DroidSecTester: Towards context-driven modelling and detection of Android application vulnerabilities

1st Ivan Baheux
Univ. Grenoble Alpes
Grenoble INP LCIS, F-38000
Valence, France
ivan.baheux@protonmail.com

2nd Oum-El-Kheir Aktouf
Univ. Grenoble Alpes
Grenoble INP LCIS, F-38000
Valence, France
oum-el-kheir.aktouf@lcis.grenoble-inp.fr

3rd Mohammed El Amin Tebib
Univ. Grenoble Alpes
Grenoble INP LCIS, F-38000
Valence, France
mohammed-el-amin.tebib@lcis.grenoble-inp.fr

4th Mariem Graa
CNAM
621 Avenue Centrale, CNAM
Nantes, France
mariem.graa@gmail.com

5th Pascal Andre
LS2N, CNRS
2 rue de la houssiniere, LS2N
Nantes, France
pascal.andre@ls2n.fr

6th Yves Ledru
Univ. Grenoble Alpes, CNRS
Grenoble INP, LIG, F-38000
Grenoble, France
Yves.Ledru@univ-grenoble-alpes.fr

***Abstract*—In the dynamic Android application security landscape, traditional vulnerability assessment faces challenges posed by the increasing complexity of execution environments. These environments encompass a diverse array of contextual factors that influence application behavior, highlighting the imperative for adaptive testing. Current security analysis techniques for Android apps often struggle to capture the intricate interplay between static and dynamic contexts, impeding precise vulnerability detection. This constraint becomes more evident as execution environments diversify.**

To address these limitations, this paper introduces DroidSecTester, a novel toolchain for testing Android application security by focusing on context-driven vulnerability modeling. Our innovation lies in developing three Domain Specific Languages (DSLs): Context Definition Language (CDL), Context-Driven Modelling Language (CDML), and Vulnerability Pattern (VPat) for Model-Based Security Testing (MBST). Collectively, these DSLs provide a framework for security assessment by embracing both static and dynamic contexts intrinsic to smartphone environments.

Our work resulted in VPatChecker, a tool designed to identify vulnerabilities and generate abstract exploits. Merging application and context models with a vulnerability pattern library — dynamic and expandable to accommodate new Common Vulnerability and Exposure (CVE) entries — the tool offers limitless extensibility. We evaluated the tool on the GHERA benchmark and found that at least 38% of the vulnerabilities in the benchmark can be modelled and detected.

This work underscores the pivotal role of context in Android security testing and presents a solution for vulnerability identification through the integration of MBST and DSLs.

***Index Terms*—Android Application Security, Vulnerability Detection, Context-Awareness, Model-Based Security Testing, Domain Specific Language**

I. Introduction

Over 5 billion people use a mobile phone today, meaning that 2/3 of the world population is prone to an attack via a mobile application [1]. Research [2] shows that 74.1% of Android apps have security flaws, with 25% considered as high severity. In mobile applications, we face one main difference with traditional software security testing, that is the heterogeneity of the context surrounding the application's execution [3].

In our work, we focus on the usage of Model-Based Security Testing (MBST) for mobile apps, taking into account the application behaviour and its context. We develop an approach that is supported by a tool, named DroidSecTester.

DroidSecTester, an open-source toolchain for model-based context-aware mobile application testing and security testing [4]. It aims to provide an extensible tool for secure Android development, using a contextual model to evaluate coverage based on chosen criteria. Its latest extension, *VPatChecker*, generates abstract exploits for studied applications using contextual model coverage and predefined vulnerability patterns.

VPatChecker aims at allowing developers to detect vulnerabilities on their application during development. The vulnerability patterns are separated from the application and can be designed by pentesters each time a new vulnerability is revealed, making the tool extensible over time.

DroidSecTester makes use of Domain Specific Languages (DSLs) as models for their potential, both in terms of automation and usability. We implemented the toolchain in Java and Xtext from the Eclipse Modelling Framework. The tools, installation procedures and associated documents can be found on GitHub [4]. We think that with its newest extension, *DroidSecTester* can provide an interesting set of tools for developers and pentesters, and if widely used would improve the Android development ecosystem.

In the following sections, we delve into a comprehensive exploration of our work, beginning with an analysis of related work and background in Section II, followed by an in-depth presentation of our methodology in Section III. Subsequently, we detail in Section IV the design and implementation of DroidSecTester, accompanied by an evaluation of its performance in Section V. We then discuss the limitations we encountered and outline potential directions for future research

in Section VI. Finally, we draw our insights together and present a conclusive summary in Section VII.

II. Related Work and background

The lack of user awareness with regards to security issues leads to security weaknesses in mobile apps. More and more applications are aware of the computing environment in which they run, and adapt their behaviours according to computing, usage, physical, or time contexts [5]. This is part of context-aware computing and brings new types of vulnerabilities, related to context information, such as:

{**1**} Integrated third-party code such as advertisement code exploits [6].
{**2**} Insecure communication such as TLS-related vulnerabilities.
{**3**} Untrusted sensors [7].
{**4**} Sensor APIs and backward compatibility [8].
{**5**} Unsecure environment [9].

Traditional methods to detect vulnerabilities in applications are based on security tests, carried out before the deployment phase. The increasing number of attacks targeting private user data [10] [11] and the lack of context-related security requirements analysis in the early stages of the application life cycle [12] make security tests targeting context-related vulnerabilities necessary. Some research works on mobile application testing based on the context information analysis exist [13] [14]. For example, in the paper by Majchrzak et al. [15], the authors cover the need to handle frequent context changes in Android application testing.

Model-based approaches to formal verification such as VAnDroid [16] and PermDroid [17] have been created to prevent vulnerabilities in earlier stages of development, the former focusing on inter component vulnerabilities and the latter on permission related vulnerabilities.

Context-driven security testing has been introduced in the papers of Shebaro et al. [18] with their work on access control system and the work by Railkar [19] et al. proposes an attack taxonomy for Context Aware Proactive Systems (CAPS), modelling the attacking activities directly.

In a more recent work, Adwan [5] introduces a method for context-aware mobile application testing. His work led to the creation of ConTest in DroidSecTester by introducing the first iteration of CDL and CDML. However, these works focus mainly on application functionalities and not on their security. The complexity of managing security issues linked to the context information makes the development of targeted security tests a mandatory requirement in order to guide the development of relevant mitigation measures. This is the main purpose of this work.

The initial work introduced by Adwan [5] leveraged a Model Based Testing (MBT) methodology in order to create a process of testing using the advantages that models bring. The MBT methodology brings forward the idea of avoiding the complications of usual testing methods by testing a model of the System under Test (SuT) instead of the model. In our case we are able to focus on the context of the application while limiting the combinatorial explosion created by testing complex processes.

III. Methodology

Our process extends the work by Adwan [5] by adding security information in the early application model, with the objective of being able to detect vulnerabilities related to function level exploits and information flow.

In order to introduce security testing mechanisms to the pre-existing work, we analysed existing vulnerabilities in the Android ecosystem. Our results led us to define a vulnerability in terms of information flow. This definition can be written as these two sentences :

- *A private source to a public sink*
- *A controlled source to a dangerous function*

This means that we want to be able to model flows of data on top of the context and behaviour models.

This methodology follows the MBST principle, a direct extension of MBT. We define MBST as *a methodology to test security requirements efficiently*. More specifically, we use MBST to carefully model the context of an application and to abstract most of the application in order to gain efficiency and create a process that is able to cover different contextual events. One of the difficulties in analysing context when trying to detect vulnerabilities is that due to the situation of mobile phones, there are much more contextual data than in classic software, so the cost is very high. Using models specifically for context representation allows us to test the context while keeping the processing cost low.

For the purpose of our project, we derived our context model from the definition of context given by Gomez et al. [20] and Almeida et al. [13]:

- 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 agregation of contexts or specific context values.

Our methodology is supported by DroidSecTester, which is introduced below.

IV. DroidSecTester: Design and Implementation

In this section, we outline the toolchain's general steps in Subsection IV-A, introduce the base application model in IV-B, cover its security enrichment in IV-C, and the context model in IV-D. In Subsection IV-E, we introduce test generation via ConTest subsection, concluding with our latest addition, *VPatChecker* in Subsection IV-F.

A. DroidSecTester

Figure 1 represents the toolchain of *DroidSecTester*. The toolchain works as a four step process.

(**1**) First, a Hierarchical Finite State Machine (HFSM) of the studied application is built by the developer using the CDML (Context-Driven Modelling Language) DSL provided by the toolchain.

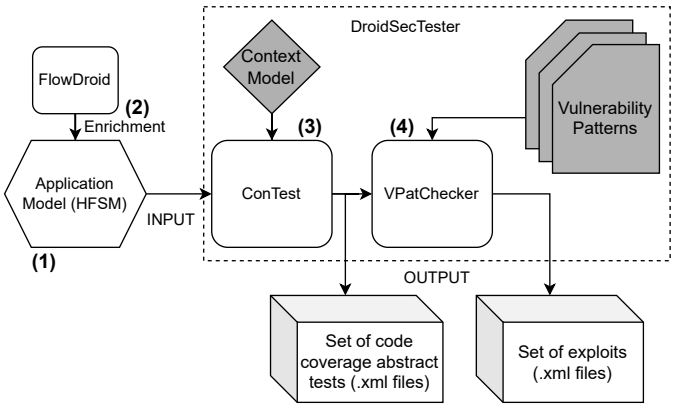

Figure 1. Global architecture DroidSecTester

(**2**) In order to contain key information on data flows on the studied application, the second step makes use of FlowDroid [21] to detect the flow of execution of both input values and private values. This step is important in order to provide a wider range of vulnerability detection for the last step. We follow input values to be able to control them when generating exploits.

(**3**) Step three involves using ConTest, a Context-Aware test generator that creates tests to achieve model coverage. The default coverage criteria includes all transitions and contexts in the model, and a predefined generic context model (CDL) in the toolchain is provided to simplify the application model (CDML). This step generates a set of abstract tests in *XML* format, which can be translated into real code for code coverage and used in the next toolchain step.

(**4**) The final stage involves security testing with the use of *VPatChecker*. This step examines the coverage tests to identify if they can be converted into exploits for previously defined vulnerability patterns. When a match is detected between a code coverage abstract test and a pattern, the inputs and outputs of the corresponding tests are adjusted to create an exploit. The resulting abstract exploits are saved in *XML* format and can be transformed into real exploits to validate the existence of the vulnerability.

B. Application model

The base model defined in step 1 and step 2 (cf. IV-A) in Subsection IV-A is written in a DSL named CDML. This model follows a Hierarchical Finite State Machine (HFSM) representation of the Android application. By separating each component of an application into a Finite State Machine, we are able to have a representation that is able to simplify a complex code. A complete example called **ExampleCDML.cdml** can be found on GitHub: https://github.com/Myshtea/VPatChecker/.

In Listing 1, is shown the representation of a single component as a State Machine. Super states are connected to other State Machines, in this example, it is connected to the *SEND_MESSAGE_ACTIVITY_SM* State Machine found in Listing 2. Normal states (called atomic states) work as usual states in a State Machine.

```
statemachine ABSTRACT_SM {
    state START {
        transition on APP_STARTED -> SEND_MESSAGE_ACTIVITY
    }

    super state SEND_MESSAGE_ACTIVITY abstracts
            ↪ SEND_MESSAGE_ACTIVITY_SM {
        transition on TERMINATE_BUTTON_CLICKED -> EXIT
        transition on SUCCESS -> HANDLE_SUCCESS
    }

    state HANDLE_SUCCESS {
        transition on BACK_BUTTON_CLICKED ->
                ↪ SEND_MESSAGE_ACTIVITY
    }

    state EXIT
}
```

Listing 1. Extract from ExampleCDML.cdml: Statemachine (SM) containing the atomic states Start, Handle_Success and Exit, and the super state Send_Message_Activity

```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {
    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED -> SHOW_ANSWER
    }

    state SHOW_ANSWER
}
```

Listing 2. Extract from ExampleCDML.cdml: Statemachine (SM) containing the atomic states SEND_MESSAGE and SHOW_ANSWER

When we start in a super state, such as in the state *SEND_MESSAGE_ACTIVITY* of Listing 1, the execution flow continues to the related statemachine shown in Listing 2. Once this last statemachine has reached a final state, the execution flows continues with the transitions found in the initial super state, which in our example is *TERMINATE_BUTTON_CLICKED* and *SUCCESS*. The exported keyword found in Listing 2 shows that this state machine can also be called by another application and thus the execution flow of the application could start here.

C. FlowDroid enrichment

As explained in III, we found that looking at the incorrect/unsafe dataflow allowed to define a large amount of vulnerabilities.

In order to add this information to our process and as introduced in step 2 (cf. IV-A), we made use of FlowDroid [21], a static data flow analysis tool that parses the APK [1] built from the studied application and outputs flows between a set of source and sink functions. It's originally intended to be used with real sources and sinks (i.e: flows of private values into public sources) but we are able to control which functions are analysed by the tool to effectively follow streams to selected functions. Through this method, we are able to follow flows to functions that may be deemed vulnerable not only in terms of information leakage but also in their actions (such as privilege escalation for example).

With a custom script, we can visualise these flows and add them to our model.

In Listing 3 and Listing 4 we demonstrate the enriched versions of Listing 1 and Listing 2. In this case, FlowDroid found that our code stored the internet status in

[1]APK: Android PacKage is the format of Android applications

the state “HANDLE_SUCCESS” of the “ABSTRACT_SM” state machine and this could leak at least some part of its information in the state “SEND_MESSAGE” of the “SEND_MESSAGE_ACTIVITY_SM”. The resulting model is the one we call behaviour model (or CDML model).

```
statemachine ABSTRACT_SM {
    state START {
        transition on APP_STARTED -> SEND_MESSAGE_ACTIVITY
    }

    super state SEND_MESSAGE_ACTIVITY abstracts
        ↪ SEND_MESSAGE_ACTIVITY_SM {
        transition on TERMINATE_BUTTON_CLICKED -> EXIT
        transition on SUCCESS -> HANDLE_SUCCESS
    }

    state HANDLE_SUCCESS {
        transition on BACK_BUTTON_CLICKED ->
            ↪ SEND_MESSAGE_ACTIVITY
        dataflows {
            source internet_status
        }
    }

    state EXIT
}
```

Listing 3. Extract from ExampleCDML.cdml after enrichment: Showing the source of a dataflow, this source could leak the internet status

```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {
    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED -> SHOW_ANSWER
        dataflows {
            sink “log.d” ( source
                ↪ ABSTRACT_SM.HANDLE_SUCCESS.internet_status)
        }
    }

    state SHOW_ANSWER
```

Listing 4. Extract from ExampleCDML.cdml after enrichment: Showing the sink of a dataflow, this sink leaks the internet status

D. Context model

In order to generate the set of context-dependent abstract tests defined in step 3 (cf. IV-A), we leverage a CDL model. This model is not written by the developer but is generic, it defines the values that an application context can take. This model takes the appearance of a list of contexts with additional information such as which component provides the specific context value. The full file used to define the context called **ContextModel.cdl** can be found on GitHub: https://github.com/Myshtea/VPatChecker/.

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

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

Listing 5. Extract from ContextModel.cdl: Context model for internet connectivity

In Listing 5, is shown an extract of this file introducing the internet connectivity context and its different values (Defined by the Connectivity property). As seen in Listing 2, states can be **awareof** a specific context (in this case: INTERNET_CONNECTIVITY). This way, the context can affect the execution flow of the program.

The specificity of the **awareof** keyword is that it allows the context to directly affect the execution flow of the program. We are able to create an extension of a specific state that is “aware of” a specific context. In Listing 6 we show an extension (called adaptation) of the state *SEND_MESSAGE* of Listing 4 when the internet is disconnected.

```
adaptation for INTERNET_DISCONNECTED at SEND_MESSAGE {
    state SEND_MESSAGE {
        transition on SEND_MESSAGE_CLICKED -> HANDLE_ERROR
    }

    state HANDLE_ERROR{
        transition on BACK_BUTTON_PRESSED -> external
            ↪ SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE
    }
}
```

Listing 6. Extract from ExampleCDML.cdml: An extension/adaptation of the state SEND_MESSAGE when internet is disconnected

E. ConTest: Generation of tests

The first result of our methodology is the generation of context-aware tests by the ConTest subprocess as defined in step 3 (cf. IV-A).

In its current form, ConTest will generate the tests of each state machine by following the selected criteria: All-Transitions and All-Situations (Every context change). These simple criteria allow us to test every transition from state to state and every context change in the states that are context-aware, s the state **SEND_MESSAGE** in Listing 2.

Once every test of every state machine (called subpaths) is created following the criterion, we select every starting node this can be either the main entry point of the application or an exported component. We then aggregate the state machines with the other state machines from every starting node. This is where we leverage the real advantage of using HFSMs, separating the generation process in two parts, subpath generation and aggregation.

F. VPatChecker: Context in security

In step 4 (cf. IV-A), VPatChecker aims to dispatch the work between developers and pentesters. Developers create their application and its model while pentesters analyze new vulnerabilities and design patterns using VPat, a simple DSL written with Xtext. VPat provides a way to define vulnerabilities and build an open-source database of vulnerabilities, written abstractly to be used by security testing tools. The language allows for the definition of vulnerability patterns with context-dependency. For instance, the pattern in Listing 7 presents a fake vulnerability pattern which considers the function “vulnerableFunction” as vulnerable in Android API level 31 when its parameters are a private value and the static string “EXPLOITME”. VPatChecker detects vulnerabilities if it can provide the function “vulnerableFunction” with specific values and execute the exploit in Android API level 31.

Although VPat can define context-independent vulnerabilities, it can also define context-dependent ones that are specific to some values of the static or dynamic context. For instance, starting from Android API level 28, cleartext is disabled by default in network configurations (although it wasn’t before [22]). A minSdk version set too low can result in certain device versions being vulnerable when running the application.

In Listing 3 and Listing 4, is shown a vulnerability that we call **log.d leak**. This vulnerability prints some private value to the logs of the device. In this situation, we can write the generic vulnerability pattern shown in listing 8.

```
Vulnerability "vulnerableFunc EXPLOITME" {
    description "The function vulnerableFunction leaks data
        ↪ when the second parameter is EXPLOITME in
        ↪ android version 31"

    context {
        apiversion "31"
    }

    function {
        main Sink "vulnerableFunction" {
            parameter {
                private,
                static "EXPLOITME"
            }
        },
        Source private *
    }
}
```

Listing 7. Example of vulnerability pattern file (.vpat)

```
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 8. Log.d leak vulnerability pattern file (.vpat)

This pattern defines a context-independent pattern that is vulnerable when we send a flow of a private value to the log.d function (**private** is the name of the source, and the **wildcard** shows that any source is accepted).

G. Generic tests

Both ConTest and VPatChecker generate tests and exploits in XML format, which include every state and transition required to reach a specific execution state, as well as the necessary dynamic context changes. The *XML* results also include the static context in which the tests/exploits should be run, as well as the sources and sinks from FlowDroid [21] and required input controls to exploit a vulnerability. The *Examples* folder in the GitHub repository [4] contains sample tests and exploits such as the one presented in listing 8. I

V. Evaluation

The evaluation we performed for DroidSecTester was based on the Ghera [23] dataset. The evaluation provides an insight into the current state of the range of vulnerabilities that DroidSecTester is able to implement as a vulnerability pattern in order to generate the related exploit for a specific vulnerable application.

In order to correctly evaluate our tool, we had to modify some of the vulnerable demonstrators of Ghera. Our tool detects two types of vulnerabilities: Private data is leaked or a specific function is used (possibly with specific inputs). This led us to add small data leakages when the demonstrator simply printed “Hacked” or equivalent.

Table I
Summarised result of vulnerabilities detectable by VPatChecker on the GHERA benchmark

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%

Table I summarises the analysis we performed using the Ghera dataset. The result of the web subgroup is to be considered as being out of the scope of our project. The main reason being that we are unable to model JavaScript code in our vulnerability model. These vulnerabilities are still counted in the total as not being detected.

As shown in table I, a total of 38% (or 48% if we don’t count out of scope types) of vulnerability types can be implemented as patterns and then detected on the testbench in the current state of the vulnerability pattern definition. Some vulnerability types are well represented using our methodology, this is the case with the Crypto, System and Storage subgroups. This result is due to the nature of those vulnerabilities, which are more related to functions and behaviour, while vulnerabilities from the Network subgroup are less represented because they are tied to network configurations.

We think that this evaluation can grow for the better with additional work on the meta-design of the vulnerability patterns specifically on the definition of the static context in the vulnerability model.

VI. Limitations and future work

In the proposed DroidSecTester toolchain, *ConTest* and *VPatChecker* modules, are not yet mature enough to join a proper development process as of today. A few key points have been identified to be considered in the future development of the project.

- An automated process: The project currently asks the user to rewrite the abstract tests into executable tests or at least to read the abstract tests. This is not something that a professional developer may use. The creation of this automation process is the key point to reach a mature state of the project. We think that the usage of DSL makes this automation process possible and easy to implement.
- A database of vulnerability patterns: The project aims at using open-source processes to build up its vulnerability database, but no platform has been set up yet. In order

to provide a complete platform for both developers and pentesters this is an important part of the future work.
- A complete language to define vulnerabilities: A wider range of vulnerability detection can be achieved by enhancing the vulnerability pattern language VPat, the current version is simple and could advance to a higher stage if worked on.

VII. Conclusion

In the work of Adwan [5] has been developed a tool called *ConTest* (Context and Test). This tool supports a process that, through a model of the application and the contexts, allows a developer to generate abstract tests by following a selected coverage criteria. An extension of this work has been proposed, which instead of analysing an application, directly scans coverage tests for matches against patterns of vulnerabilities. This process has been named *VPatChecker* and is available on GitHub [4] [2] (For Vulnerability Pattern Checker). The end goal of this contribution is to separate a developer's work from the security tester's one. This is allowed by designing the tool so that the patterns are separated from the application code.

We think that the proposed toolchain provides a step forward in the field of MBST with its use of DSL for security testing. We have managed to show the possibilities of the presented methods for a complete tool in software security testing.

VIII. Acknowledgements

This work is supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02). Special thanks to Abdallah Adwan for his work on CDL and CDML which allowed this toolchain to be created and Karl Palmskog for his guidance during the project.

References

[1] S. Kemp, "Digital 2021: Global Overview Report," 2021. [Online]. Available: https://datareportal.com/reports/digital-2021-global-overview-report

[2] V. team, "State of Software Security Volume 12," *Veracode*, 2022. [Online]. Available: https://www.veracode.com/state-of-software-security-report

[3] Zameer, "Understanding Mobile Context Awareness," Jan. 2021. [Online]. Available: https://medium.com/ascentic-technology/understanding-mobile-context-awareness-887a9d380d21

[4] I. Baheux, "Vpatchecker," https://github.com/Myshtea/VPatChecker, 2023.

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

[6] T. Liu, H. Wang, L. Li, X. Luo, F. Dong, Y. Guo, L. Wang, T. Bissyandé, and J. Klein, "MadDroid: Characterizing and Detecting Devious Ad Contents for Android Apps," in *Proceedings of The Web Conference 2020*, ser. WWW '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1715–1726. [Online]. Available: https://dl.acm.org/doi/10.1145/3366423.3380242

[7] L. Gonzalez-Manzano, U. Mahbub, J. M. de Fuentes, and R. Chellappa, "Impact of injection attacks on sensor-based continuous authentication for smartphones," *Computer Communications*, vol. 163, pp. 150–161, Nov. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366420319095

[2] A complete walkthrough of how DroidSecTester is used is available at https://youtu.be/3bkMXv7MeG0.

[8] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Understanding the Evolution of Android App Vulnerabilities," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 212–230, Mar. 2021, conference Name: IEEE Transactions on Reliability.

[9] G. Averlant, E. Alata, M. Kaâniche, V. Nicomette, and Y. Mao, "SAAC: Secure Android Application Context a Runtime Based Policy and its Architecture," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, Nov. 2018, pp. 1–5.

[10] Y. Lotfy, A. Zaki, T. Abd El-Hafeez, and T. Mahmoud, "Privacy Issues of Public Wi-Fi Networks," May 2021, pp. 656–665.

[11] M. E. A. Tebib, M. Graa, P. André, and O.-E.-K. Aktouf, "A Survey on Secure Android Apps Development Life-Cycle: Vulnerabilities and Tools," Jul. 2023, vol. 16, p. 54.

[12] I. U. Haq and T. A. Khan, "Penetration Frameworks and Development Issues in Secure Mobile Application Development: A Systematic Literature Review," *IEEE Access*, vol. 9, pp. 87 806–87 825, 2021, conference Name: IEEE Access.

[13] 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, no. 1, p. 12, Dec. 2019. [Online]. Available: https://doi.org/10.1186/s13173-019-0093-7

[14] T. B. Nguyen, T. T. B. Le, O.-E.-K. Aktouf, and I. Parissis, "Mobile Applications Testing Based on Bigraphs and Dynamic Feature Petri Nets," in *Intelligence of Things: Technologies and Applications*, ser. Lecture Notes on Data Engineering and Communications Technologies, N.-T. Nguyen, N.-N. Dao, Q.-D. Pham, and H. A. Le, Eds. Cham: Springer International Publishing, 2022, pp. 215–225.

[15] T. A. Majchrzak and M. Schulte, "Context-dependent testing of applications for mobile devices," *Open Journal of Web Technologies (OJWT)*, vol. 2, pp. 27–39, 01 2015.

[16] A. Nirumand, B. Zamani, and B. Tork Ladani, "Vandroid: A framework for vulnerability analysis of android applications using a model-driven reverse engineering technique," *Software: Practice and Experience*, vol. 49, no. 1, pp. 70–99, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2643

[17] M. E. A. Tebib, P. André, O.-E.-K. Aktouf, and M. Graa, "Assisting developers in preventing permissions related security issues in android applications," in *Dependable Computing - EDCC 2021 Workshops*, R. Adler, A. Bennaceur, S. Burton, A. Di Salle, N. Nostro, R. L. Olsen, S. Saidi, P. Schleiss, D. Schneider, and H.-P. Schwefel, Eds. Cham: Springer International Publishing, 2021, pp. 132–143.

[18] B. Shebaro, O. Oluwatimi, and E. Bertino, "Context-based access control systems for mobile devices," *Dependable and Secure Computing, IEEE Transactions on*, vol. 12, pp. 150–163, 03 2015.

[19] P. N. Railkar and P. N. Mahalle, "Activity modeling and threat taxonomy for context aware proactive system (caps) in smart phones," *International Journal of Computer Applications*, vol. 70, pp. 26–31, 2013.

[20] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 88–99. [Online]. Available: https://doi.org/10.1145/2897073.2897088

[21] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[22] Google, "Network security configuration." [Online]. Available: https://developer.android.com/training/articles/security-config

[23] 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*, ser. PROMISE. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 43–52. [Online]. Available: https://doi.org/10.1145/3127005.3127010