A Modeling and Static Analysis Approach for the Verification of Privacy and Safety Properties in Kotlin Android Apps

1st Bara' Nazzal
School of Computing
Queen's University
Kingston, Canada
21bn7@queensu.ca

2nd Manar H. Alalfi
Department of Computer Science
Toronto Metropolitan University
Toronto, Canada
manar.alalfi@torontomu.ca

3rd James R. Cordy
School of Computing
Queen's University
Kingston, Canada
cordy@queensu.ca

***Abstract*—The safety and privacy of medical devices is critical, as they directly affect the health of users and handle sensitive personal data. Ensuring that these devices meet safety and security standards is essential, especially with the rise of do-it-yourself solutions such as open-source artificial pancreas systems (APSs) for insulin delivery. In this work, we study AndroidAPS, an APS controller written in Kotlin, and propose an approach to detect safety and security issues. We develop a modeling and analysis framework for Kotlin applications that extracts a structural model and supports detecting logging vulnerabilities and ensuring the application of safety constraints. We conduct two experiments. The first examines logging behavior to check for privacy risks. Out of 3,059 logging instances, our tool identified 48 sinks that received 144 sensitive flows, with 68% precision due to coarse-grained flagging. The second experiment verifies that calculation-related values are validated against safety constraints before being set to the profile. We show that AndroidAPS generally adheres to its safety design properties, but it has one calculation-related value that is not explicitly validated at the plugin level and only partially validated earlier in the flow.**

***Index Terms*—Software, Privacy, Static analysis, Modeling**

I. Introduction

Ensuring the safety and privacy of medical device users is critically important due to the nature of these devices. They can directly impact the user's health and life, and they handle sensitive data that users expect to be treated with trust and confidentiality. An example of such a device is an artificial pancreas system (APS).

APS represent life-critical medical devices that are both widespread and readily accessible. These systems have been the subject of previous research and benefit from active, publicly available open-source software projects for the controller software component, making them a strong candidate for further investigation.

Software modeling techniques provide the ability to analyze and explore software states to ensure the establishment and preservation of safety and security properties as shown by Freitas et al. [8]. Previous APS research has generally concentrated on creating models for generic, theoretical devices and ensuring that these models satisfy safety and security requirements, this is shown in recent surveys by Silvia Bonfanti [3] and Nazzal et al. [17]. By contrast, we propose applying modeling techniques to existing, practical, open-source APS software by using model extraction and analysis techniques to validate and verify real-world software. In this paper, we aim to explore the following research questions:

- **RQ1:** Can a model suitable for verifying privacy and safety properties be automatically extracted from the source code of a production open-source APS application?
- **RQ2:** To what extent do current open-source APS applications adhere to established safety and privacy properties?

Our contributions are as follows:

- The design and implementation of an analysis framework for the verification of safety and security properties in Kotlin applications. It works by modeling the structural relationship from the code, which enables the analysis of its behavior using static analysis techniques.
- An empirical evaluation of AndroidAPS, a widely used open-source artificial pancreas system, focusing on its security with respect to privacy, and safety. We make sure that AndroidAPS avoids sensitive information leakage through logging vulnerabilities and we validate its safety by ensuring that it checks and applies safety constraints to values related to user profiles and insulin calculations.

II. Background

A. Medical Devices and AndroidAPS

Our research targets medical devices, which are safety-critical systems due to their direct impact on users' health and their handling of sensitive personal data.

These devices require certification to meet safety standards. This is increasingly needed with the rise of do-it-yourself (DIY) solutions, such as open-source artificial pancreas systems (APSs), which may bypass official certification. This highlights the importance of studying their safety and security.

APSs consist of a continuous glucose monitor (CGM), a controller, and an insulin pump. The controller either advises the user or automatically delivers insulin. Notable open-source APSs include OpenAPS [18], AndroidAPS [1], and Loop [14].

This paper takes AndroidAPS as a case study, which is an Android app users build and install themselves. It is written mainly in Kotlin with around 157,000 lines of code. We used *version 3.1.0* for our tests.

According to OpenAPS documentation [19] and Toffanin et al. [23], the controller regularly reads CGM data, queries the pump, and uses user inputs, such as carb intake and insulin sensitivity, to compute insulin on board (IOB) and carbohydrate impact. Based on this, it calculates basal and bolus insulin doses using different algorithms. AndroidAPS includes algorithms like Advanced Meal Assist (AMA) and SuperMicroBolus (SMB), described in AndroidAPS' documentation [2]. AMA accelerates temporary basal rates adjustment after meals, while SMB delivers smaller, rapid boluses, rather than adjusting basal rates, for faster insulin action and supports various configurations.

B. Software Modeling

Source code can be represented in different ways to facilitate analysis. Prior literature often uses formal methods [3], which apply mathematical techniques for software verification. This typically involves abstracting the software into a mathematically verifiable model, such as program states, logic, or events, and then verifying that the model adheres to the required specifications. This process can be used to ensure program correctness and compliance with security requirements. It also supports program simulation, aiding in debugging and detecting security flaws.

In our research, we design a custom framework for modeling and analysis. We use a modeling approach based on structural relationship between source code entities. In this approach, the model closely mirrors the source code and can be automatically extracted.

III. Methodology

Our framework is based on modeling the source code and applying static analysis techniques. The first step in our approach is to provide a model for Kotlin code. This will serve as the basis for our tool and analysis. The model we provide is based on the structural relationships between Kotlin entities. The main entities in Kotlin code that we are interested in are the following: Files, Packages, Classes, Objects, Interfaces, Functions, Variables, Values, and Properties. In our model, we track the structural relationships between these entities within the code. The relationship schema is explained in more detail in Table I.

As an example, a source code is provided in Listing 1. The code is a simple Kotlin program contained in a single file, *File 1*, which imports a package from *File 2*, defines one class, declares a property *A*, and implements the function *test*. The function declares a value **constant** and variables *x* and *y*, which are passed as parameters to the function *sum*, which is called in the assignment to variable *z*.

TABLE I
Model Relationship

Relationship	**Entity 1**	**Entity 2**	**Meaning**
Contains: - Entity1 - Entity2	File	Class Object Interface Functions	Kotlin source file may contain the following components
Associate: - Entity1 - Entity2	Package	File	Associate package with file and its contents
Imports: - Entity1 - Entity2	File	Package	Imports make other packages accessible in the current package. Declared with the syntax import
Declares: - Entity1 - Entity2	Class Object Interface Function	Class Object Interface Function, Variable Value Property	Entity 1 introduces Entity 2. Entity 2 specification is optional
Implements: - Entity1 - Entity2	Class Object Interface Function	Class Object Interface Function	Entity 1 provides Entity 2's specification and initialization
AssignedTo: - Entity1 - Entity2	Variables Values Properties	Variables Values Properties	Tracks variable through assignment statements
Calls: - Entity1 - Entity2	Functions Variables Values Properties	Function	Tracks function calls in assignment statements to variables or function calls within other functions
Passed: - Entity1 - Entity2	Variables Values Properties	Function	Variables are passed as parameters into a function
Returns: - Entity1 - Entity2	Function	Variables Values Properties	Functions can return values
CreatesInstance: - Entity1 - Entity2	Class Object Interface Function, Variables Values Properties	Class Object	A statement creates an instance of a class

The resulting model is represented in Graphical Modeling Language (GML) [9] in our tool, and a graph representation of the resulting model is shown in Figure 1.

Based on the above, we can generate an entity-relationship model between the different components in Kotlin. For example, each file imports packages and contains a class or an object. Each class has functions and may include property declarations. Each function contains statements that declare variables, assign values to them, or modify them, and may include calls to other functions or constructions of objects.

While we can easily map the top-level relationships, creating a full mapping of all relationships can be challenging due to the size of the application and the lack of a clear entry point. To address this, we utilize concepts from program analysis, specifically static analysis, as will be demonstrated

Listing 1. Example of Kotlin Code

```
!Start Filename "File1.kt"

import org.package2
package org.package1

class File1 {
    var A

    fun test(){
        val constant = 5
        var x = 3
        var y = x
        var z = sum(y, constant)
    }
}
!End Filename "File1.kt"
```

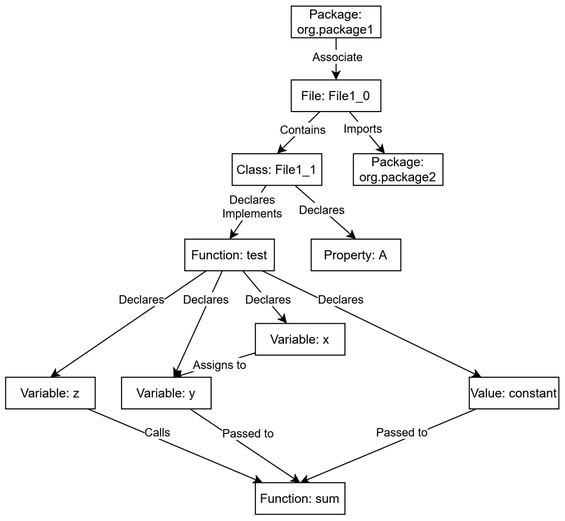

Fig. 1. Graph Representation of the Kotlin Model

in the following section.

A. Static Analysis

Program analysis can be done either statically, without executing the source code, or dynamically, during runtime. Static analysis requires access to the source code but has the advantage of providing more coverage and enables us to examine the structure of the code.

We use TXL [7], a source transformation language, to implement our framework. We use it to perform sink-source-based backward slicing of the Kotlin application. In this context, *sources* refer to potentially sensitive data, while *sinks* represent code locations where this data might be exposed. If there is a flow from the source to the sink that is not properly safeguarded, the flow is considered *tainted*. We used this concept to develop a TXL-based tool for analyzing Kotlin app source code to detect potential tainted flows.

Our TXL program consists of two main components responsible for abstract syntax tree generation and flow detection: **Grammar** and **Transformation Rules**.

The tool takes a Kotlin source code file as input and outputs the relevant parts of the code that contain identified data flows. We developed our own by referencing the Kotlin syntax documentation [10]. The grammar, similar to a context-free grammar, defines the language components, ranging from non-terminal annotations, headers, and imports to statements and terminal components. Accurate grammar is essential for precise analysis, as it governs the relationships between these components. Our grammar was validated on a large Kotlin dataset and is designed to be flexible, allowing for adaptation to other static analysis tasks. This grammar enables precise parsing and analysis of Kotlin code, forming the foundation of the analysis capabilities.

After merging the Kotlin files in the project, the resulting merged file is input to TXL and is processed by it. Using the provided grammar, the tool parses the input text according to Kotlin's reference syntax. If parsing fails at this stage, the process terminates and returns an error.

If parsing is successful, the TXL transformation rules perform the following tasks: **Sink identification**, where the source code is parsed and functions that correspond to the specified sink are marked. **Local backward tracing**, where lines containing variables that influence the sink are identified, such as parameters or assignments. **Global tracing**, where the rest of the application is scanned for invocations of the function containing the sink, as well as any other functions involved in the flow. This process is recursively repeated until all code segments that impact the relevant sink are identified. **Custom rules, code cleanup, and vulnerability reporting**, where custom rules are enforced, such as ignoring flows that are encrypted or otherwise mitigated; otherwise, the flow is retained. Finally, the tool performs a cleanup step by removing code sections that are not marked as relevant to the flow toward the sink.

This approach provides the flexibility to test for different properties and to add custom rules. For example, we can check for propagator functions within the flow and exclude flows that do not contain them. We can also check for functions which mitigate the vulnerability, and if they are present, the flow is marked as benign; if not, it is marked as potentially leaking sensitive data.

For instance, to detect leakage of sensitive data through a sink, we implement a main rule that identifies the sink calls across the codebase. A second custom rule then filters these calls to check whether they include sensitive variables as parameters. Moreover, we can chain multiple TXL transformations to preprocess the code or to use it as part of a pipeline with other commands. In general, our method is adaptable and can be customized to detect various sinks and sources.

After marking the data flow and adding tags, we generate a graph representation using Graph Modeling Language (GML), which models the data flows within the application. Running the TXL slicer on the Kotlin source code produces a marked Kotlin file that highlights both the sink statements and the statements leading to them. GML allows us to assign custom attributes to each node, such as whether it represents a source or a sink, or its level of impact. These impact levels can then be used to filter and generate data flows that exceed a specified impact level.

This answers **RQ1** and show our approach of combining modeling and static analysis which can be employed to detect various types of vulnerabilities and properties in the code. We will demonstrate this in two experiments on AndroidAPS.

IV. Privacy Experiment: Logging Vulnerabilities

In the first experiment, we demonstrate how our tool can be used to identify logging vulnerabilities. It is recommended that programmers avoid logging sensitive information as noted by the CERT coding standards [4] and as shown by the work Zhiyuan Chen [5] [6]. This is because logs can be obtained by an unauthorized party. In the case of medical devices, it is also important to have transparency with the user of the nature of the data that can be logged, even if it is used for debugging.

AndroidAPS makes extensive use of logs. It defines a custom logging function, *aapsLogger*, which is used 3,059 times throughout the application. We aim to examine how many of these logs contain sensitive information.

To detect logging vulnerabilities, we examine the relationships between entities leading to logging statements. Specifically, we aim to determine whether data passed to logging functions is sensitive and potentially subject to leakage. In this experiment, we treat any logging function as a potential sink. A more thorough examination would analyze the implementation details of the logging functionality itself to determine its risk level. However, this is considered out of scope since AndroidAPS relies on an external library for its main logging functionality.

The primary challenge in this experiment is identifying the sensitivity of variables passed to logging functions. Given the extensive use of logging and the large number of variables involved, manual inspection is impractical. Instead, we utilize the GML and flow reporting stages of our framework to automate this process.

Our framework supports a customizable approach to filtering entity relationships. It allows for both fine-grained and coarse-grained definitions of sensitive variables. For this experiment, we employed a coarse-grained approach by categorizing classes and files based on their potential to handle sensitive data. After manually reviewing the codebase, we labeled classes accordingly and tracked variable flows based on the class of origin, which determined their sensitivity. The sink flow is tracked through impactful files, while ignoring flows through ones that were flagged irrelevant.

1) Logging Test Results: We ran our tool using *aapsLogger* as a sink to trace whether any sensitive data flowed into the logging calls. We identified 144 potential flows across 48 sinks where a variable, originating from a sensitive context, was passed to the logger. Upon manual inspection, we determined that 56 of these flows (approximately 32%) were false positives. This rate is largely due to the coarse-grained tagging approach, so variables that are not sensitive but were contained in files that were flagged impactful, were included. This could be reduced by incorporating more fine-grained sensitivity flags.

An excerpt of the code is shown in Listing 2. In this example, the logger records the Total Daily Dose (TDD) value, which is considered sensitive information and should not be accessible to others without the user's consent.

Listing 2. AndroidAPS Logging

```
data class TotalDailyDose(
//...
val tdd = TotalDailyDose (timestamp = startTime)
tdd.totalAmount = tdd.bolusAmount + tdd.basalAmount
aapsLogger.debug (LTag.CORE, tdd.toString ())
```

However, it should be noted that this approach has its limitations. Due to its coarse-grained analysis, it is susceptible to false positives. When examined in more detail, as shown in Listing 3, we observe that the log records the value *result*, which originates from the *fromString* function in the *Bolus* class. Although the flow passes through impactful files, since data related to bolus values is typically sensitive, the specific value logged in this instance is merely a generated result code and does not contain sensitive information.

This experiment demonstrates the potential for using our approach to check for security concerns. We have shown that our framework can incorporate sensitivity flags, offering a flexible and extensible means of addressing additional vulnerability types. This answers **RQ2** regarding one privacy concern, which is logging sensitive data. Our analysis indicates that AndroidAPS makes extensive use of logging, some of which may involve sensitive data and thus warrant closer inspection. However, we acknowledge that our current file-level flagging strategy introduces false positives, suggesting the need for more fine-grained flagging to improve precision.

Listing 3. AndroidAPS Logging False Positive

```
fun fromString (name : String ?) = values ().firstOrNull
    {} ? : NORMAL
//...
val serviceUUID = UUID.fromString (GattAttributes.
    SERVICE_RADIO)
//...
result = rileyLinkBle.readCharacteristicBlocking (
    serviceUUID, radioDataUUID)
//...
aapsLogger.error (LTag.PUMPBTCOMM, "FAIL: got invalid
    result code: ${result.resultCode}
```

V. Safety Experiment: Checking Constraints

In the second experiment, we further examine the usage of our tool, this time looking into the safety of AndroidAPS. AndroidAPS, and the algorithm, oref0 underneath, depend on certain features to insure the safety of its usage. One of these properties is the enforcement of constraints to sensitive data. This problem can be converted into source-sink pattern problem, which our tool is capable of detecting in the Kotlin source code.

We look at areas of the application where sensitive values are set in the profile and used in calculations relating to the APS operations. The sources are defined as the parts of the code where the values are introduced, and the sinks are the part of the codes where the values are committed to the profile. The criteria for insuring safety is that the values are validated and confined to the safety limits when input and modified through the flow from the source and sink.

AndroidAPS can switch between two algorithms, *Advanced meal assist (AMA)* and *Super Micro Bolus (OpenAPS SMB)* which can be set by the user through GUI interactions. These algorithms are implemented as plugins that are connected with an adapter. The adapter communicates with a calculator file and sends the result. The point of interest for us is the data handling parts of the plugin and the adapter. Specifically where the plugin sets the profile data through a *setData* function which is later used by the calculations.

For our experiment, we are interested in the data handling part of the code, mainly happening in the plugin *setData* function. Our goal is to validate that all the values going into *setData* are checked before reaching the function. In other words, any flow going into the sink *setData*, should have a validation step. This can be done using our framework, by slicing the code with *setData* being the sink, then generating GML representation of the flows, flagging nodes that match the validation criteria. In the following sections, we will apply this approach in the two algorithms.

In the AMA plugin, there are 12 values that are set to the profile. These values are: *profile*, *maxIob*, *maxBasal*, *minBg*, *maxBg*, *targetBg*, *baseBasalRate*, *iobArray*, *glucoseStatus*, *mealData*, *lastAutosensResult.ratio*, and *isTempTarget*.

After running our test, with the sink *setData*, the source being the entry points where the values are input, we tested for verification using the queries *check*, *valid* and *verify*. We found that the values are either checked using *constraintsChecker* or *verifyHardLimits* functions, while there are some values that were not checked or only have a null check.

1) Constraint Checks: For *maxIob*, *maxBasal*, and *lastAutosensResult.ratio*, the test matches through the verification function *constraintChecker*. This check can limit the value further and provide reasoning for applied restrictions. The flow traces these functions back to the *Constraints* interface, with the functions implemented in *constraintsChecler* and *SafetyPlugin*.

2) Hard Limits Check: For *minBg*, *maxBg*, *targetBg*, and *baseBasalRate* the test matches through the verification function *verifyHardLimits*. This check is implemented in the *HardLimits* class where constants are provided as hard limits depending on the age of the user, whether they are a child, a teenager, an adult, a resistant adult, or pregnant. The values are called using corresponding functions and the limits are verified through *verifyHardLimits*.

3) Null Check: For *profile* and *glucoseStatus*, the values are only checked to be not null. For *profile*, and *glucoseStatus*, the values are taken from a *get* function, and thus the plugin does not modify the data at that stage. For the profile, this is the object of the targeted profile set by the user, including the profile ID. In itself it is not a value that is used calculation, so validating it to be selected and not null is adequate. On the other hand, the *glucoseStatus* is an input given by the source sensor and does not have an explicit lower or upper constraint by the plugin; the main check is that it is not missing and thus not null.

In the case of *IoBArray*, our tool did not detect an explicit checking step that matches our query in its assignment or before *setData*. Tracking the slice backwards we see that the flow comes from the *IoB Calculation plugin*, through a series of functions and the sources being the functions *calculateIobFromBolusToTime* and *calculateIobToTimeFromTempBasalsIncludingConvertedExtended*. Tracking the values through this functions, *calculateIobFromBolusToTime*, does a validation step to the values being processed, while *calculateIobToTimeFromTempBasalsIncludingConvertedExtended* does not have an explicit check that matches our query. It should be noted that this does not mean that the values are not safe. It shows that the function is not being explicitly validated before being set in the profile.

The other value that is not checked is *isTempTarget*. This is a binary value that is turned on or off in the AMA plugin and thus does not require an additional checking step.

We also extended the experiment to the other algorithm plugin, the SMB plugin. Our tests show that the plugin follows and matches the same patterns for validation that the AMA plugin had which were already discussed in the previous section. It introduces four new values: *getMealDataWithWaitingForCalculationFinish*, *smbAllowed.value*, *uam.value* and *advancedFiltering.value*, these are checked through *constraintChecker*.

This answers **RQ2** regarding one aspect of safety, which is the application of constraints to values used in calculations. Overall, we can confirm that AndroidAPS follows its design principles and provides mechanism to apply them through hard limits and constraints. Nonetheless, we see that there is one impacting variable, *IoBArray* in each plugin that is not explicitly validated before being set to the profiles; instead it relies on other plugins to do the calculation and validates the input data at one entry point. We also see that the calculation AMA and SMB plugins do not modify data in an unsafe manner, but some of the variables are not explicitly checked at the *setData* point.

VI. Related Work

Previous surveys by Nazzal et al. [17], John Majikes et al. [15] and Silvia Bonfanti [3] show the popularity of formal methods and modeling techniques and the exploration of applying them to the verification of medical software. However, one observation is that there is a gap in the research when it comes to applying these techniques to real software, such as do-it-yourself APS. Some of the recent contributions to addres the safety of do-it-yourself APS include the work of Chiara Toffanin et al. [23], whcih aims to simulate safety testing in silico, and Jana Schmitzer et al. [21] which utilized MATLAB/Simulink to speed the process.

Although there is academic research and many Android static analysis tools [13], very few addresses Kotlin directly. Existing static analysis tools, including SonarQube [22], PMD

[20], Polyspace [16], and CodeSonar [11], are primarily designed to detect general code issues, such as bugs, errors, and code smells, but do not give flexibility to address privacy and safety issues of a specific targeted app. Krishnamurthy et al. [12] explored the limitations of applying Java taint-analysis tools to Kotlin applications, and implemented some, but not all, solutions to the challenges. Our approach differs by targeting Kotlin directly, taking its grammar into consideration, which allows flexibility in using the tool to address a range of different problems.

VII. Limitations and Future Work

In this paper, we demonstrated our framework using a single application. We plan to publish additional results from testing other applications in the future.

While our study demonstrates the viability of using modeling and static analysis to address security and safety concerns, it also has limitations, particularly the level of expertise required to design effective tests and select appropriate sources and sinks. The framework currently relies heavily on the user's understanding of the application's structure and logic, which reduces accessibility for non-experts and third-party reviewers. In the future, automation and large language models could help with source and sink selection and rule definition, and the flexibility of our tool supports the integration of automatically generated configurations.

To our knowledge, this is the first study to focus on directly analyzing Kotlin Android apps from source code, specifically targeting AndroidAPS; as a result, there were no existing benchmarks for direct comparison. Although we manually evaluated the tool's performance, the absence of benchmarks presents a threat to external validity.

VIII. Conclusion

We found that modeling and static analysis techniques can be effectively combined and applied together to real-world medical applications. In particular, we demonstrated that it is possible to automatically extract a relational model directly from Kotlin source code, including AndroidAPS, using our TXL-based tool. Using static analysis, we can then track specified data flows.

We conducted two experiments: one focused on privacy by testing for logging vulnerabilities, and another on the enforcement of safety constraints. Our results show that, overall, AndroidAPS adheres to secure coding standards in most cases. However, we found that the extensive logging system may require further review to ensure private data is not exposed, with 144 potential data flows reaching 48 sinks. It is important to note that this result was obtained with a coarse-grained flagging approach and a precision of 68%. Additionally, the logging library used is third-party and falls outside the scope of our current analysis. Regarding safety, we found that the application validates most data inputs related to insulin delivery; however, one data path did not match the expected validation patterns and would require the developer's discretion.

References

[1] AndroidAPS. Androidaps. androidaps.readthedocs.io/. Accessed: 2025-02-20.

[2] AndroidAPS. Key aaps features. https://androidaps.readthedocs.io/en/latest/DailyLifeWithAaps/KeyAapsFeatures.html. Accessed: 2025-02-20.

[3] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. A systematic literature review of the use of formal methods in medical software systems. *Journal of Software: Evolution and Process*, 30(5):e1943, 2018.

[4] CERT Coordination Center. CERT Android coding standard - drd04-j. https://wiki.sei.cmu.edu/confluence/display/android/DRD04-J.+Do+not+log+sensitive+information. Accessed: 2025-02-20.

[5] Zhiyuan Chen. A comprehensive study of privacy leakage vulnerability in android app logs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2510–2513, New York, NY, USA, 2024. Association for Computing Machinery.

[6] Zhiyuan Chen, Soham Sanjay Deo, Poorna Chander Reddy Puttaparthi, Yiming Tang, Xueling Zhang, and Weiyi Shang. From logging to leakage: A study of privacy leakage in android app logs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2484–2485, New York, NY, USA, 2024. Association for Computing Machinery.

[7] J.R. Cordy, C.D. Halpern, and E. Promislow. Txl: a rapid prototyping system for programming language dialects. In *Proceedings. 1988 International Conference on Computer Languages*, pages 280–285, 1988.

[8] Leo Freitas, William E. Scott, and Patrick Degenaar. Medicine-by-wire: Practical considerations on formal techniques for dependable medical systems. *Science of Computer Programming*, 200:102545, 2020.

[9] Gephi. Gml format from gephi's documentation. https://gephi.org/users/supported-graph-formats/gml-format/. Accessed: 2025-03-29.

[10] Google. Kotlin docs - grammar. https://kotlinlang.org/docs/reference/grammar.html. Accessed: 2025-02-20.

[11] GrammaTech. Codesonar. https://www.grammatech.com/codesonar-cc. Accessed: 2025-02-20.

[12] Ranjith Krishnamurthy, Goran Piskachev, and Eric Bodden. To what extent can we analyze kotlin programs using existing java taint analysis tools? In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 230–235. IEEE, 2022.

[13] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of Android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[14] Loop. Loop. https://loopkit.github.io/loopdocs/. Accessed: 2025-02-20.

[15] John J Majikes, Rahul Pandita, and Tao Xie. Literature review of testing techniques for medical device software. In *Proceedings of the 4th Medical Cyber-Physical Systems Workshop (MCPS'13), Philadelphia, USA*, 2013.

[16] MathWorks. Polyspace. https://www.mathworks.com/products/polyspace.html. Accessed: 2025-02-20.

[17] Bara' Nazzal, Manar H. Alalfi, and James R. Cordy. A survey on the verification and validation of artificial pancreas software systems. In *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–9, 2024.

[18] OpenAPS. Openaps. https://openaps.org/. Accessed: 2025-02-20.

[19] OpenAPS. Openaps reference design. https://openaps.org/reference-design/. Accessed: 2025-02-20.

[20] PMD. PMD. https://pmd.github.io/. Accessed: 2025-02-20.

[21] Jana Schmitzer, Carolin Strobel, Ronald Blechschmidt, Adrian Tappe, and Heiko Peuscher. Efficient closed loop simulation of do-it-yourself artificial pancreas systems. *Journal of Diabetes Science and Technology*, 16(1):61–69, 2022.

[22] SonarSource. Sonarqube. https://www.sonarsource.com/products/sonarqube/. Accessed: 2025-02-20.

[23] Chiara Toffanin, Milos Kozak, Zdenek Sumnik, Claudio Cobelli, and Lenka Petruzelkova. In silico trials of an open-source Android-based artificial pancreas: a new paradigm to test safety and efficacy of do-it-yourself systems. *Diabetes technology & therapeutics*, 22(2):112–120, 2020.