Context-aware security testing of Android applications

Degree Project Presentation - LCIS, Valence

Ivan Baheux

School of Electrical Engineering and Computer Science
Kungliga Tekniska högskolan (KTH)

February, 2023

Project supervised by :

- Karl Palmskog, Lecturer for KTH
- Oum-El-Kheir Aktouf, Professeure des Universités for LCIS

Motivation

- A large amount of people use mobile phones
- An average user spends 3h39 actively using internet daily
- 200 billion applications with approximately 75% still having flaws
- 70% of the market share is held by Android

Why is this thesis usefull

In mobile development we face multiple issues:

- **Need for security**
- **Context**

  → Research has shown that vulnerabilities can happen in specific contexts of an application, whether it is dynamic or static context

Goals

During the thesis ours goals where:

1. Analyse the vulnerabilities linking an application to it's context
2. Security testing process to detect vulnerabilities related to the context

A few constraints have been added during the pre-study phase:

- Usage of Domain Specific Languages (**DSLs**)
- Extendable solution
- Allow the separation between the usage of the tool and the vulnerabilities

Defining the context

Following the definition by Almeida et al. [1]. We give the following definitions :

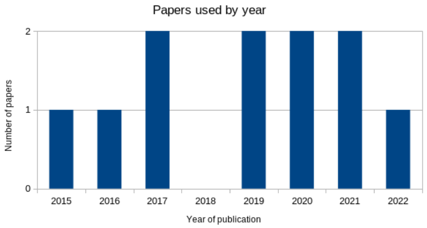
- **Static Context**: Context that does not evolve with time during the application's execution
- **Dynamic Context**: Context that may evolve with time during the application's execution
- **Derived Context (or Situation)**: High level contexts acting as an agglomeration of contexts or specific context values

Selection process

First step, analyze the vulnerabilities of Android applications.

Paper research rules:

- No paper that came out before 2015
- Vulnerabilities relevant to Android applications



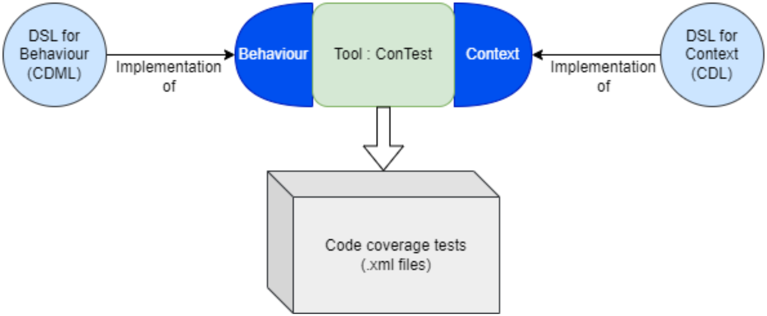
Processed result

Vulnerability	Is application code enough.	Context	Explanation
Untrusted or Manipulated Sensors	Not applicable	Dynamic	Detected with unusual context modifications. May be detected with unusual sensor patterns
Outdated API version, Sensitive API	No	Static	Detected through application calls to vulnerable functions or old API configurations.
SQLite	Yes		Detected through code review.
Storage Access Vulnerability	Sometimes	Static	Detected by checking configuration of readable content.
Hardcoded private data or broken cryptography	Yes	Static	Detected through bad cryptographic function written, hardcoded values or bad libraries.
Outdated library or third-party library	Yes		Detected through bad library usage.
Intra library collusion	Not applicable	Dynamic	Detected through contextual checking of other applications using the same library. Also checking library code.
Intent spoofing	Yes	Static	Detected through bad configurations of the activities/services or code that gives too much rights to incoming intents.
Unauthorized intent receipt	No	Dynamic	Detected through bad coding practices when writing broadcasts. Detected through strange activity overlap between applications.
Untrusted user inputs	Yes		Detected through input sanitizing.
Incorrect data flow	No	Dynamic	Detected by checking application code or private data leakage on public channels

Table 1: Vulnerabilities and their detectable features

Base of the idea

Part of the base idea of this master thesis wouldn't have been possible without Abdallah's [2] submission.



Usage of his contribution

His theory has been implemented and extended for the field of security.

His methodology is interesting:

- Usage of Models: MBT (Model Based Testing)
- Context-aware analysis
- Abstract tests

Global methodology

The project's methodology was created around 5 axis:

- Select specifications based on our vulnerability survey
- Conceptualise the architecture of the selected solution
- Build the project around the MBST methodology
- Measure the resulting tool
- Allow the tool to be reused in future works

Data Collection and testing methodology

To validate our findings we used a free application repository specifically made for security testing, GHERA [3].

Vulnerability Type	Crypto	ICC	Networking	NonAPI	Permission	Storage	System	Web
Number of examples	5	17	8	2	2	7	7	12

Limitations:

- One test per vulnerability type
- Sometimes "fake" vulnerabilities

Reproducibility

In order for this work to be reusable in future works:

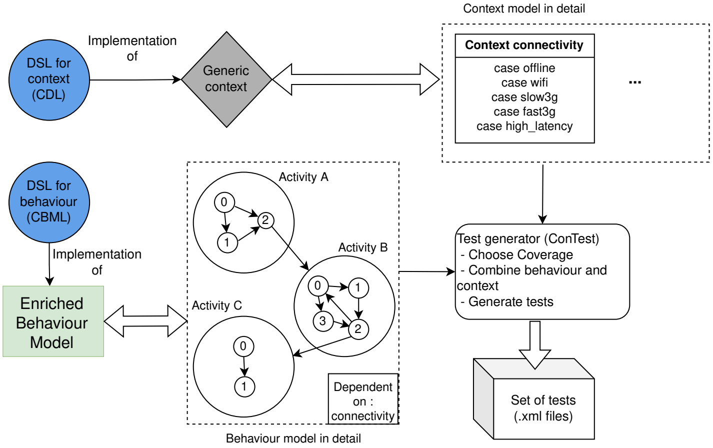
- Full documentation at function level and installation
- The full code, documentation and associated reports on GitHub

Realisations - TL;DR of the project

During the master thesis we created two tools:

- ConTest: Based on the work of Abdallah [2], context-aware Android Application code coverage with abstract tests.
- VPatChecker: Context-Aware vulnerability checker and abstract exploit generator.

Global architecture of ConTest



CDL - Context Definition Language

- Defines the contexts that applications can have
- Follows our definition of static, dynamic and derived contexts
- Is used by the application model to properly define context

```
context INTERNET_CONNECTIVITY {
    providers: [WIFI_ADAPTER, CELL_ADAPTER],
    properties: [connectivity: Connectivity]
}
```

```
type Connectivity {offline, wifi, slow3G, fast3G, _4g, high_latency}
```

Listing 1: Example of context model for internet connectivity

Context-Driven Modelling Language

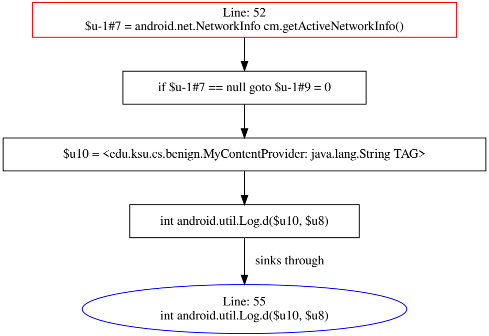
- Defines the behaviour of an application in regards to its context
- Depends on the CDL implementation for context

```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {

    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED −> SHOW_ANSWER
    }

    state SHOW_ANSWER {
    }
}
```

Listing 2: Excerpt from an example of application behaviour model

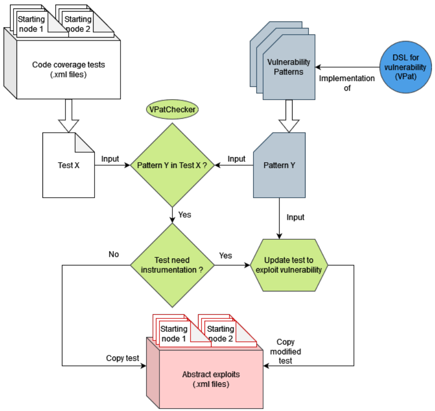
FlowDroid

- Allows to determine flows of information from source to sink
- We can select what functions are sources and sinks in order to also see the flows of user inputs to specific functions


Line: 52
$u-1#7 = android.net.NetworkInfo cm.getActiveNetworkInfo()
if $u-1#7 == null goto $u-1#9 = 0
$u10 = <edu.ksu.cs.benign.MyContentProvider: java.lang.String TAG>
int android.util.Log.d($u10, $u8)
sinks through
Line: 55
int android.util.Log.d($u10, $u8)

Listing 1: Flowgraph given by the enrichment script, from FlowDroid's output

Global architecture of VPatChecker



Objectives :

- Allow developers to get feedback on the vulnerabilities they add during the development of their application
- Allow pentesters to update the tool with each new vulnerability without changing the code

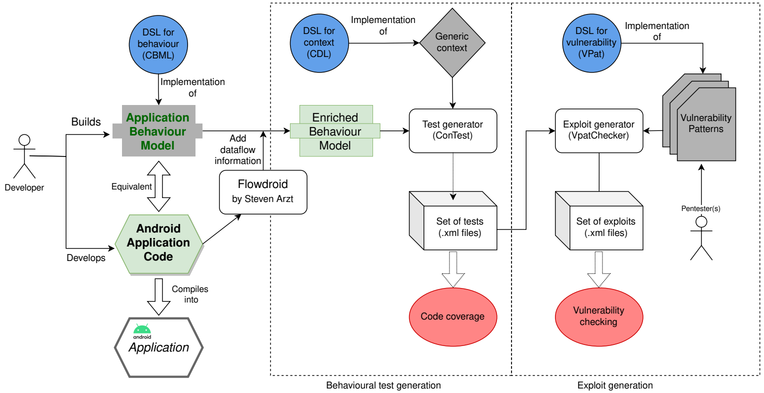
VPat - Vulnerability Pattern

- Allows a pentester to write the equivalent of a vulnerability to be checked on any application by VPatChecker
- A vulnerability can be defined by the function it uses, the context it depends on and the flow of data

```
Vulnerability "log.t API 26" {
    description "log.t can be exploited by giving his first
                 argument 'EXPLOITABLE'"
    context {
        apiversion "26"
    }
    function {
        main Sink "log.t" {
            parameter {private, static "EXPLOITABLE"}
        },
        Source private *
    }
}
```

Listing 3: Example of vulnerability pattern

General Architecture



Generating tests

The first step of the full process is to create the model, for our example let's use the following model:

Generating tests - Behaviour model : Context

```
model TranslationApp {

    contexts {
        INTERNET_CONNECTIVITY
    }

    static contexts {
        minSdk = "26",
        maxSdk = "",
        targetSdk = "32"
    }

    situations {
        INTERNET_DISCONNECTED : INTERNET_CONNECTIVITY,
        INTERNET_SLOW : INTERNET_CONNECTIVITY
    }
```

Listing 4: Excerpt from an example of application behaviour model

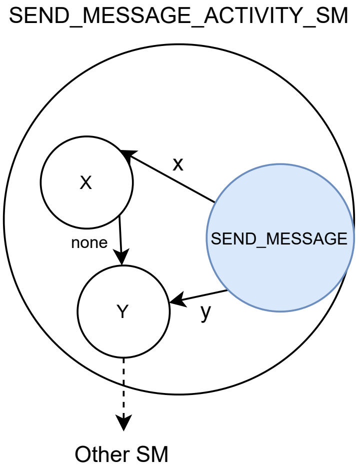
Generating tests - Behaviour model : States

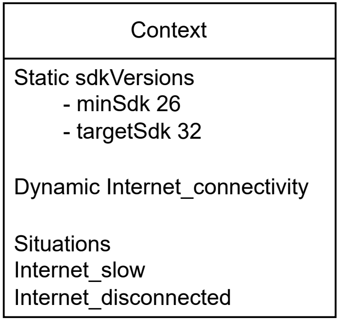
```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {
    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED −> SENDER
        dataflows {
            source internet
        }
    }
```

. . .

```
adaptation for INTERNET_SLOW at SEND_MESSAGE {
    state SEND_MESSAGE {
        transition on NONE −> HANDLE_SLOW_INTERNET
    }

    state HANDLE_SLOW_INTERNET {
        transition on NONE −> external SEND_MESSAGE_ACTIVITY_SM.SENDER
        dataflows {
            sink "log.d" ( source SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE.internet )
        }
    }
}
```

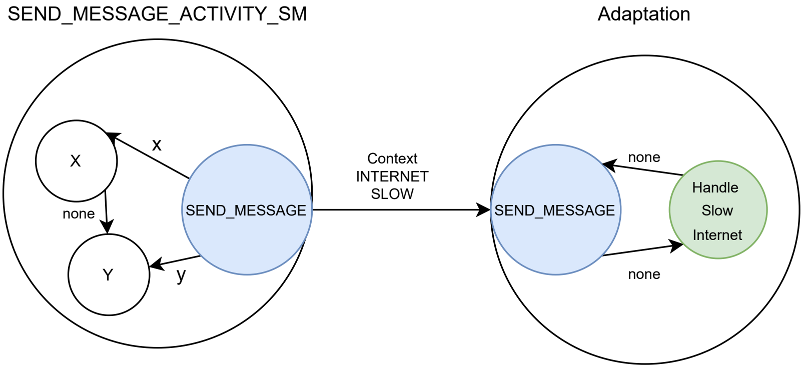
Generating tests - Base StateMachine



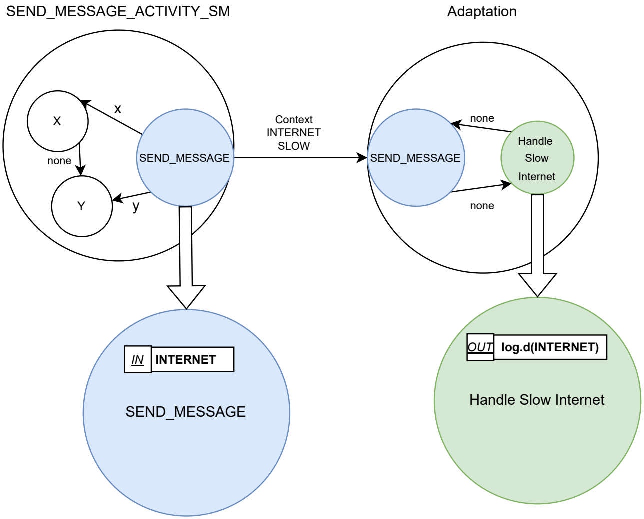


Context
Static sdkVersions - minSdk 26 - targetSdk 32 Dynamic Internet_connectivity Situations Internet_slow Internet_disconnected

Generating tests - Adapted State



Generating tests - Dataflows



Generating tests - Results from ConTest

```
 <TestPath>
  <state name="SEND_MESSAGE">
      <transition name="NONE">
          <contexts>
              <context origin="INTERNET_CONNECTIVITY">slow3G</context>
          </contexts>
          <situations>
              <situation origin="DEFAULT_ORIGIN">INTERNET_SLOW</situation>
          </situations>
      </transition>
      <dataflows>
          <dataflow name="internet" type="Source"/>
      </dataflows>
  </state>
  <state name="HANDLE_SLOW_INTERNET">
      <transition name="NONE"/>
      <dataflows>
          <dataflow name="log.d" type="Sink">
              <parameters>
                  <parameter origin="source">internet</parameter>
              </parameters>
          </dataflow>
      </dataflows>
  </state>
  ...
</TestPath>
```

Generating exploit - Patterns

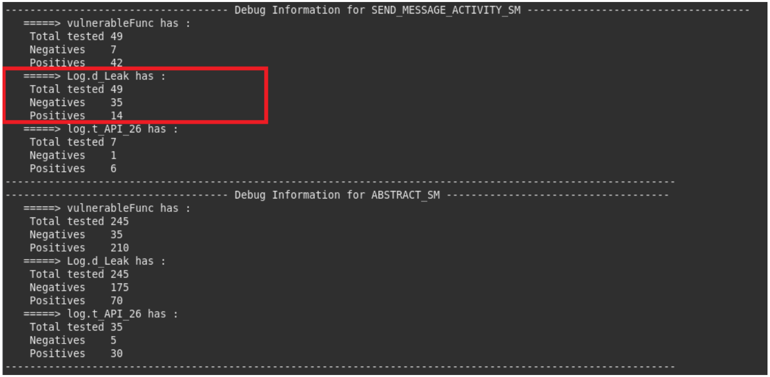
If we use the check the generated tests with the following pattern (The other patterns are hidden for readability):

```
Vulnerability "Log.d Leak" {
    description "Log.d kept in code makes it vulnerable to leakage of data"

    function {
        main Sink "log.d" {
            parameter {
                private
            }
        } ,
        Source private *
    }
}
```

Listing 5: Example of vulnerability pattern

Generating exploit - Reporting

The following report is generated, one for each starting node and for each vulnerability pattern:



```
------------------------------------ Debug Information for SEND_MESSAGE_ACTIVITY_SM ------------------------------------
   =====> vulnerableFunc has :
    Total tested 49
    Negatives    7
    Positives    42
   =====> Log.d_Leak has :
    Total tested 49
    Negatives    35
    Positives    14
   =====> log.t_API_26 has :
    Total tested 7
    Negatives    1
    Positives    6
--------------------------------------------------------------------------------------------------------------
------------------------------------ Debug Information for ABSTRACT_SM ------------------------------------
   =====> vulnerableFunc has :
    Total tested 245
    Negatives    35
    Positives    210
   =====> Log.d_Leak has :
    Total tested 245
    Negatives    175
    Positives    70
   =====> log.t_API_26 has :
    Total tested 35
    Negatives    5
    Positives    30
--------------------------------------------------------------------------------------------------------------
```

Generating exploit - Results from VPatChecker

For our "log.d leak" tests are not modified but when an input needs to be controlled we can transform the test into an exploit:

```
<TestPath>
  ...
  <state name="SENDER">
        <transition name="WRONG"/>
        <dataflows>
            <dataflow name="enter_value" type="Input" value="EXPLOITED"/>
        </dataflows>
    </state>
    <state name="DISPLAY_WARNING">
        <transition name="BACK_BUTTON_PRESSED"/>
        <dataflows>
            <dataflow name="vulnerableFunc" type="Sink">
                <parameters>
                    <parameter origin="source">internet</parameter>
                    <parameter origin="source">enter_value</parameter>
                </parameters>
            </dataflow>
        </dataflows>
    </state>
    ...
</TestPath>
```

Results

Vulnerability type	Number of total vulnerabilities	Number of detectable vulnerabilities	Percentage of detectability
Permission	2	1	50%
NonApi	2	0	0%
Crypto	5	4	80%
ICC	17	4	24%
Networking	8	2	25%
Storage	7	5	71%
System	7	7	100%
Web	12	Out of scope	Out of scope
Total	60	23	38%

Realisations

Our results show that:

- DSL's and Model-Based Security Testing can be impactful in the current Android security ecosystem
- Models can ease the detection of vulnerabilities
- The methodology used with VPatChecker gives an insight on how an open-sourced security tool may be created

Limitations

- A limited amount of vulnerability pattern have been created for the report
- The process is not automated. The abstract tests are not translated
- The matching algorithm is simplistic and does not detect very complex or hidden vulnerabilities (Think obfuscation)

Future work

A certain upgrades can be made and should be made to have a fully mature project:

- Automating the processes
- Providing a more precise guidance to develop vulnerability pattern (classification..)
- More complex vulnerability patterns

[1] D. R. Almeida, P. D. L. Machado, and W. L. Andrade, "Testing tools for Android context-aware applications: a systematic mapping," *Journal of the Brazilian Computer Society*, vol. 25, p. 12, Dec. 2019.

[2] A. Adwan, "Context-dependent Model-based Testing of Mobile Apps," Master's thesis, University Grenoble Alpes, France, 2021.

[3] J. Mitra and V.-P. Ranganath, "Ghera: A Repository of Android App Vulnerability Benchmarks," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, (New York, NY, USA), pp. 43–52, Association for Computing Machinery, Nov. 2017.

Thank You

Code and documentation:

https://github.com/Myshtea/VPatChecker

Annex

Additional slides

Specifications

Our design will follow the goals we defined in introduction. We could detail the motivations with these bullet points:

- Allow a developer to have a tool giving him a demonstration of what vulnerabilities reside in his code
- Separate the design in two actors: The developer handles the code of the application and the pentester defines what vulnerabilities should be looked for
- Simplify the expansion process of the tool, giving the possibility to enhance the amount of vulnerabilities that can be detected
- Allow to detect vulnerabilities in specific contexts (Specific permissions, android versions or network configurations)

Generating tests - Behaviour model : Context

```
model TranslationApp {

    contexts {
        INTERNET_CONNECTIVITY
    }

    static contexts {
        minSdk = "26",
        maxSdk = "",
        targetSdk = "32"
    }

    situations {
        INTERNET_DISCONNECTED : INTERNET_CONNECTIVITY,
        INTERNET_SLOW : INTERNET_CONNECTIVITY
    }
```

Listing 6: Excerpt from an example of application behaviour model

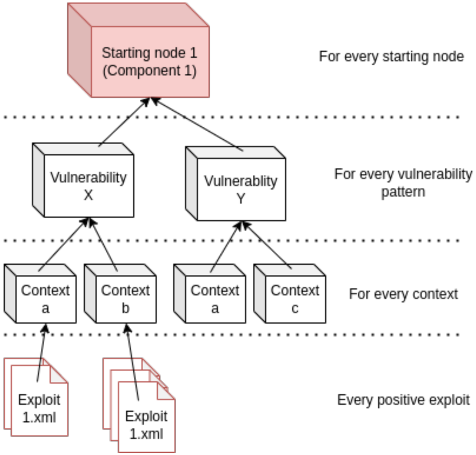
Generating tests - Behaviour model : States

```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {
    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED -> SENDER
        dataflows {
            source internet
        }
    }
```

. . .

```
adaptation for INTERNET_SLOW at SEND_MESSAGE {
    state SEND_MESSAGE {
        transition on NONE -> HANDLE_SLOW_INTERNET
    }

    state HANDLE_SLOW_INTERNET {
        transition on NONE -> external SEND_MESSAGE_ACTIVITY_SM.SENDER
        dataflows {
            sink "log.d" ( source SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE.internet )
        }
    }
}
```

Reporting



Listing 2: Reporting created by VPatChecker