why the actual recall is around 80%.

Table 5: Evaluation of our abstraction. Prec. is "Precision"

Tool	GPP			AdFraudBench		
	Prec.	Recall	F-1	Prec.	Recall	F-1
Gator	100.0%	1.2%	0.024	92.3%	24.5%	0.387
Venus^{-S}	53.8%	85.2%	0.660	63.8%	84.6%	0.727
Venus^{-B}	69.0%	80.9%	0.745	79.6%	75.0%	0.772
Venus	94.7%	86.8%	0.906	96.3%	91.2%	0.937

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. [5] Clearly, 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 spacial 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.

[5] Alternatively, we could consider a node to represent all views nested within it; however, this requires doing significant additional analysis that Gator does not perform.

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, Cornidroid [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.

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

[1] Alibaba. 2020. Alibaba UC Market Ads Guide. http://aliapp.open.uc.cn/wiki/?p=140. [Online; accessed 13-Mar-2020].

[2] Apple. 2020. iOS Design. https://developer.apple.com/design/tips. [Online; accessed 13-Mar-2020].

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26. https://doi.org/10.14722/ndss.2014.23247

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[5] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 659–674.

[6] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. 2013. Contextual policy enforcement in android applications with permission event graphs.. In *NDSS*. 234.

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252. https://doi.org/10.1145/512950.512973

[8] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: Automated Ad Fraud Detection for Android Apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/3236024.3236045

[9] EU. 2020. Art. 7 GDPR – Conditions for consent. https://gdpr-info.eu/art-7-gdpr/. [Online; accessed 4-Apr-2020].

[10] EU. 2020. Article 5: Principles relating to processing of personal data. https://www.privacy-regulation.eu/en/article-5-principles-relating-to-processing-of-personal-data-GDPR.htm. [Online; accessed 13-Mar-2020].

[11] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 576–587. https://doi.org/10.1145/2635868.2635869

[12] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. 2017. Automatically learning android malware signatures from few samples. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)(San Diego, California, USA*.

[13] Google. 2018. Android Security & Privacy 2018 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf. [Online; accessed 13-Mar-2020].

[14] Google. 2020. AdMob policies and restrictions. https://support.google.com/admob/answer/6128543?hl=en&ref_topic=2745287&visit_id=637149126866279343-1579955165&rd=1. [Online; accessed 13-Mar-2020].

[15] Google. 2020. Android Developer - Design. https://developer.android.com/design. [Online; accessed 13-Mar-2020].

[16] Google. 2020. Disallowed interstitial implementations. https://support.google.com/admob/answer/6201362?hl=en. [Online; accessed 13-Mar-2020].

[17] Google. 2020. Guidelines for programmatic native ads using app code. https://support.google.com/admanager/answer/7031536?hl=en. [Online; accessed 4-Apr-2020].

[18] Google. 2020. Material Design. https://material.io/. [Online; accessed 13-Mar-2020].

[19] Google. 2020. Play Store Ads Guide. https://play.google.com/intl/en-GB_ALL/about/monetization-ads/ads/. [Online; accessed 13-Mar-2020].

[20] Google. 2020. Requesting Consent from European Users. https://developers.google.com/admob/android/eu-consent. [Online; accessed 13-Mar-2020].

[21] Google. 2020. The type system. https://material.io/design/typography/the-type-system.html. [Online; accessed 4-Apr-2020].

[22] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe. In *NDSS*, Vol. 15. 110. https://doi.org/10.14722/ndss.2015.23089

[23] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133926

[24] Peter J Huber et al. 1972. The 1972 wald lecture robust statistics: A review. *The Annals of Mathematical Statistics* 43, 4 (1972), 1041–1067.

[25] Konstantin Kuznetsov, Vitalii Avdiienko, Alessandra Gorla, and Andreas Zeller. 2018. Analyzing the User Interface of Android Apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 84–87. https://doi.org/10.1145/3197231.3197232

[26] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: Static Analysis Framework for Android Hybrid Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 250–261. https://doi.org/10.1145/2970276.2970368

[27] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.

[28] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291. https://doi.org/10.1109/ICSE.2015.48

[29] Chafik Meniar, Florence Opalvens, and Sylvain Hallé. 2017. Runtime Verification of User Interface Guidelines in Mobile Devices. In *International Conference on Runtime Verification*. Springer, 410–415. https://doi.org/10.1007/978-3-319-67531-2_27

[30] Ali Mesbah and Shabnam Mirshokraie. 2012. Automated analysis of CSS rules to support style maintenance. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 408–418. https://doi.org/10.1109/ICSE.2012.6227174

[31] Ramon E Moore. 1966. *Interval analysis*. Vol. 4. Prentice-Hall Englewood Cliffs.

[32] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated Reporting of GUI Design Violations for Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 165–175. https://doi.org/10.1145/3180155.3180246

[33] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 248–259. https://doi.org/10.1109/ASE.2015.32

[34] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android intercomponent communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 77–88. https://doi.org/10.1109/ICSE.2015.30

[35] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis* (Seattle, Washington) *(SOAP '13)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/2487568.2487569

[36] Pavel Panchekha, Adam Timothy Geller, Shoaib Kamil, Michael Ernst, Zachary Tatlock, and Emina Torlak. 2020. The Cassius Framework. https://cassius.uwplse.org/. [Online; accessed 13-Mar-2020].

[37] Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 181–194. https://doi.org/10.1145/2983990.2984010

[38] Raghavendra Satish Peri. 2021. 18 Free Mobile Accessibility Testing Tools. https://www.digitala11y.com/free-mobile-accessibility-testing-tools/. [Online; accessed 13-Feb-2021].

[39] PRESTO. 2017. GATOR: Program Analysis Toolkit For {Android}. , 12 pages. http://web.cse.ohio-state.edu/presto/software/gator/

[40] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 143–153. https://doi.org/10.1145/2581122.2544159

[41] Soufflé Developers. 2020. Soufflé - Datalog. https://souffle-lang.github.io/datalog. [Online; accessed 13-Mar-2020].

[42] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[43] VirusTotal. 2020. VirusTotal. https://www.virustotal.com/. [Online; accessed 13-Mar-2020].

[44] W3C. 2021. Web Accessibility Evaluation Tools List. https://www.w3.org/WAI/ER/tools/. [Online; accessed 13-Feb-2021].

[45] Guangliang Yang and Jeff Huang. 2018. Automated generation of event-oriented exploits in android hybrid apps. In *Proc. of the Network and Distributed System Security Symposium (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23241

[46] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873. https://doi.org/10.1109/ASE.2015.76

[47] Yifei Zhang, Yulei Sui, and Jingling Xue. 2018. Launch-Mode-Aware Context-Sensitive Activity Transition Analysis. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 598–608. https://doi.org/10.1145/3180155.3180188