SeMA: Extending and Analyzing Storyboards to Develop Secure Android Apps

Joydeep Mitra
Kansas State University, USA
joydeep@ksu.edu

Venkatesh-Prasad Ranganath
rvprasad.free@gmail.com

January 27, 2020

Abstract

As security of mobile apps is crucial to modern-day living, there is a growing need to help developers build apps with provable security guarantees that apps do not leak sensitive user information or cannot be exploited to perform actions without the user's consent. The current prevalent approach to mobile app security is to curatively address vulnerabilities after apps have been developed. This approach has downsides in terms of time, resources, user inconvenience, and information loss.

As an alternative, we propose a design-based mobile app development methodology called SeMA to prevent the creation of vulnerabilities in mobile apps. SeMA enables app designers and developers to iteratively reason about the security of an app by using its storyboard, an existing and prevalent design artifact. A proof of concept realization of SeMA using Android Studio tooling is able to prevent 49 known vulnerabilities that plague Android apps.

1 Introduction

Android apps are ubiquitous. They help us with critical tasks such as banking and communication. Consequently, they need access to our personal information. Hence, developers try to ensure their apps do not cause harm to the user. Despite such efforts, apps exhibit vulnerabilities that can be exploited by malicious apps either installed in the device or executing remotely. For example, in 2019, a vulnerability in Google's camera app allowed a malicious app without required permissions to gain full control of the camera app and access the photos and videos stored by the camera app [5]. A 2019 study [29] showed that approximately 11% of 2,000 Android apps collected from a couple of app stores, including Google Play, are vulnerable to MITM and phishing attacks.

In the last decade, researchers have developed a plethora of tools and techniques to help detect vulnerabilities in Android apps [26]. Even so, apps with vulnerabilities find their way to app stores because app developers do not use these tools, or the tools are ineffective. The latter reason is supported by the findings of Ranganath and Mitra [23]: from a set of 42 previously known vulnerabilities that plague Android apps, fourteen freely available vulnerability tools could collectively detect only 30 vulnerabilities. Additionally, Pauck et al. [21] assessed six prominent taint analysis tools aimed at discovering vulnerabilities in Android and found tools to be ineffective in detecting vulnerabilities in real-world apps.

Existing approaches to secure Android apps are *curative, i.e., detect vulnerabilities after they occur*. Identifying and fixing a vulnerability in a curative manner increases the cost of development [27]. Therefore, given the current landscape of Android app security, we are exploring a preventive approach that can help prevent the occurrence of vulnerabilities in apps (as opposed to curing apps of vulnerabilities).

In this context, we propose a methodology, SeMA, based on an existing mobile app design technique called *storyboarding*. SeMA treats security as a first-class citizen in the design phase and enables analysis and verification of security properties in an app's design. We demonstrate that SeMA can help prevent 49 of the 60 vulnerabilities captured in the Ghera benchmark suite [18], which is more than the vulnerabilities collectively detected by tools evaluated by Ranganath and Mitra [23] (against an earlier version of Ghera).

2 Background

SeMA borrows heavily from Model-Driven Development (MDD) and UX design techniques for Android apps.

Model Driven Development (MDD) In MDD, software is developed by iteratively refining abstract and formal models [3]. A model is meant to capture the application's behavior and is expressed in a domain-specific language (DSL). Apart from models, the domain-specific platform is a crucial entity in MDD. The domain-specific platform provides frameworks and APIs to enable easy refinement of a model into a platform-specific implementation. Since every aspect of an application can seldom be specified in the DSL, the resulting implementation is often extended with additional code; mostly, the business logic of the application.

Today, numerous tools exist to enable MDD in various domains. For example, Amazon uses TLA+ to develop web services [14, 30]. Mendix is a commercial tool that enables MDD for enterprise applications [8]. Tools like Alloy [13] and UML/OCL [24] help create and analyze models of software behavior, which form the basis for further development. Similar tools exist for mobile app development; most of them aid cross-platform development of mobile apps. Brambilla and others [4, 6] extended the Information Flow Modeling Language, a standard for depicting UI behavior, to enable the specification of a mobile app's GUI in a platform-independent manner. Heitkotter et al. [7] developed MD^2 to create cross-platform mobile apps in a high-level DSL. Vaupel et al. [28] developed an approach to model an app's behavior at different abstraction levels.

Storyboarding Android app development teams use storyboarding to design an app's navigation [22, 15]. A storyboard is a sequence of screens and transitions between the screens. It serves as a model for the user's observed behavior in terms of screens and transitions. Numerous tools such as Xcode [1], Sketch [25], and Android's JetPack [11] help express a storyboard digitally. The storyboarding process is *participatory* in nature because it allows designers to get feedback from potential users about the features captured in the storyboard and from developers about the feasibility of implementing those features. However, traditional storyboards cannot capture an app's behavior (beyond UI and navigational possibilities). This limitation of storyboards hinders collaboration.

MDD with Storyboarding Existing MDD approaches to mobile app development do not use storyboards. Extending storyboards with capabilities to capture an app's behavior to enable MDD has numerous benefits. First, storyboards are becoming an integral part of the mobile app development process. Therefore, a methodology based on storyboarding can fit into the mobile app development life cycle. Second, an extended storyboard can serve a common substrate for collaboration between designers and developers to specify an app's behavior along with its UI and navigational features. Third, an extended storyboard can serve as a basis for formal analysis of an app's behavior. The abstractness of the models helps with the analysis because analyzing behaviors captured in an abstract model is relatively easier than extracting and analyzing the behaviors captured in code. Finally, the storyboard can serve as a reference for an app's behavior when auditing the app's implementation.

3 The Methodology

The proposed methodology, SeMA, enables the reasoning and verification of security properties of an app's storyboard via iterative refinement. The development process of SeMA is shown in Figure 1. The process begins with a developer sketching the initial storyboard of an app. The developer then extends the storyboard with the app's behavior and checks if the behaviors satisfy various pre-defined security properties. The developer may repeat the previous steps to revise and refine the behaviors. Once the storyboard has captured the behavior as intended by the developer while satisfying pre-defined security properties, the developer generates an implementation from the storyboard with a push of a button. As the final step, the developer adds business logic to the implementation via hooks provided in the generated code.

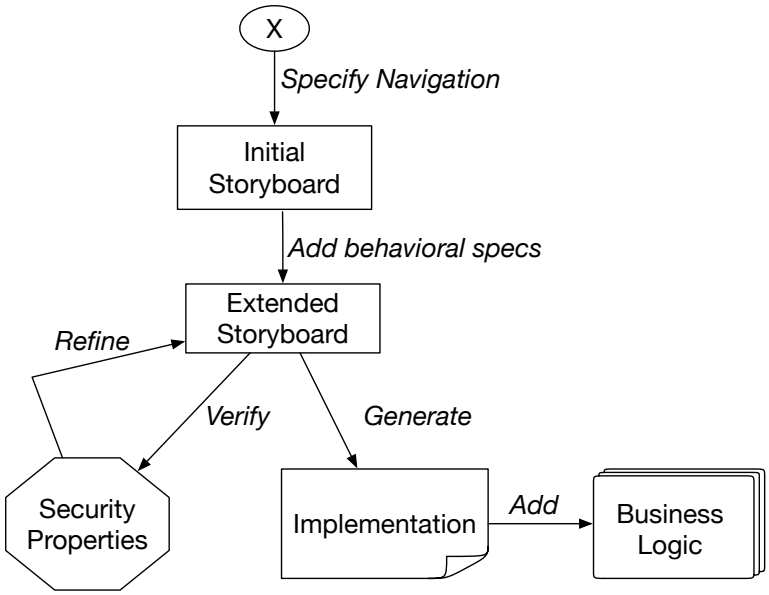


Figure 1: A schematic of the steps in SeMA

3.1 Extended Storyboard

A *traditional storyboard* used in the design of mobile apps is composed of screens and transitions between screens. A screen is a collection of named widgets that allow the user to interact with the app, e.g., clicking a button. A transition (edge) between two screens depicts a navigation path from the source screen to the destination screen. The basic structure of a storyboard defines the navigational paths in an app.

For example, Figure 2 shows the traditional storyboard of an app with four screens: *Messenger*, *Contacts*, *MsgStatus*, and *SaveStatus*. Starting from the *Messenger* screen, a user can either add contact numbers to the app via *Contacts* screen or send a message to all saved contact numbers.

A traditional storyboard does not support the specification of app behavior. Hence, we propose the following extensions to traditional storyboards to enable the specification of app behaviors in storyboards, i.e., enrich a traditional storyboard as in Figure 2 into an extended storyboard as in Figure 3.

App Identity Every storyboard has an *app* attribute, which is a unique string constant used to identify the app described by the storyboard.

Extensions to Screens Screens are extended with a mandatory *name*, optional *input parameters* (in green in Figure 3.1), and optional *URIs* (in purple in Figure 3).

Input parameters of a screen are similar to parameters of a function in mainstream programming. Input parameters bind to the values (arguments) provided when the screen is activated by either another screen in the app via an incoming transition or an app via the screen's *URI*.

A *URI* associated with a screen can be used to access the screen from outside the app. A URI can have input parameters; similar to query parameters in web URLs. URI input parameters serve as input parameters of the screen. All URIs associated with a screen must have the same set of input parameters. For example, both URIs associated with the *Contacts* screen in Figure 3, have the input parameter y. External apps accessing a screen via its URI must provide the arguments corresponding to the URI's input parameters. Every URI without its parameters must be unique in an app.

Proxy Screens of External Apps Apps often interact with external apps. To capture this interaction, *proxy screen* representing a screens of an external app can be included an extended storyboard of an app;

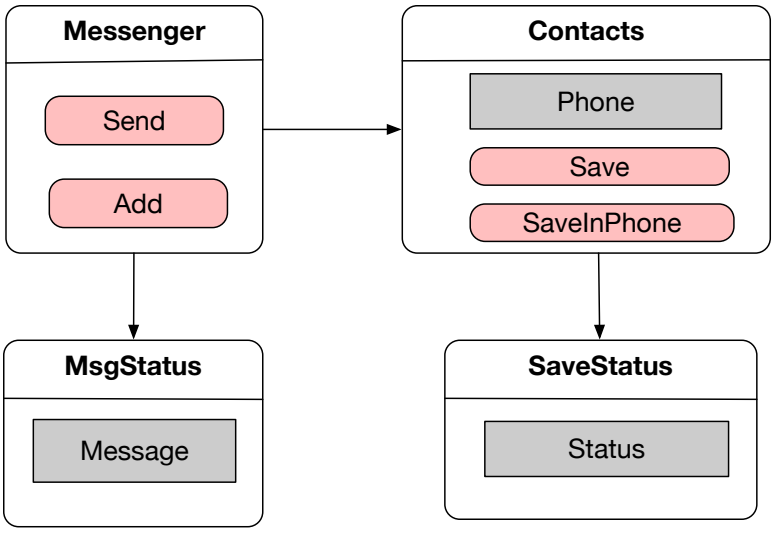


Figure 2: Example of an initial storyboard.

see *PhoneApp* in Figure 3. Such proxy screens have a mandatory *name*, a mandatory *URI*, and an optional *app* attributes. If *app* is specified in a proxy screen, then the proxy screen denotes the screen identified by the *URI* in the app named *app*. If *app* is not specified, then the proxy screen denotes any screen identified by the *URI* in an app installed on the device and determined by Android.

Extensions to Widgets Widgets are extended with a mandatory *value* that can be assigned by the developer (e.g., in labels), entered by the user (e.g., in fields), provided by an input parameter of the containing screen (e.g., when the screen is activated by a transition), or returned by an operation. Based on the displayed content, widgets can be of different types, e.g., a label and text widget display plain text while a web widget displays web content. Further, depending on the widget's type, a widget can be configured with a pre-defined set of rules that regulate the data displayed in a widget, e.g., a *WebView* widget can be configured with a whitelist of trusted URL patterns (via *trusted-patterns* attribute) to display content only from URLs in the whitelist.

Resources Android provides apps with resources with different capabilities, e.g., storage, networking. Hence, to complement this aspect, storyboards are extended with a pre-defined set of resources with specific capabilities that can be used by the apps being designed. [1] Android apps can offer UI-less services to other apps, e.g., broadcast receivers, content providers. Such services are denoted by custom resources in storyboards. A *custom resource* offers capabilities that can be used by apps installed on the device. Each custom resource has a mandatory identifier that is unique to the app. Each capability of a resource has a mandatory identifier that is unique to the resource. Also, each capability that has security implications can be marked as *privileged*. Access to a resource and its capabilities can be controlled via the *access* attribute of the resource. This attribute can take on one of the following three values: *all* implying any app can be accessed the resource/capability, *user* implying user must grant a specific permission to access the resource/capability, and *own* implying only the resource defining app x or an app that shares the digital signature of app x. For example, in Section 3.1, NOTIFICATION_MGR is a custom resource that offers NOTIFY capability with a *notify* operation. Based on its access attribute, a client will need to seek the user's permission to use its NOTIFY capability.

Operations In an extended storyboard, an *operation* indicates a task to be performed, e.g., read from a file, get contents from a web server. An operation has a *name*, returns a value, may have *input parameters*, and may use a *capability* (provided by a resource). An operation is used by mentioning its name along with

[1] The current realization of SeMA supports a subset of pre-defined resources offered by Android.

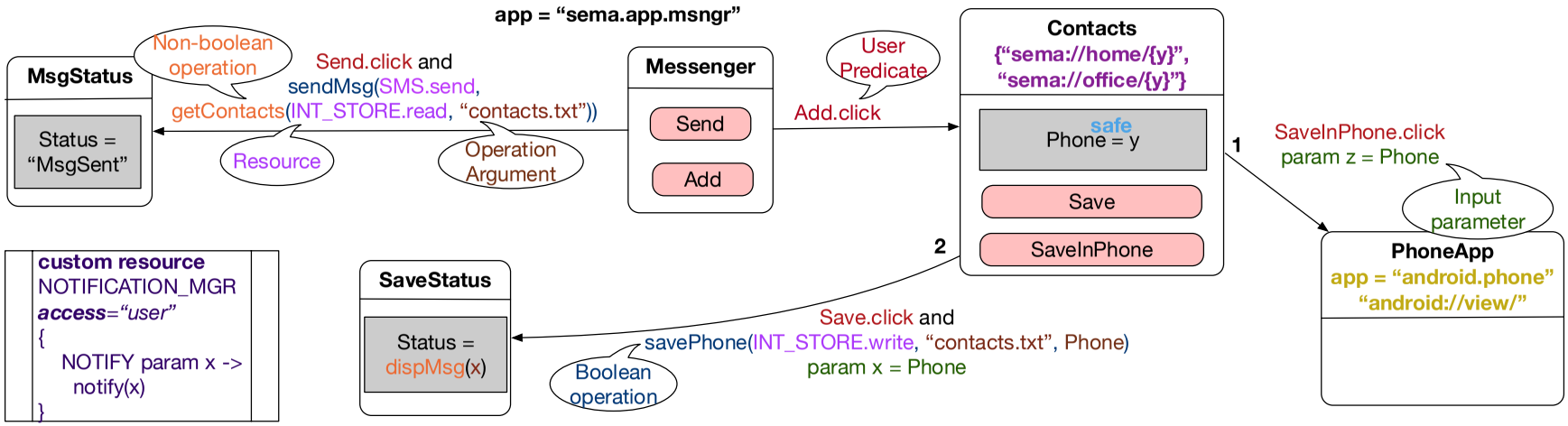


Figure 3: Example of an extended storyboard. The bubbles are not part of the storyboard. They help the reader understand the storyboard entities.

arguments and any required capabilities. For example, in Figure 3, operation *savePhone* is used to save data in the device's internal storage by using the *write* capability of the internal storage device exposed as the resource *INT_STORE*. A use of an operation introduces (declares) it in the storyboard. An operation is defined in the generated implementation (described in Section 3.4). Use of operations *must* be consistent, i.e., a non-boolean operation cannot be used in a boolean value position.

Extension to Transitions Transitions between screens can be adorned with constraints that when satisfied enable/trigger a transition. A constraint is a conjunction of a *user action* (e.g., click of a button) and a set of *boolean operations*. A constraint is satisfied when the user action is performed (e.g., *save* button is clicked) and every boolean operation evaluates to true. For example, in Figure 3, the transition from *Contacts* screen to *SaveStatus* screen is taken only when the user clicks the *Save* button and the *savePhone* operation evaluates to *true*, i.e., the value of *Phone* is successfully saved in "contacts.txt", a file on the internal storage of the device.

As part of a transition, arguments are provided to the input parameters of the destination screen (in green 3). An argument can be a literal specified in the storyboard, a value available in the source screen of the transition (e.g., value of a contained widget, input parameter to the screen), or a value returned by an operation. Further, every transition to a screen must provide values (arguments) for every input parameter of that screen. For example, if there are two transitions *t1* and *t2* to screen s with input parameters x and y, then arguments for both x and y must be provided along both *t1* and *t2*.

Multiple outgoing transitions from a screen may be simultaneously enabled when their constraints are not mutually exclusive. Hence, to handle such situations, all outgoing transitions from a screen must be totally ordered; see *Contacts* screen in Figure 3. The implementation derived from the storyboard will evaluate the constraints according to the specified order of transitions and take the first enabled transition.

3.2 Security Properties

An Android app often interacts with other apps on the device, the underlying platform, and remote servers. Such interaction involves sharing of information, responding to events, and performing tasks based on user actions. Many of these interactions have security implications that should be considered during app development, e.g., can the user's personal information be stored safely on external storage? (information leak and data injection), who should have access to the content provided by the app? (permissions and privilege escalation), how should an app contact the server? (encryption).

While security implications are relevant, our current effort focuses on implications related to confidentiality and integrity of data.

Confidentiality *An app violates confidentiality if it releases data to an untrusted sink.* Hence, an app (and, consequently, its storyboard) that violates confidentiality is deemed as insecure. We explain the concepts of untrusted sinks in a storyboard in Section 3.3.

Integrity *An app violates integrity if it uses a value from an untrusted source.* Hence, an app (and, consequently, its storyboard) that violates integrity is deemed as insecure. We explain how a source is identified as untrusted in Section 3.3.

3.3 Analysis

There are multiple ways to check if extended storyboards satisfy various properties. In our current realization of SeMA, we use *information flow analysis* and *rule checking* to check and help ensure extended storyboards satisfy security properties concerning confidentiality and integrity.

3.3.1 Information Flow Analysis

This analysis tracks the flow of *information* in the form of values from sources to sinks in a storyboard.

A *source* is either a widget in a screen, an input parameter of a screen, or (the return value of) an operation. The set of sources in a storyboard are partitioned trusted and untrusted sources based on the

guarantee of data integrity. Specifically, a source is *untrusted* if it is an input parameter of a screen's URI or it is an operation that reads data from an HTTP server, an open socket, device's external storage, or device's clipboard; or uses the capability of a resource provided by another app. All other sources are deemed as trusted.

A *sink* is either a widget in a screen or an argument to a screen or an operation. The set of sinks in a storyboard are partitioned trusted and untrusted sinks based on the guarantee of data confidentiality. Specifically, a sink is deemed as *untrusted* if it is an argument to an external screen identified without the *app* attribute or to an operation that writes data to an HTTP server, an open socket, device's external storage, or device's clipboard; or uses the capability of a resource provided by another app. All other sinks are deemed as trusted.

To reason about the flow of information between sources and sinks, we define a binary reflexive relation named *influences* between sources and sinks as follows: *influences(x,y)* (i.e., source x influences sink y) if

1. x is assigned to an input parameter y of screen s on an incoming transition to s,
2. x is an argument to operation y, or
3. value of operation x or input parameter x of screen s is assigned to widget y in screen s.

Here are few instances of this relation in Figure 3. *influences(y, Phone)* because input parameter y of *Contacts* screen is assigned to *Phone* widget. *influences(x,dispMsg)* because x is provided as an argument to operation *dispMsg*. *influences(dispMsg, Status)* because the value of operation *dispMsg* is assigned to *Status* widget in *SaveStatus* screen.

Further, all data flows inside the app are guaranteed to preserve the confidentiality and integrity of used/processed data. The operations provided as part of the capabilities of custom resources provided by the app are assumed to preserve the confidentiality and integrity of used/processed data.

With the above (direct) influence relation, guarantees, and assumptions, we use the transitive closure of *influences* relation to detect violation of security properties. We detect *potential violation of integrity* by identifying transitive (indirect) influences $(x, y) \in \textit{influences}^*$ in which x is an *untrusted* source. Likewise, we detect *potential violation of confidentiality* by identifying transitive (indirect) influences $(x, y) \in \textit{influences}^*$ in which y is an *untrusted* sink. All such identified indirect influences are reported to the developer and must to be eliminated. Such an indirect influence can be eliminated by either replacing untrusted sources/sinks with trusted sources/sinks or indicating the indirect influence as safe by marking one or more of the direct influences that contribute the indirect influence as *safe*.[2][3]

For example, in Figure 3, the y parameter of *Contacts* screen is untrusted as it is provided by an external app. So, the analysis will flag $\textit{influences}^*(y, \textit{Phone})$ as violating integrity due to the assignment of y to *Phone* widget in *Contacts* screen. A developer can fix this violation either by removing the assignment of y to *Phone* or marking the assignment as *safe* (as done in Figure 3).

Correctness of the analysis The purpose of the analysis is to identify violations of confidentiality or integrity. This is done in two steps: 1) calculate the potential flows in the storyboard using the transitive closure of *influences* relation and 2) check if a flow involves an untrusted source or sink and is not marked as *safe*. Since the second step is based on pre-defined classification of sources and sinks and developer-provided *safe* annotations, the correctness of the analysis hinges on the first step.

If the *influences* relation correctly captures the direct flow between sources and sinks in a storyboard, then the transitive closure $\textit{influences}^*$ will capture all possible flows between the sources and sinks, including all flows violating confidentiality or integrity. Hence, *the analysis is complete* in identifying every violation of confidentiality or integrity.

The $\textit{influences}^*$ relation does not consider the effect of constraints on flows between sources and sinks, i.e., all constraints on transitions are assumed to be true. Consequently, the analysis may identify a flow between a source and a sink when there is no flow between the source and the sink (at runtime). For example, suppose a screen s has two incoming transitions i_1 and i_2 and two outgoing transitions o_1 and o_2 along with

[2]When multiple sequences of direct influences between a source and a sink support an indirect influence, at least one direct influence in each sequence should be marked as safe to indicate the indirect influence is safe.

[3]This marking is similar to data declassification in traditional information flow analysis.

transition constraints that dictate transition o_x must be taken if and only if s was reached via transition i_x. Further, suppose an input parameter m of s is assigned a value along i_1 and i_2 and used as an argument along o_1 and o_2. In this case, the value assigned to m along i_1 (i_2) will not flow out of m along o_2 (o_1). However, the analysis will incorrectly identify the value assigned to m along i_1 (i_2) may flow out of m along o_2 (o_1). Since the analysis may report invalid violations, *the analysis is unsound.*

3.3.2 Rule Checking

Prior research has developed guidelines and best practices for secure Android app development [20, 12]. Based on these standards, we have developed rules that can be enforced at design time to prevent the violation of properties related to confidentiality and integrity.

Following is the list of rules supported by the current realization of the methodology along with the reasons for the rules.

1. *Capabilities offered by custom resources must be protected by access control.* If any external client can access a custom resource (i.e., its *access* attribute is set to *all*) and the resource offers privileged capabilities, then malicious clients can gain access to privileged capabilities without the user's consent. Further, *Android's policy of least privilege* stipulates that apps should have minimal privileges and acquire the privileges required to use protected services.

2. *WebView widgets must be configured with a whitelist of URL patterns.* A WebView widget in an app works like a browser – it accepts a URL and loads its contents – but it does not have many of the security features of full-blown browsers. Also, a WebView widget has the same privileges as the containing app, has access to the app's resources, can be configured to execute JavaScript code. Hence, loading content from untrusted sources into WebView widgets facilitates exploitation by malicious content.

3. *Operations configured to use HTTPS remote servers must use certificate pinning.* HTTPS remote servers are signed with digital certificates issued by certificate authorities (CAs). Android defines a list of trusted CAs and verifies that the certificate of an HTTPS remote server is signed with a signature from a trusted CA. However, if a trusted CA is compromised, then it can be used to issue certificates for malicious servers. Hence, to protect against such situations, certificates of trusted servers are pinned (stored) in apps and only servers with these certificates are recognized as legit servers by the apps.

4. *Operations configured to use SSL sockets must use certificate pinning.* The reasons from the case of certificate pinning for HTTPS applies here as well.

5. *Cipher operations must use keys stored in secure key stores (containers).* The results of cipher operations can be influenced by tampering the cryptographic keys used in cipher operations. Further, since cryptographic keys are often used across multiple executions of an app, they need to be stored in secondary storage that is often accessible by all apps on a device. Hence, to protect against unwanted influences via key tampering, cipher keys should be stored in secure key stores (containers).

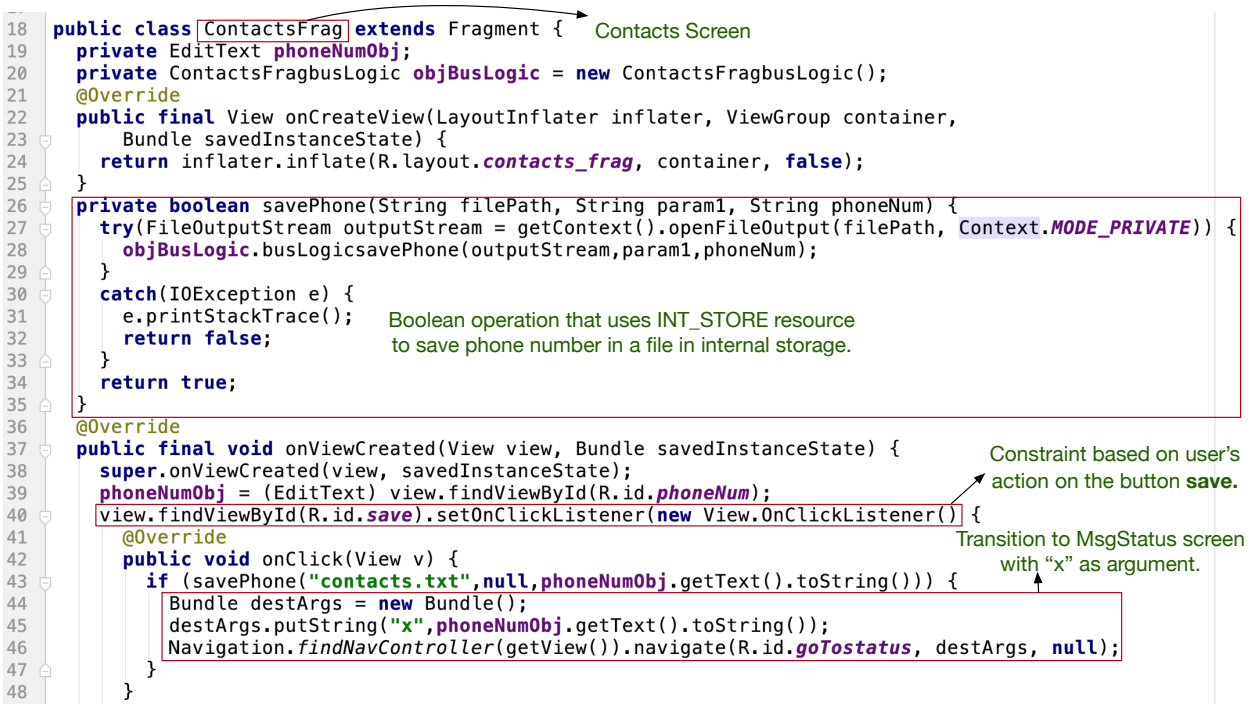
Realization of Rule Checking Violations of rule 1 are detected by checking if a custom resource offers a privileged capability and has its *access* attribute set to *all*.

The *trust-patterns* attribute of *WebView* widget is used to specify the whitelist of trusted URL patterns. Violations of rule 2 is detected by checking if *trust-patterns* attribute is specified for every *WebView* widget.

Violations of rule 5 are detected by checking if the key argument provided to a cipher operation is the value returned by a pre-defined operation to keys from a secure container.

Violations of rules 1, 2, and 5 are flagged as errors and must be addressed before moving to the code generation phase.

Certificate pinning is enabled by default in every storyboard in the methodology. However, since techniques other than certificate pinning can be used to secure connections to servers, a developer can disable certificate pinning by setting *disableCertPin* attribute in a network-related operation. Such cases are detected as violations of rules 3 and 4. They are flagged as warnings but do not inhibit the developer from moving to the code generation phase.




```
18  public class ContactsFrag extends Fragment {
19    private EditText phoneNumObj;
20    private ContactsFragbusLogic objBusLogic = new ContactsFragbusLogic();
21    @Override
22    public final View onCreateView(LayoutInflater inflater, ViewGroup container,
23        Bundle savedInstanceState) {
24      return inflater.inflate(R.layout.contacts_frag, container, false);
25    }
26    private boolean savePhone(String filePath, String param1, String phoneNum) {
27      try(FileOutputStream outputStream = getContext().openFileOutput(filePath, Context.MODE_PRIVATE)) {
28        objBusLogic.busLogicsavePhone(outputStream,param1,phoneNum);
29      }
30      catch(IOException e) {
31        e.printStackTrace();
32        return false;
33      }
34      return true;
35    }
36    @Override
37    public final void onViewCreated(View view, Bundle savedInstanceState) {
38      super.onViewCreated(view, savedInstanceState);
39      phoneNumObj = (EditText) view.findViewById(R.id.phoneNum);
40      view.findViewById(R.id.save).setOnClickListener(new View.OnClickListener() {
41        @Override
42        public void onClick(View v) {
43          if (savePhone("contacts.txt",null,phoneNumObj.getText().toString())) {
44            Bundle destArgs = new Bundle();
45            destArgs.putString("x",phoneNumObj.getText().toString());
46            Navigation.findNavController(getView()).navigate(R.id.goTostatus, destArgs, null);
47          }
48        }
```

Figure 4: Code generated for the Contacts screen in the storyboard depicted in Figure 3

3.4 Code Generation

Once the developer has verified that the specified storyboard does not violate properties related to confidentiality and integrity, she can generate code from the storyboard. Figure 4 shows a fragment of generated code for the running example.

3.4.1 Mapping and Translation Rules

The current realization of SeMA hinges on various choices in mapping and translating storyboard-level entities and concepts into code-level entities and concepts. These choices are encoded in the following mapping and translation rules used during code generation.

1. A Screen is translated to a `Fragment`. For each input parameter of the screen, a function to obtain the value of the parameter is generated. If an input parameter is not available at runtime, then the corresponding function raises a runtime exception.

2. A widget is translated to the corresponding widget type in Android, e.g., a widget displaying text is translated to `TextView`. The value of the widget is the corresponding value specified in the storyboard. For example, if the value is provided by a screen's input parameter x, then the return value from the getter function of x is set as the widget's value, e.g., `TextView.setText(getX())`. The value in a widget is obtained via the corresponding getter function, e.g., `TextView.getText()`.

3. The constraint associated with a transition from a source screen to a destination screen is a conjunction of a user action and boolean operations. The user action part of the constraint is translated to a listener/handler function in the source screen that is triggered by the corresponding user action, e.g., button click. If the constraint has a boolean operation, then a conditional statement is generated with the boolean operation as the condition in the body of the listener function. The *then* block of the conditional statement has the statements required to trigger the destination screen. If the constraint has no boolean operations, then the body of the listener function has statements required to trigger the destination screen. If the constraint has no user action, then the checks corresponding to the boolean operations are performed when the source fragment is loaded.

We use Android's navigation APIs to trigger a destination screen. If the destination screen is a proxy screen, then intents are used to trigger the destination screen determined by the *URI* and *app* attribute. Arguments to destination screens are provided as key/value pairs bundled via the `Bundle` API.

When a screen has multiple outgoing transitions, the statements corresponding to the transitions are chained in the specified order of the transitions in the storyboard.

4. An operation is translated to a function with appropriate input parameters and return value. Each reference to the operation in a storyboard is translated to call the corresponding function.

   The type of the input parameters depends on the type of the arguments provided to the function. For example, if the argument is provided by a widget that displays text, then the type of the parameter will be `String`. The return type depends on how the function is used. For example, if the function is used as a boolean operation in a constraint, then its return type will be `boolean`. If the function is assigned to a widget that displays text, then the function's return type will be `String`.

   If the operation uses a capability provided by a pre-defined resource, then the body of the corresponding function will contain the statements required to use the capability. Otherwise, the function will have an empty body that needs to be later filled in by the developer. For example, on line 27 in Figure 4, function *savePhone* contains the statement required to create a file in the device's internal storage since the same operation uses *write* capability of the resource *INT_STORE* in the storyboard.

5. A developer can extend the generated definitions of functions. For example, on line 28 in Figure 4, the generated code provides a hook for the developer to extend *savePhone*.

6. A custom resource is translated to an appropriate Android component. The capabilities provided by a custom resource can be accessed via an Android intent. Currently, we only support broadcast receivers as custom resources.

7. The use of a resource in the storyboard indicates an app depends on the resource. Such dependencies are captured in the app's configuration during code generation while relying on the Android system to satisfy these dependencies at runtime in accordance with the device's security policy, e.g., grant permission to use a resource at install time.

3.4.2 Challenges in Property Preservation

Code generation offers two challenges to ensure the implementation generated from a storyboarded app does not violate the security properties verified in the storyboard.

Protect Generated Code Against Modifications Even trivial modifications to generated code may lead to violation of security properties. One way to protect against modifications is to have a clear separation between generated code and developer added code. A simple way to achieve this separation is to store generated code and developer added code in separate files. This separation allows the fingerprinting of the generated code at generation time. While building the app, this fingerprint can be used to detect and warn the developer about potential modifications.

In the current realization of the methodology, while generated code and developer added code are stored in separate files, generated code is not fingerprinted to detect potential modifications.

Ensure Business Logic Preserves Properties When developers add business logic, they may use the features of the app or the underlying platform in ways that violate the verified security properties. Such violations can be detected by using a combination of techniques such as runtime checks [16, 19] and app sandboxing [2]. For example, the storyboard shown in Figure 3 uses the read/write capabilities provided by Android's internal file system (INT_STORE). A sandbox will ensure that the app *only* uses files in the app's internal storage.

The current realization of the methodology does not ensure the business logic does not interfere with verified security properties.

4 Canonical Examples

We illustrate the methodology with three canonical examples. Each example demonstrates a vulnerability that can be prevented by the methodology at design time.

4.1 Data Injection Example

Expected app behavior Consider an app p that allows users to log in and view their profile information. From another app, a valid user of app p can navigate only to the screen showing profile information. [4]

Specified app behavior Figure 5A shows the storyboard of app p with four screens: *Start*, *LoginFrag*, *Home*, and *Profile*. In the *Start* screen, when the user clicks on the *Launch* button, the app transitions to *LoginFrag* screen and "Home" is passed as *fragAddr* argument. In *LoginFrag* screen, user can enter her credentials and click on the *Login* button. If the boolean operations *verify* and *isFragProfile* evaluate to true, then the app transitions to *Profile* screen and the return value of operation *getToken* is passed as *token* argument to *Profile* screen. On the other hand, if the boolean operations *verify* and *isFragHome* evaluate to true, then the app transitions to *Home* screen and the value of *Email* widget is passed as *user* argument to *Home* screen.

The *Profile* screen has an outgoing transition with a constraint based on the boolean operation *validToken*. If *validToken* evaluates to *false*, then the app transitions to *LoginFrag* screen and the return value of operation *getFrag* is passed as *fragAddr* argument to *LoginFrag* screen.

The operations *getFrag* and *validToken* consume the *token* parameter of *Profile* screen. The *token* parameter of the URI for *Profile* screen will be provided by an external app. Further, the *token* argument in *validToken* is marked *safe* because the developer trusts *validToken* to preserve confidentiality and integrity.

Security violation *isFragProfile* and *isFragHome* operations consume the input parameter *fragAddr* as argument. On the transition from *Profile* screen to *LoginFrag* screen, *fragAddr* takes on the return value of *getFrag* operation that consumes *token* parameter of *Profile* screen. Since an external app provides the *token* argument when *Profile* screen triggered via its URI, an external app can manipulate *token* to gain access to *Home* screen. Information flow analysis will detect and flag this violation by following the chain of flow from untrusted source and sinks.

Fix This vulnerability can be fixed by changing *param fragAddr = getFrag(token)* to *param fragAddr = "profile"* on the transition from *Profile* screen to *LoginFrag* screen as this breaks the dependence between the app's navigation and *token* parameter.

4.2 Data Leak Example

Expected app behavior Consider an app p that offers two screens to lookup the address of a person and save it. One screen supports name based lookups while the other supports SSN based lookups. Since the person's name, address, and SSN are sensitive; the app wants to ensure that they are not shared without the apps user's consent.

Specified app behavior Figure 5B shows the storyboard of app p. From *Home* screen, a user can navigate to either *NameFinder* or *SSNFinder* screens. In *NameFinder* screen, when a user enters a name and presses the *Find* button, the app transitions to *Display* screen and the return value of operation *getAddr* is passed as argument x *Display* screen. In *SSNFinder* screen, when a user enters an SSN and presses the *Search* button, the app transitions to *Display* screen and the return value of operation *getAddr* is passed argument x to *Display* screen. In *Display* screen, when the user presses the *Save* button, the app transitions to *Home* screen if the boolean operation *saveInfo* evaluates to true, i.e., parameter x is successfully saved in *info.txt* file in external storage.

[4] The example is inspired by the Fragment Injection vulnerability discovered in real-world apps [9].

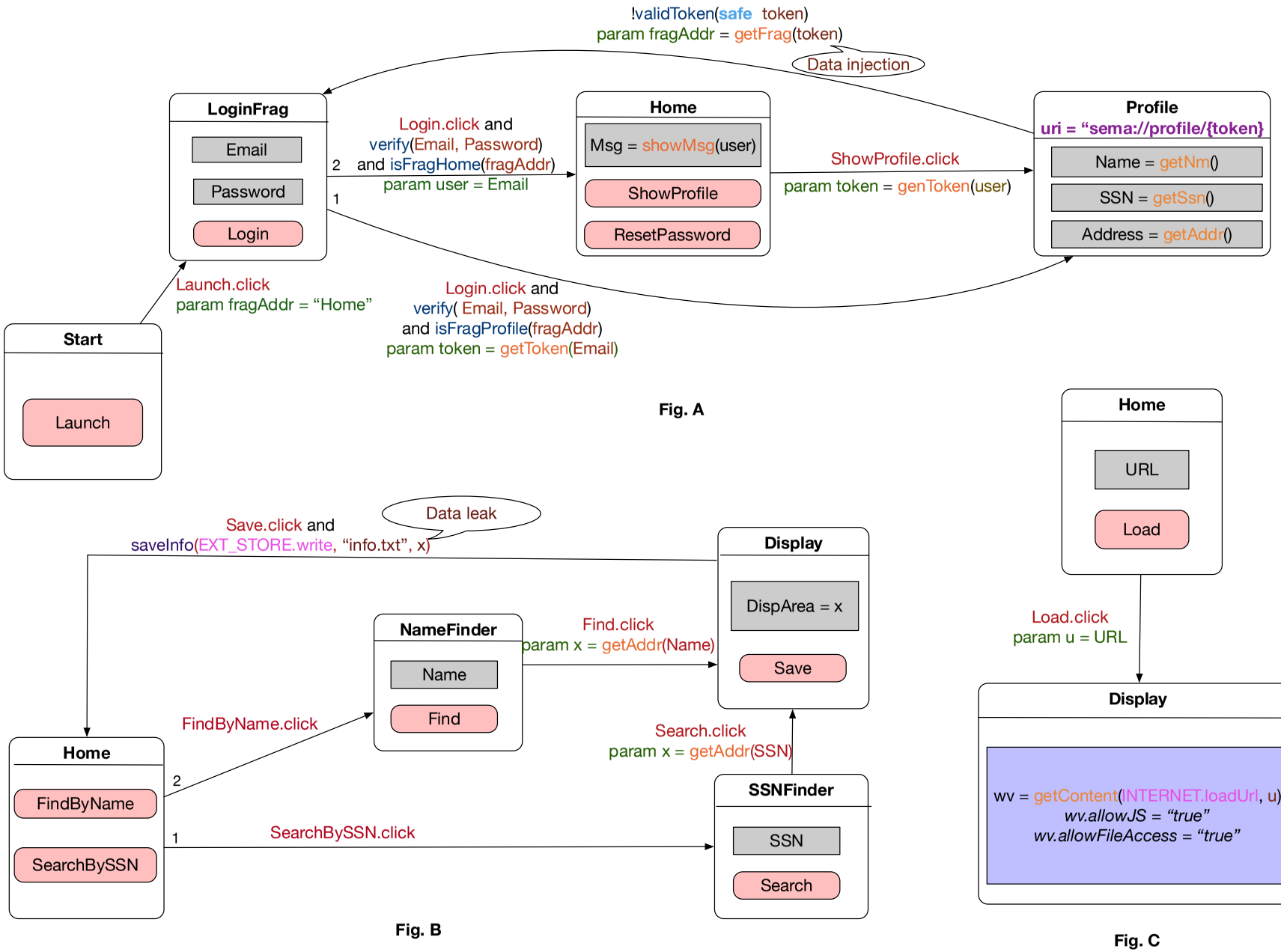


Figure 5: Canonical Examples. The bubbles are not part of the storyboard.

Security violation A person's address obtained from a trusted source, *getAddr()*, is stored in parameter x and is written to the external storage on the transition from *Display* screen to *Home* screen. Since all apps on a device often have access to all parts of the external storage, an app without sufficient permissions can access a person's address. This violates the confidentiality of data read from trusted sources. Information flow analysis will detect and flag this violation by following the chain of flow from sources and untrusted sinks.

Fix This vulnerability can be fixed by changing *EXT_STORE* to *INT_STORE* in the *saveInfo* operation or by marking the argument x to *saveInfo* operation as *safe*.

4.3 Insecure Configuration Example

Expected app behavior Consider an app p that uses a *WebView* widget to display web content from trusted URLs. The app allows execution of JavaScript code embedded in web content. The app also allows JavaScript to access the app's files.

Specified app behavior Figure 5C shows the storyboard of app p. In *Home* screen, a user can provide a URL and press the *Load* button to display the content available at the provided URL in *WebView* widget *wv* in screen *Display*. *WebView* is configured to execute JavaScript (JS) and access the app's files via the *allowJS* and the *allowFileAccess* attributes, respectively.

Security violation Since a whitelist of trusted URLs is not specified for *wv* widget via *trust-patterns* attribute, JavaScript code embedded in web content available from an *untrusted URL* can access the app's files. Rule checking will flag this as a violation of rule 2.

Fix This vulnerability can be fixed by specifying a whitelist of trusted URLs for *wv*, e.g., *wv.trust-patterns*={**sema.org**}.

5 Implementation

Android JetPack Navigation (AJN) is a suite of libraries that helps Android developers design their apps' navigation in the form of navigation graphs. A navigation graph is a realization of a traditional storyboard in Android Studio. We have extended navigation graphs with capabilities that enable developers to specify an app's storyboard, as illustrated in Figure 3, in Android Studio. The developer can visually represent the screens, widgets, and transitions in the navigation graph. While a developer cannot specify operations and constraints visually, she can specify them in the corresponding XML structure of the navigation graph.

We have extended Android Lint [10], a static analysis tool to analyze files in Android Studio, to implement the analysis and verification of security properties. The analysis is packaged as a Gradle Plugin that can be used from Android Studio.

We have implemented a code generation tool that takes a navigation graph and translates it into Java code for Android. A developer can extend the generated code with business logic in Java or Kotlin. The code generation tool is also packaged as a Gradle Plugin that can be used from Android Studio.

6 Evaluation

We used the Ghera benchmark suite [18] to evaluate the methodology. Ghera has 60 benchmarks. Each benchmark captures a unique vulnerability. We used Ghera because the vulnerabilities in Ghera benchmarks are *valid*, i.e., they have been previously reported in the literature or documented in Android documentation. These vulnerabilities are *general and exploitable* as they can be verified by executing the corresponding benchmarks on vanilla Android devices and emulators. Further, each vulnerability is *current* as they are based on Android API levels 22 thru 27, which enable more than 90% of Android devices in the world and are targeted by both existing and new apps. Finally, the benchmarks are known to be *representative* of

real-world apps in terms of the APIs they use [23]. Hence, the benchmarks in Ghera are well-suited for this evaluation.

For each Ghera benchmark, we used the methodology to develop an app with the vulnerability captured in the benchmark. Table 1 summarizes the results. The methodology prevented 49 of the 60 vulnerabilities captured in Ghera benchmarks. Of the 49 prevented vulnerabilities, 30 were prevented by information flow analysis and rule checking of the storyboard. The remaining 19 were prevented by code generation.

Of the 30 vulnerabilities detected by the analysis on the storyboard, 15 were prevented by information flow, nine were prevented due to rule analysis, and six were prevented by a combination of information flow and rule analysis. Two of these six vulnerabilities were prevented by rule analysis and could have been prevented by code generation. These two vulnerabilities relate to connecting HTTP remote servers and connecting to HTTPS remote servers without certificate pinning. While such vulnerabilities can be prevented by code generation, we chose rule checking in our realization of the methodology to offer flexibility in using HTTP vs. HTTPS and certificate pinning in storyboards when connecting to remote servers.

Of the 49 vulnerabilities prevented by the methodology, 28 can be detected curatively (by source code analysis) after implementing the apps. Detecting the remaining 21 vulnerabilities by source code analysis is harder due to combinations of factors such as the semantics of general-purpose programming languages (e.g., Java), security-related specs provided by the developer (e.g., source/sink APIs), and the behavior of the underlying system (e.g., Android libraries and runtime). Hence, many existing tools based on source code analysis fail to detect vulnerabilities in real-world apps effectively [23, 21, 17].

In [23], Ranganath and Mitra evaluated 14 security analysis tools based on deep and shallow analysis. While security analysis tools in isolation could detect at most 15 vulnerabilities, the full set of tools collectively detected 30 vulnerabilities. This suggests combining different analysis will likely be more effective in detecting vulnerabilities. Our experience with the proposed methodology suggests the same is likely true in the context of preventing vulnerabilities: a combination of information flow analysis (deep), rule checking (shallow), and code generation (shallow) helped detect and prevent 49 vulnerabilities.

The current realization of the methodology was not applicable to 11 benchmarks in Ghera. Of these, three benchmarks capture vulnerabilities that cannot be prevented by the methodology, e.g., unhandled exceptions. The remaining eight apps use features that are not yet supported by the methodology, e.g., Content Providers.

The current realization of SeMA is available in a publicly available repository: `https://bitbucket.org/secure-it-i/sema/src/master/` .

7 Future Work

We plan to explore the following directions to extend this effort.

1. Study the effectivenss of the methodology in preventing vulnerabilities while developing real-world apps.
2. Study the usability of the methodology while developing real-world apps.
3. Extend code generation to preserve the integrity of the generated files.
4. Extend the storyboard with additional capabilities to help specify richer behavior, e.g., Content Providers.

8 Summary

In this paper, we have proposed a methodology that extends storyboarding and combines it with model-driven development to help build secure Android apps. The methodology can prevent developers from introducing confidentiality and integrity related vulnerabilities into Android apps. This is achieved by using information flow anlaysis, rule checking, and automatic code generation. We empirically evaluated the methodology against a set of 60 known Android app vulnerabilities. The methodology successfully helped prevent 49 of these vulnerabilities.

Benchmark	Vulnerability Description	Method
BlockCipher-ECB-InfoExposure	Block Cipher in ECB mode weakens the encryption.	CG
BlockCipher-NonRandomIV-InfoExposure	Non-random IV weakens the encryption.	CG
ConstantKey-Forgery	Use of a constant secret key exposes the secret key.	RC
ExposedCredentials-InfoExposure	No authentication of KeyStore exposes the KeyStore.	CG
PBE-ConstantSalt-InfoExposure	Constant Salt in password-based secret key exposes the key.	CG
DynamicBroadcast-UnrestrictedAccess	Dynamic registration of broadcast receiver exposes it.	RC
EmptyPendingIntent-PrivEscalation	Sharing empty pending intent leads to intent hijacking.	IF
FragmentInjection-PrivEscalation	Dynamic Fragment loading enables fragment injection.	IF
HighPriority-ActivityHijack	Low priority activity can be hijacked by high priority activity.	IF
ImplicitPendingIntent-PrivEscalation	Sharing implicit pending intent leads to intent hijacking.	IF
IncorrectHandlingImplicitIntent-UnauthorizedAccess	Not validating implicit intents grants unauthorized access.	IF
NoValidityCheckOnBroadcast-UnintendedInvocation	Not verifying a broadcast message exposes the receiver.	RC
OrderedBroadcast-DataInjection	Accepting input from a receiver enables data injection.	IF
UnprotectedBroadcastRecv-PrivEscalation	Not protecting broadcast receiver enables privilege escalation.	RC
TaskAffinity-ActivityHijack	Starting activity in a new task enables activity hijack.	CG
TaskAffinity-LauncherActivity-PhishingAttack	Non-empty task affinity of launcher activity enables phishing.	CG
TaskAffinity-PhishingAttack	Startign activity in a new task enables phishing.	CG
TaskAffinityAndReparenting-PhishingAndDoSAttack	Non-empty task affinity of an activity enables phishing and DoS	CG
CheckValidity-InfoExposure	Incorrect checking of server certificates enables MITM.	CG
IncorrectHostNameVerification-MITM	Incorrect verification of server hostname enables MITM.	CG
InsecureSSLSocket-MITM	Not verifying the hostname of an SSL socket enables MITM.	CG
InsecureSSLSocketFactory-MITM	*SSLCertificateSocketFactory.getInsecure()* API enables MITM.	CG
InvalidCertificateAuthority-MITM	Incorrect checking of certificate authority enables MITM.	CG
OpenSocket-InfoLeak	Sending data via open socket enables data theft.	IF
UnEncryptedSocketComm-DataInjection	Reading data from an open socket enables data injection.	IF
UnPinnedCertificate-MITM	Absence of pinned certificates enables MITM.	RC
UnnecesaryPerms-PrivEscalation	Using unnecessary permissions enables privilege escalation.	CG
WeakPermission-UnauthorizedAccess	Weak permissions do not provide adequate protection.	RC
ExternalStorage-DataInjection	Reading from files in external storage enables data injection.	IF
ExternalStorage-InformationLeak	Writing to files in external storage enables data theft.	IF
InternalStorage-DirectoryTraversal	Unsanitized paths to internal storage exposes internal files.	IF
InternalToExternalStorage-InformationLeak	Writing to files in external storage from files in internal storage enables data theft.	IF
CheckCallingOrSelfPermission-PrivilegeEscalation	API *CheckCallingOrSelfPermission* does not provide protection	CG
CheckPermission-PrivilegeEscalation	API *CheckPermission* does not provide protection	CG
EnforceCallingOrSelfPermission-PrivilegeEscalation	API *EnforceCallingOrSelfPermission* does not provide protection	CG
EnforcePermission-PrivilegeEscalation	API *EnforceCallingOrSelfPermission* does not provide protection	CG
ClipboardUse-InformationExposure	Writing data to the clip board enables data theft.	IF
DynamicCodeLoading-CodeInjection	Executing code from unverified source enables code onjection.	IF
UniqueIDs-IdentityLeak	Sharing unique system ID enable ID theft.	IF
WebView-CookieOverwrite	Malicious URLs in a WebView overwrite cookies.	IF & RC
HttpConnection-MITM	Connecting to HTTP remote servers enables MITM.	RC
JavaScriptExecution-CodeInjection	Malicious URL in a WebView can inject & execute JS.	IF & RC
UnsafeIntentURLImpl-InformationExposure	Malicious URL in a WebView can embed intents to steal data.	IF & RC
WebViewAllowContentAccess-UnauthorizedFileAccess	Malicious URL in a WebView can access internal content providers.	IF & RC
WebViewAllowFileAccess-UnauthorizedFileAccess-Lean	Malicious URL in a WebView can access internal files.	IF & RC
WebViewIgnoreSSLWarning-MITM	Ignoring SSL errors enables MITM.	CG
WebViewInterceptRequest-MITM	A WebView not validating resource requests enables MITM	RC
WebViewLoadDataWithBaseUrl-UnauthorizedFileAccess	A WebView with no base URL enables file access to a malicious URL	IF & RC
WebViewOverrideUrl-MITM	A WebView not validating page requests enables MITM	RC

Table 1: Results showing how a vulnerability in a benchmark was detected and prevented. CG, IF, and RC refer to Code Generation, Information Flow Analysis, and Rule-based Analysis respectively.

Acknowledgement

We thank Dr. Torben Amtoft for providing valuable feedback and ideas to improve this effort.

References

[1] Apple. Storyboards in Xcode. https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html, 2019. Accessed: 05-Feb-2019.

[2] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and von Styp-Rekowsky, P. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), USENIX Association, pp. 691–706.

[3] Brambilla, M., Cabot, J., and Wimmer, M. *Model-Driven Software Engineering in Practice: Second Edition*, 2nd ed. Morgan & Claypool Publishers, 2017.

[4] Brambilla, M., Mauri, A., and Umuhoza, E. Extending the interaction flow modeling language (ifml) for model driven development of mobile applications front end. In *Mobile Web Information Systems* (2014), I. Awan, M. Younas, X. Franch, and C. Quer, Eds., Springer International Publishing, pp. 176–191.

[5] Checkmarx. How attackers could hijack your android camera to spy on you. Available at https://www.checkmarx.com/blog/how-attackers-could-hijack-your-android-camera.

[6] Fatima, I., Anwar, M. W., Azam, F., Maqbool, B., and Tufail, H. Extending interaction flow modeling language (ifml) for android user interface components. In *Information and Software Technologies* (2019), R. Damaševičius and G. Vasiljevienė, Eds., Springer International Publishing, pp. 76–89.

[7] Heitkotter, H., Kuchen, H., and Majchrzak, T. A. Extending a model-driven cross-platform development approach for business apps. *Science of Computer Programming 97* (2015), 31 – 36. Special Issue on New Ideas and Emerging Results in Understanding Software.

[8] Henkel, M., and Stirna, J. Pondering on the key functionality of model driven development tools: The case of mendix. In *Perspectives in Business Informatics Research* (2010), Springer Berlin Heidelberg, pp. 146–160.

[9] IBM. Android collapses into fragments. Available at https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf.

[10] Inc., G. Android lint overview. Available at http://tools.android.com/lint/overview.

[11] Inc., G. Navigation. Available at https://developer.android.com/guide/navigation.

[12] Institute, C. M. S. E. Android secure coding standards. Available at https://wiki.sei.cmu.edu/confluence/display/android/Android+Secure+Coding+Standard.

[13] Jackson, D. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11* (2002), 256–290.

[14] Lamport, L. Who builds a house without drawing blueprints? *Commun. ACM 58*, 4 (Mar. 2015), 38–41.

[15] Little, A. Storyboarding in the software design process. https://uxmag.com/articles/storyboarding-in-the-software-design-process, 2013. Accessed: 05-Feb-2019.

[16] Liu, J., Wu, T., Yan, J., and Zhang, J. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)* (2016), pp. 342–352.

[17] Luo, L., Bodden, E., and Spath, J. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), IEEE, pp. 102–114.

[18] Mitra, J., and Ranganath, V.-P. Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering* (2017), ACM, pp. 43–52.

[19] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference* (2009), IEEE Computer Society, p. 340–349.

[20] OWASP. Mobile security project. Available at https://wiki.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Controls.

[21] Pauck, F., Bodden, E., and Wehrheim, H. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), ACM, pp. 331–341. https://foellix.github.io/ReproDroid/.

[22] Prudnikov, D. UX design techniques for mobile apps. https://yalantis.com/blog/ux-design-techniques-mobile-app-design/, 2019. Accessed: 05-Feb-2019.

[23] Ranganath, V.-P., and Mitra, J. Are free android app security analysis tools effective in detecting known vulnerabilities? *Empirical Software Engineering* (2019).

[24] Richters, M., and Gogolla, M. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language* (Berlin, Heidelberg, 2002), Springer-Verlag, p. 42–68.

[25] Sketch. Sketch Design ToolKit. https://www.sketchapp.com/, 2019. Accessed: 05-Feb-2019.

[26] Sufatrio, Tan, D. J. J., Chua, T.-W., and Thing, V. L. L. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.* (2015), 58:1–58:45.

[27] Telang, R., and Wattal, S. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Trans. Softw. Eng. 33* (2007), 544–557.

[28] Vaupel, S., Taentzer, G., Gerlach, R., and Guckert, M. Model-driven development of platform-independent mobile applications supporting role-based app variability. In *Software Engineering 2016* (2016), Gesellschaft für Informatik e.V., pp. 99–100.

[29] Wang, Y., Liu, X., Mao, W., and Wang, W. Dcdroid: Automated detection of ssl/tls certificate verification vulnerabilities in android apps. In *Proceedings of the ACM Turing Celebration Conference - China* (2019), ACM TURC '19, Association for Computing Machinery.

[30] Wayne, H. *Practical TLA+: Planning Driven Development*, 1st ed. Apress, USA, 2018.