Received 23 May 2022, accepted 10 July 2022, date of publication 20 July 2022, date of current version 5 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3192562

APPLIED RESEARCH

An Automated Approach for Privacy Leakage Identification in IoT Apps

BARA' NAZZAL, (Member, IEEE), AND MANAR H. ALALFI, (Member, IEEE)
Department of Computer Science, Toronto Metropolitan University, Toronto, ON M5B 2K3, Canada
Corresponding author: Manar H. Alalfi (manar.alalfi@ryerson.ca)

ABSTRACT This paper presents a fully automated static analysis approach and a tool, Taint-Things, for the identification of tainted flows in SmartThings IoT apps. Taint-Things accurately identifies all tainted flows reported by one of the state-of-the-art tools with at least 4 times improved performance. Our approach reports potential vulnerable tainted flows in a form of a concise security slice, where the relevant parts of the code are given with the lines affecting the sensitive information, which could provide security auditors with an effective and precise tool to pinpoint security issues in SmartThings apps under test. We also present and test ways to add precision to Taint-Things by adding extra sensitivities; we provide different approaches for flow, path and context sensitive analyses through modules that can be added to Taint-Things. We present experiments to evaluate Taint-Things by running it on a SmartThings app dataset as well as testing for precision and recall on a set generated by a mutation framework to see how much coverage is achieved without adding false positives. This shows an improvement in performance both in terms of speed up to 4 folds, as well as improving the precision avoiding false positives by providing a higher level of flow and path sensitivity analysis in comparison with one of state of the art tools.

INDEX TERMS Internet of Things (IoT), security, smart homes, static analysis.

I. INTRODUCTION

Today, more devices such as everyday utilities, home appliances, cars and other items are being embedded with software and are getting connected through the internet, giving rise to the concept of Internet of Things (IoT). While this technology brings with it lots of advantages, it also opens the door for many vulnerabilities, making the study of the security aspects of this technology very crucial.

One of the main concerns in the field of IoT is the potential risk of sensitive data leaking. And with the increasing popularity of the technology, tackling this issue becomes more necessary. By their nature, IoT apps and devices communicate through the internet, messages and notifications, but with bad coding practices this could end up posing a serious risk of exposing the users' private information. Furthermore, malicious applications could be specifically designed to hide their malicious behavior through undeclared breaches.

The associate editor coordinating the review of this manuscript and approving it for publication was Aasia Khanum.

In a report by the Open Web Application Security Project (OWASP) that listed top 10 IoT vulnerabilities for the year 2018 [24], shown in Table 1, it included the risk of insufficient privacy protection as number 6. This indicates the risk of apps using sensitive information in a non secure manner or without permission. With the vulnerability of insecure network services being number 2 on the list, this increases the importance of providing a way to track private information within the app and detecting whether the information can be sent over networks, whether as messages, notifications or through the internet, which might not be secure.

The scope of our problem is detecting the potential data leaks in IoT applications in the form of tainted data flows from tainted sources, which are variables or parts of the code containing sensitive information, to sinks, which are functions that can leak the information. This happens when certain variables are communicated or pushed through a channel which can be compromised. This can be caused either by carelessness by the programmers or intentionally by an attacker. Programs can falsely provide descriptions and ask for permissions to do certain functionalities which might

TABLE 1. OWASP IoT Top 10 Vulnerabilities [24].

I1 Weak, Guessable, or Hardcoded Passwords
I2 Insecure Network Services
I3 Insecure Ecosystem Interfaces
I4 Lack of Secure Update Mechanism
I5 Use of Insecure or Outdated Components
I6 Insufficient Privacy Protection
I7 Insecure Data Transfer and Storage
I8 Lack of Device Management
I9 Insecure Default Settings
I10 Lack of Physical Hardening

violate users' privacy without them knowing. This can be done either maliciously or due to bad programming practices. We aim in this research to deal with this issue and provide an approach to check the source code for cases where such leakage could potentially happen. Our approach provides a quicker core analyzer as well as adding sensitivities as modules which can help detect the parts in the code that contains tainted flow and potential leakage. This could help the developer or the reviewer to scrutinize them further and put safety measures in place if needed.

This paper is an extension to our previous short paper [29], where we briefly presented an efficient and scalable static analysis approach and tool, Taint-Things, to identify information leakage in smart things apps. The approach provides security auditing reporting via computing and presenting tainted flow security slice directly from the code using an inductive transformation paradigm [19].

And the main contribution of this extended paper is:

- Extends Taint-Things with flow-, path-, and context sensitivity analysis and that to improve the tool's Precision and Recall.
- An experiment that evaluates Taint-Things Precision and Recall after adding flow-, path-, and context sensitivity analysis.
- We demonstrate that static analyses on IoT can be pushed to be quicker and cheaper by doing the analysis directly on the source code, with the right tools and without needing a lot of intermediate representation. This gives us the ability to explore ways to further increase in precision with less cost.

Our approach and tool can be used to give more transparency to the user in terms of applications' functionality by reporting tainted flows which are potential areas in the code that contain and may leak sensitive information, as well as providing developers and reviewers with a beneficial tool which can automate the process of detecting tainted data flows. Our approach reports potential vulnerable tainted flow in a form of a concise security slice, that is the relevant parts of the code containing the flow. Our approach and tool could provide security auditors with an effective and precise tool to pinpoint security issues in SmartThings apps under test.

In the following section, Section II, we provide a background and an overview of some the concepts related to our research, which includes static analysis, sources and sinks,

```

   definition(
      name: "Sample App",
      //...
   )

   preferences {
      section("User input example:") {
	input "phone", "number", title: "User inputted
	     number"
      }
   }

//..
   def initialize() {
      subscribe(phone, "number", eventHandler)
   }

   def eventHandler(evt) {
      sendSms "Your value is: $phone"
   }

   def closureExample(){
      def List numberList = [1, 2, 3]
      numberList.each{print it}
   }

   def foo() {...}
   def bar() {...}

   def callByReflection(){
      def methodName = "foo"
      $methodName
   }
```

Listing. 1. SmartThing app structure.

and sensitivities. In Section VI, we provide a literature survey of publications related IoT security as well as some relevant research done on static analysis in Android apps. In Section III we demonstrate our approach and its implementation. The following Section IV we explore ways for adding more precision by handling flow, path and context sensitivities, respectively and evaluating them and on Section V we present an evaluation for our tool and the approaches that add precision.

II. BACKGROUND

The scope of this paper is the issue of privacy leakage in IoT apps. We present a method of detecting potentially leaky IoT apps, illustrated by SmartThings apps, by analysing their source code, and trying to see ways of improving the analyses. To start off, we provide some of the important concepts that are used within this study.

A. THE PLATFORM

IoT has many frameworks, platforms and vendors, and while they might differ in the way they handle permissions and granularity, they share key concepts. Such platforccms include: Samsung's SmartThings [27], Apple's HomeKit [3], openHAB (open source) [23], Vera Control's Vera3 [10], Google's Weave/Brillo [15], and Open Connectivity Foundation's AllJoyn [14].

Celik et al [8] point to five IoT specific challenges when it comes to the security of the platforms. Namely, the issues of physical channels, simulation and modeling, test generation, multi-app analysis, and interaction between devices and platform services.

Physical channels is an issue because IoT devices control physical devices, which puts additional security risk through the physical processes. This can be through side-channel leaks, health related risks, and risk from indirect interactions.Side-channel leaks can happen for example when an adversary use the changes in an IoT device to infer whether someone is at home or not, Health related risk is a serious issue with IoT devices, since they control things like temperature or sound. IoT could also be indirect controlled, for example, by altering the environment around it, one can control certain devices.

Simulating IoT platforms is also challenging, since IoT devices often communicate with each other and act as a complex system. This, and the fact that IoT devices interact physically with their environment makes modeling and simulation challenging. This can affect how well one can study the security aspect of the whole system.

Additionally, since IoT is a new field, systematic and automatic test generation is still an issue that is not explored well. This is especially important for dynamic analysis.

IoT apps can also interact with each other; their events could be tied to each other or they can interact with the same device at the same time. This proposes a challenge when analyzing the apps and makes it necessary to consider the behavior of multiple apps at once.

Furthermore, IoT devices can interact with other platforms, such as network APIS or authentication services. This adds to the complexity of the analyses where the platform service and its interactions should be considered in the analyses.

We chose to start our study with a focus on Samsung's SmartThings because it is one of the more mature platforms with a good user base and it shares the important principles with the other platforms. SmartThings has three components: the hub, the apps and the cloud back-end. For our research, we are concerned with the apps and potential vulnerabilities in their programming. The scope of our research is thus specific to challenges relevant to the app side rather than the whole system.

SmartThings apps have security measures such as privilege separation, secure storage and apps are written in a sandbox environment, where features are limited for more security. Nonetheless, the platform raises some security concerns such as WebServices where HTTP endpoints get exposed and the use of call by reflection. This can propose a vulnerability if combined with over-privilege; an attacker can use this to execute command injection attacks. There is also little restriction on the communication abilities through the internet or through SMS, which can be used to leak sensitive information

One of the challenges when studying the system is the fact that it is closed-source, uses a proprietary environment for the execution and the system does not have publicly available APIs to obtain binaries. This makes dynamic analysis hard.

The apps in SmartThings are written in Groovy, a language like Java, and developed in a sandbox which limits it to specific functionalities relevant to IoT development. The usual structure of a SmartThings app is comprised of the following sections: definitions, preferences and the events/actions sections.

The definition section is where the application's name, description, category and other information are described. The preferences section is where permissions are defined for different devices as well as user inputs. Event/action sections define the methods which will perform actions required whenever an event is triggered.

SmartThings apps are event-driven by design, like Android applications, they do not have main functions. Instead, they have events that triggers the method calls. Some of the language-specific unique features include closures and call by reflection. Closures can be used to loop and perform action on a defined list of elements and call by reflections allows for the calling of methods by using their name in strings. Listing 1 shows an example of the structure of a SmartThings app as well as examples of closure and call by reflection usage.

B. STATIC ANALYSIS

Program analysis can be used to solve different problems, whether it is checking for the correctness of a program, finding ways to optimize it or improving its security. Under the umbrella of security, program analysis can be used with IoT apps to detect sensitive data leakages. It can be done either static, without executing the source code, or dynamic, during run time. 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, while dynamic analyses is limited by the scope of the code being executed.

When it comes to analysis tools, we care about certain attributes such as soundness and completeness. Where soundness deals with the correctness of the reports and completeness deals with covering all what's there to report. Measures used to evaluate correctness and completeness are precision and recall. Precision gives an idea of how many false positives are being reported from the total positives and Recall gives an idea about completeness by calculating how many true positives are reported out of all the positive cases. Making a perfect static analysis tool can be an impossible task, so we depending on the task, we can try to achieve certain features with trade-off from others.

We use a static analysis approach to tackle the problem of data leakage in IoT programs, where we perform the analysis directly on the code without executing it. The goal of the analysis here is to detect the flow from sources of potential sensitive data to sinks which are potential data leakage points in the code. If a flow contains sensitive information, we consider it tainted and we report it.

Different factors might contribute into the analysis precision. Precision being the value of avoiding false positives. In the following subsections, we present an overview of patterns that exist in IoT programs which can affect precision and how an analysis could takes them in consideration, making it a sensitive one:

1) FLOW SENSITIVITY

A flow sensitive analysis takes in consideration the statement execution order in the program as well as content change in variables. Listing 2 shows a sample code where the original variable *message* contains sensitive data and is passed to the sink *sendPush* but after it is changed, it is sent to *sendSms*. A flow sensitive analysis takes that into consideration and marks *sendSms* and *sendPush*'s data flows differently with *sendSms* being not tainted, since it accurately detects no sensitive data being sent through it, while an insensitive approach will confuse the two and mark *sendSms* as a tainted sink, since *message* contained sensitive data at one point.

```
// $sensitiveData defined
def message = "This contains $sensitiveData";
sendPush(message);
message = "no sensitive data";
sendSms(message);
```

Listing. 2. **Flow sensitivity example.**

To achieve flow sensitivity we can deal with each changed variable as a new variable all together; each variable would only be assigned a value once.

2) PATH SENSITIVITY

A path sensitive analysis takes the execution path in consideration. This is exemplified in how it deals with conditional statements; a path sensitive analysis would treat each conditional block as a separate path. Listing 3 illustrates this; a path sensitive analysis would only detect *sendSms* on line 5 as a tainted sink, while an insensitive approach will mark both *sendSms* and *sendPush*.

```
1   //$sensitiveData defined
2   def message = "no sensitive data";
3   if (condition) {
4      message = "This contains $sensitiveData";
5      sendSms(message)
6   } else {
7      sendPush(message)
8   }
9
10  if ($sensitiveData) {
11     def newMessage = "no sensitive data";
12     sendSms(newMessage)
13  }
```

Listing. 3. **Path sensitivity example.**

Another issue related to path sensitivity is implicit flows, which are flows that occur implicitly when a conditional statement depends on sensitive data. In Listing 3 *sendSms* on line 12 doesn't send any sensitive data, but it is in a conditional path that depends on testing sensitive data. In a path sensitive approach that takes implicit flows in consideration, the whole conditional block would be considered tainted and *sendSms* on line 12 would be declared a tainted sink, while a path insensitive approach or one that doesn't take implicit flows in consideration won't mark it as potentially containing or leaking sensitive information.

```
state.firstCounter = x
state.secondCounter = y
```

Listing. 4. **Field sensitivity example.**

```
1   //$sensitiveData defined
2   def takeAction() {
3      def message = "This contains $sensitiveData
           ";
4      def firstCall = returnMessage(message);
5      def secondCall = returnMessage("no
           sensitive data");
6      sendSms(secondCall);
7   }
8
9   private returnMessage(message) {
10     return message;
11  }
```

Listing. 5. **Context sensitivity example.**

3) FIELD SENSITIVITY

A field sensitive approach differentiates between the fields in an object as apposed to treating them as if they were the same. This is mainly relevant when dealing with global variables in SmartApps; global variables are stored as fields in an external object. Listing 4 shows an example of two global state variables. In a field sensitive analysis, each one would be modeled as its own, while an insensitive one will treat them as if they were the same variable throughout the program.

4) CONTEXT SENSITIVITY

Context sensitivity mainly deals with function calls and callbacks within the program. A context sensitive analysis identifies each call as its own and can track back to the context of the call.

Listing 5 shows a case where a method gets called twice. In a context insensitive analysis, both calls on lines 4 and 5 might be conflated, so the tainted return in *firstCall* will also be considered in *secondCall*, marking the flow tainted. A context sensitive approach on the other hand, doesn't confuse method calls and distinguishes each call site, so it won't mark the flow to *sendSms* tainted.

C. TXL

TXL [11] is a programming language that can be used for program transformation. It is a Functional/Rule-based hybrid language. A TXL program has two main parts, a grammar and transformation rules. The grammar part defines the syntax of the inputted program and allows for the TXL to recognize and parse its structure, while the transformation rules modifies it by replacement and alteration. Listing 6 is an example of a grammar for program made of a simple arithmetic statement; it specifies arithmetic expressions' priorities, their structures and their syntax. Listing 7 is an example of a transformation rule using the provided grammar to replace addition expressions with the result of the operation. The rule tries to find any expression that matches the provided pattern, and replaces the two numbers it matches with their sum. The *number [+ number]* is the built-in TXL expression for addition.

```
define program
   [expression]
end define

define expression
    [term]
   | [expression] + [term]
   | [expression] - [term]
end define

define term
    [primary]
   | [term] * [primary]
   | [term] / [primary]
end define

define primary
    [number]
   | [term]
   | ([expression])
end define
```

Listing. 6. **TXL grammar.**

```
include "grammar.grm"

rule main
   replace [expression]
     N1 [number] + N2 [number]
   by
     N1 [+N2]
end rule
```

Listing. 7. **TXL transformation.**

D. TESTING PLAN

To test our we work we conduct a comparative analysis with the state-of-the-art available tool on a dataset of the available SmartThings apps which are either provided officially or by third-parties by the community. We used a computer with similar specs to what the state-of-the-art the tool used to evaluate the speed and we look into the real time and CPU time taken to analyse the full dataset. We also manually check the reported flows by both tools by looking into the code and confirming that the results are sound.

For the added sensitivities, we also use a mutation dataset, which is based on the original dataset but adds patterns which contain patterns that require flow, path and context sensitive analyses to avoid false positives. We calculate the precision and recall for the results for our tool when adding the sensitive analysis and compare it to the original without it.

III. TAINT-THINGS

To implement our static analysis approach we use TXL [19]. The components of a TXL program are a grammar and a set of transformation rules. The grammar specifies the structure of the input while the rules specify the patterns that TXL will detect and replace to produce the output.

The challenge for the identification of a tainted flow is to trace dependencies backwards in the program to mark only those statements that can influence the marked tainted sinks.

Static taint analysis techniques, such as SAINT [9], build a dependency graph for the program and then use graph algorithms to reduce it to the tainted flow, which is mapped back to source statements afterward. The idea of using dependency graphs goes back to Thomas Reps. *et al.* [17]. However, in our approach we compute dependency chains directly, using the inductive transformation paradigm.

The approach uses a related TXL paradigm called cascaded markup. As Figure1-*"Taint Analysis Core Module"* demonstrates, the approach starts with marking sink statements, analyzing them recursively, finding and marking statements which directly influence them and then those that influence those statements, and so on until a fixed point is reached. This fixed point occurs when a potential tainted source is identified or when no more propagation can be done. As stated earlier, an example of potential tainted sources are user input identifiers that have been defined as part of the preference block in the SmartThings app. For the cascaded markup, we consider an assignment to a variable to be part of a tainted flow if any subsequent use of that variable exists in a sink.

The other markup propagation rules are simply special cases of this basic rule that propagate markup backwards into loops and if statements, around loops and out to containing statements when an inner statement is marked, Figure1-*Path Sensitivity Analysis*. The whole set of markup propagation rules is controlled by the usual fixed-point paradigm that detects when a tainted source is hit or when no more propagation can be done. The analysis as well takes care of taint propagation via methods calls and returns, Figure1-*context sensitivity Analysis*.

A. TAINT ANALYSIS-CORE MODULE

1) BUILDING A TXL GRAMMAR

The TXL grammar is considered as the most crucial component. It specifies the way TXL parses the program, like context free grammar (CFG), it analyzes the program into its components, which are the non-terminal statements, and further specifies the components of each statement down to terminal components, such as operators and operand.

Since our first step is to support SmartThings applications, which use Groovy, we must start with a grammar for it. Groovy grammar for TXL is not readily available, so we had to develop it ourselves. This could be done in two ways; since Groovy is similar in many ways to Java, we could override the available TXL Java grammar [12], tailoring it to Groovy, or we can start from scratch, writing a Groovy specific grammar.

We tried the first approach. Starting with the Java grammar and trying to parse Groovy applications, patching it up whenever it fails. We found out that as programs got bigger and used more Groovy specific features, the grammar was unable to parse them, resulting into errors, and was getting harder to maintain and patch. We developed and adapted 1400 lines of code and were only able to get 80% of the dataset to parse. The lack of semicolons as command separators and string interpolation were the biggest issues. This is because most statements in Java must end with semicolons while in Groovy, semicolons are optional and string interpolation has the challenge of dealing with call by reflection where you can call a method from within a string.

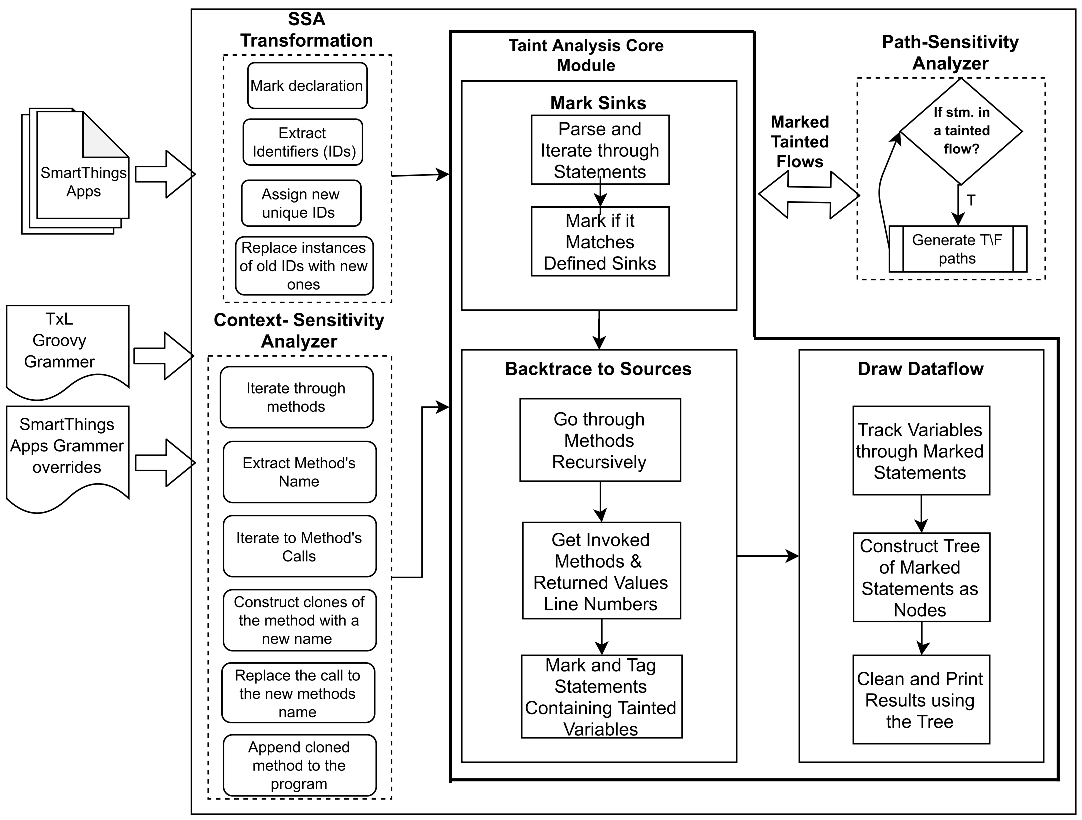

SSA Transformation
Mark declaration
Extract Identifiers (IDs)
Assign new unique IDs
Replace instances of old IDs with new ones
SmartThings Apps
TxL Groovy Grammer
SmartThings Apps Grammer overrides
Context- Sensitivity Analyzer
Iterate through methods
Extract Method's Name
Iterate to Method's Calls
Construct clones of the method with a new name
Replace the call to the new methods name
Append cloned method to the program
Taint Analysis Core Module
Mark Sinks
Parse and Iterate through Statements
Mark if it Matches Defined Sinks
Backtrace to Sources
Go through Methods Recursively
Get Invoked Methods & Returned Values Line Numbers
Mark and Tag Statements Containing Tainted Variables
Marked Tainted Flows
Path-Sensitivity Analyzer
If stm. in a tainted flow?
T
Generate T\F paths
Draw Dataflow
Track Variables through Marked Statements
Construct Tree of Marked Statements as Nodes
Clean and Print Results using the Tree

FIGURE 1. Taint-things approach architectural diagram.

```
int foo = 3
String b = "Foo is $foo"
```

Listing. 8. Example of problematic statements.

For the second approach, crafting a TXL grammar, we needed to start with an extended Backus-Naur form (EBNF) for Groovy. However, this isn't provided in Groovy's official documentation. Instead, the official repository offers parser and lex files generated from ANTLR [2] which can still give us an idea of the structure of the language. We used these as a starting point in our grammar inference process. With this approach, we developed 600 lines of code, and achieved 100% success rate in parsing our dataset.

2) IDENTIFYING SINKS

Taint sources are defined as variables or information which get passed in the application and potentially contain sensitive information. On the other hand, sinks are defined as functions which pass the information and potentially leak it. In our approach, we identify the grammatical forms for all potential sources and sinks for SmartThings IoT app described by Celik *et al.* [6]. Their list includes the following sources: device states, device information, location, user inputs, and State variables. Listing 9 shows an example of a defined user input in a SmartThings app.

Then we add those patterns to the grammar description of our analysis tool. Listing 10 provides an example of how we

```
    input "myLock", "lock", multiple:true, necessery:
        true
```

Listing. 9. Example of a user inputted variable in a SmartThings app.

```
define sink_name
    'httpDelete | 'httpGet | 'httpHead | 'httpPost
  | 'httpPostJson | 'httpPut| 'sendSms | '
      sendSmsMessage
  | 'sendNotificationEvent | 'sendNotification
  | 'sendNotificationToContacts|'sendPush| '
      sendPushMessage
end define
```

Listing. 10. TXL grammar definition for sinks.

grammatically define sinks functions used in SmartThings apps and identified by our tool. This approach provides flexibility by allowing us to add or remove potential sources or sinks; to do that, we can simply modify the grammar description for sources and sinks patterns. This change will not impact our analysis, since our rule based pattern matching engine will match by the pattern category rather than the individual patterns elements.

The first step of our approach is identifying the the sinks in a program. Since sinks are limited and usually less than the sources, it is generally easier to start with them and do backward tracing from there. In this step our tool parses the program and iterates through its statements. If a statement contains one of the defined sink functions, it labels the sink function. It also tags every variable and method declaration

```
def 32 initialize () {
   33 subscribe (33 themotion, " motion.active ",
        33 motionDetectedHandler)
}
def 36 motionDetectedHandler(36 evt) {
   def 37 message = " motionDetectedHandler called :
   $ 37 evt ";
   <sink> sendPush(38 message) </>;
   39
   theswitch.39 on ()
}
```

Listing. 11. **Sinks marking output.**

with their respective line numbers. Listing 11 shows an example based on a SmartThings apps and shows how it gets processed then at this step the sink, *sendPush(message)* gets marked with the label $<$ *sink* $>$ .. $<$ / $>$. Not that when the source code is inputted in TXL, identifiers are labeled with their line numbers.

3) DOING BACKWARD TRACING

In the second step of the analysis process, our approach tracks the variable that is being passed to the sink, traces it and marks the lines where it's contained all the way to a source. This is done by recursively going through the methods, extracting the line numbers of invoked methods and line numbers of returned values and using them to track and mark statements that contain a tainted variable, tagging it with the line number where it gets passed to.

In the following listing 12, the variable *message* is passed to the sink. And *message* has another variable *evt* which is passed to the function *motionDetectedHandler*. In the program initialization this function is subscribed with the source user input *themotion*. The subscribe statement is tagged with the line number where the function is defined.

4) IDENTIFYING TAINTED FLOW

In the next step of the analysis, the data flow gets drawn. To do that, the previously marked lines are tagged with the line numbers where their variables gets passed. Those get used in determining the feasibility of the flow by constructing a tree from the tagged statements as nodes. For the report, the code is cleaned by removing unmarked statements. The report is generated using the constructed tree for the line numbers containing tainted dataflow. Finally, variable and method declaration line numbers, tagging the source and presenting a summary of the data flow. The output from Taint-Things is shown in listing 13 where the numbers on top are the summary representing the line numbers of the flow and the source code gives the details to where it exists. An example of a real SmartThings app and the analysis steps is provided in the Appendix.

IV. PRECISION ENHANCEMENT

While we've seen that the core module of Taint-Things by itself is satisfactory in terms of detecting potential leakages as the tests on the dataset show, we wanted to see if there are ways to improve it by making it more resistant to potential false positives and more precise. Since it is fast and light in

```
preferences {
  12 section(" Turn on when motion detected: ") {
    input " themotion ", " capability.motionSensor ",
          13 required : true, 13 title : " Where ? "
  }
  ...
}
def  32 initialize () {
  <36>  33 subscribe ( 33 themotion, " motion.active ",
         33 motionDetectedHandler) </>
}
def  36 motionDetectedHandler( < > 36 evt </>) {
  def 37 message = " motionDetectedHandler called :
  $ 37 < > evt </> ";
  <sink>  sendPush( 38 message) </>;
  39 theswitch. 39 on ()
}
```

Listing. 12. **Back tracking output.**

```
33 36 37 38

def initialize() {
  <36 source  > subscribe(themotion, " motion.active "
        , motionDetectedHandler) < / >
}

def motionDetectedHandler( <37> evt < / >) {
  < 38 > def message = " motionDetectedHandler called
        : $evt "; < / >
  < sink > sendPush(message) < / >
}
```

Listing. 13. **Taint flow report.**

its performance, in this and the following sections, we want to explore how we can improve the precision by adding analyses sensitivities. This can help us minimize false-positives in certain cases. In this section we'll look into the case of flow sensitivity which is relevant in the case of variable reassignment throughout the code.

A. ADDING FLOW SENSITIVITY

A flow sensitive analysis takes in consideration the order of statement execution in the program as well as content change in variables. A flow sensitive analysis takes the variable's value changes into consideration while an insensitive approach does not. This can result into false positive reports. Currently, this is the case with both Taint-Things and SAINT, if we test the program in listing 14 they both mark the flow as malicious even though it is benign.

To avoid conflation in this case and achieve flow sensitivity, we can deal with each variable change as if it were a new variable; each variable would only be assigned a value once. This is the same concept used in static single assignment (SSA) form where each variable in the code is assigned a value once. Listing 15 shows Listing 14 in SSA form. We can process the inputted program and transform it into this form, which we later can chain into Taint-Things. This form will provide more precision when dealing with variable reassignments and will solve the issue of false positives resulting from them.

We have implemented this approach and written a TXL program that does this transformation. Using the groovy grammar, we analyse and detect all the assignment statements in a program as well as the identifiers. The program marks all declaration statements in the inputted program. It then gets

```
// $sensitiveData defined
def message = "This contains $sensitiveData";
sendPush(message);
message = "no sensitive data";
sendSms(message);
```

Listing. 14. Flow sensitivity example.

```
// $sensitiveData defined
def message1 = "This contains $sensitiveData";
message2 = "no sensitive data";
sendSms(message2);
```

Listing. 15. SSA form example.

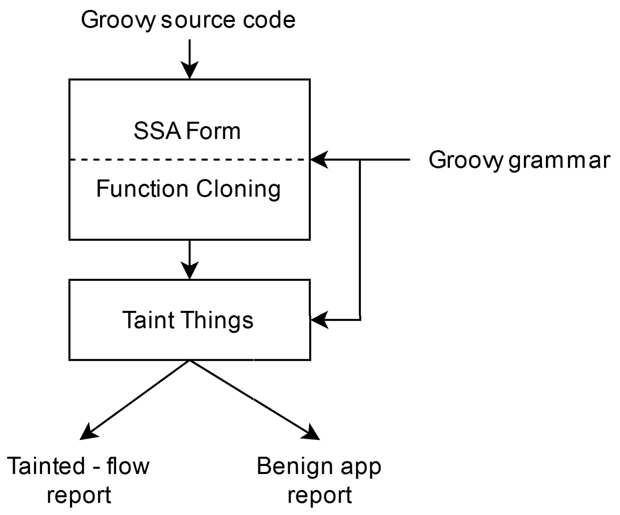


FIGURE 2. Taint-things with SSA or function cloning.

the variable's identifier from each declaration statement and assigns a unique variable identifier for that declaration statement. Finally, it replaces any instance of the old variable with the newly assigned identifier. Listing 16 shows the detected assignment statements in Listing 14. The result is then passed to the core analysis module as shown in Figure 2.

Algorithm 1 represents the SSA-form generation process. Starting from top to bottom. We lookup identifiers that match the one defined in the assignment statement, then we give that identifier set a new unique variable name. Repeating this processes, we go through each assignment statement and its associated identifiers. The result will be a program in SSA form, where every assignment statement gives us a uniquely named variable and avoids conflation in the case of reassignment. This can later be chained with our analysis tool, giving us a more precises analysis.

Going through the program's statements, if we encounter an assignment statement, we extract the variable *V* from the left-hand side (LHS). We assign a counter *C(V)* for it and use that to assign a new unique variable name for it. We then go through the rest of statements and if we find a statement containing the original variable, we replace it with the new variable. After that loop is finished, we go to the next assignment statement, and if it contained the same variable name, it will increment its counter, otherwise it'll be assigned a new counter.

Algorithm 1 Rename Variables in a Program to Adhere to SSA Form

Input: A program's source code
Output: Program's source code in SSA form

```
1  for each variable V do
2      $C(V) \leftarrow 0$
3  end
4  for each statement A in program X do
5      if A is an assignment then
6          get V from $LHS(A)$
7          $i \leftarrow C(V)$
8          $C(V) \leftarrow i + 1$
9          replace V by new Vi in $LHS(A)$
10         for statement B, from A to end do
11             if B contains a variable that is equal to V
                then
12                 $i \leftarrow C(V)$
13                 replace by new Vi
14             end
15         end
16     end
17 end
```

```
// $sensitiveData defined
def message = "This contains $sensitiveData";
message = "no sensitive data"
sendSms(message);
```

Listing. 16. Assignment statements.

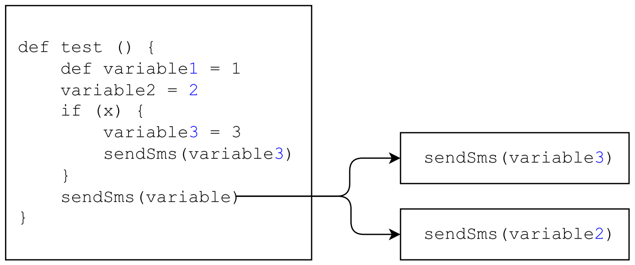
B. ADDING PATH SENSITIVITY

Following the previous section, we look in the case where the flow goes through a conditional statement, where it branches into different paths. We introduce path sensitivity and look in how we can achieve it by taking all possible branches in consideration.

A path sensitive analysis takes the execution path in consideration. This is exemplified in how it deals with conditional statements; a path sensitive analysis would treat each conditional block as a separate path. This is important when trying to convert the program into SSA form. Figure 3 showcases this where the variable sent through the second *sendSms* could either be *variable2* or *variable3* depending on whether the if-statements is executed through the true branch or the false branch.

This is a challenge when using SSA form to deal with flow sensitivity for such cases. The problem is exemplified when dealing with conditional statements, such as if-statements and switch statements. In our thesis we're going to focus on if-statements. These present branching in the program's execution path. A precise analysis takes these branches in consideration and provides all possible cases where a leakage can occur while avoiding false positives.

In its simplest form, this problem can be represented with a single if statement. This will give us two branches; a branch when true and branch when false. To achieve that



```
def test () {
    def variable1 = 1
    variable2 = 2
    if (x) {
        variable3 = 3
        sendSms(variable3)
    }
    sendSms(variable)
}
```

FIGURE 3. Example of potential paths from an if-statement.

and generate these paths from an inputted program, we wrote two TXL transformations. One that removes the if statement, while preserving the new lines. And another that extracts the statements from its body.

At this point, we have the base case solved, but a program could be more complex; it can contain an if-else, multiple if statements, nested if statements or a combination of these. We want to generate all the possible branches from these. To solve this problem, we can recursively generate the true and false paths from the program, removing the if-statements in the process, until we reach the base case.

To do this, we wrote a python script. The script would be responsible for recursively calling the TXL transformation for generating the true and false paths for one if-statement at a time, taking the output and rerunning the transformations, until no if-statement is left. This way we can generate all possible paths from any program with multiple or nested if-statements. After this stage, we would have gotten all the possible paths. Figure 4 shows an example of how the program would run on a program with two if statements, generated all the possible four paths.

For if-else statements, for the initial implementation we were converting else statement into if-statements and generating the paths from there using the same method. This approach would end up generating four paths for an if-else statement, with two impossible paths, the path of executing both the if and else clauses and the path where neither is executed. In our later implementation, we changed the way we deal with them so that if we consider the path where the *if* clause is true, the *else* clause is always skipped, and if we consider the path where the *if* clause is false, the *else* clause is always executed. This accurately produces two execution paths for the if-else statements.

As for the final output, each program would give us multiple paths. Each path would be represented by a combination of *T* and *F* symbols. *T* for true path and *F* for false. And the python script would connect these outputs with the SSA transformation and Taint-Things to preform the analysis. To give an example, if we take listing 17 as an input, the output produced by our path sensitivity framework is batch report for the two possible paths as showed in listing 18. We can see both possible paths generated and that one path has a tainted flow reported while the other is benign.

The full implementation can be described as follows. The goal of this module is generating the possible execution paths

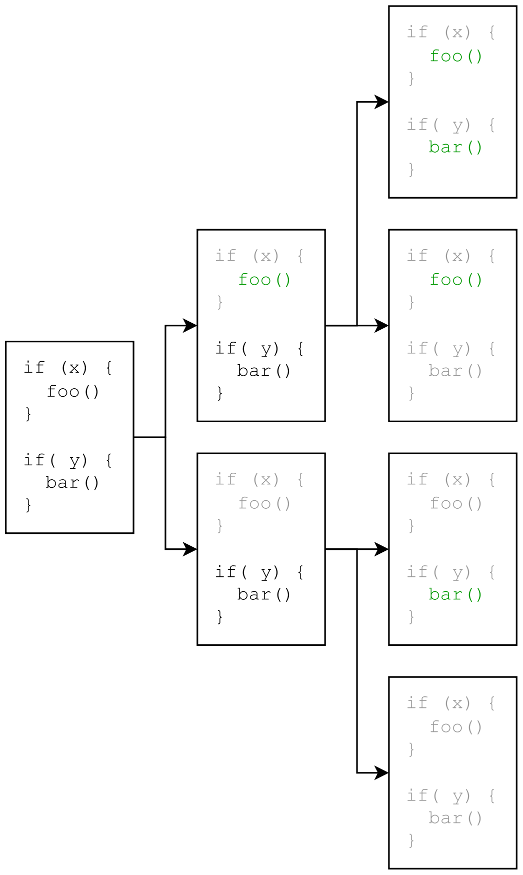


FIGURE 4. Example of path generation from two if-statements. Grey lines are deleted while green lines are kept. Top branching represent a True path and bottom branching represent the False path.

```
//sensitiveData defined
def test () {
  if (x) {
     message = sensitiveData
  } else {
     message = "benign"
  }
  sendSms (message)}
```

Listing. 17. Path sensitivity example.

in the program by generating the true and false branches from if-statements. This is done by running the analyzer to mark statements that are within potentially tainted flows. To check if an if-statement exists that contain a tainted statement, the module parses the inputted program statements and matches if-statements. It then deconstructs them to see if they contain a marked statement. If such if-statement exists, a TXL program that generates the true path and a TXL program that generates the false path are run. A false path is generated by parsing the program statements, matching if-statements, replacing its content with newlines and deconstructing else-statements and replacing it with the contained statements. A true path is generated by deconstructing the if statements and replacing it with the contained statements and replacing the else-statement with new lines. The program then recursively repeats the process from the output of the path generation until no if-statements are left. Each output is

```
Running Test.groovy
  Path: T

      def test () {
         < 8 source > message1 = sensitiveData <
              / >
         < sink > sendSms (message1) < / >
      }

      4 8

  Path: F

      def test () {
         < 8 > message2 = " benign " < / >
         < sink > sendSms (message1) < / >
      }
```

Listing. 18. Path sensitivity output.

then cleaned from statements markings and is passed to be analyzed. This is illustrated in Figure 5.

Algorithm 2 Path Generation

Input: A program's source code
Output: Multiple programs representing the possible execution paths

```
1 Function handleIf(X):
2     if X contains an if-statement then
3         truePath ← generateTruePath(X)
4         falsePath ← generateFalsePath(X)
5         handleIf(truePath)
6         handleIf(falsePath)
7     end
8 end
```

The first thing to note is that this process can be costly in terms of performance. As we have seen, with each if-statement, the program would have double the branching. This grows exponentially with each if-statement added and eventually means that the analysis will be done 2^n the times of how many if-statements there are in the program. To illustrate the frequency of if-statements in each of the dataset's apps, in each app we counted how many if-statements exist using a TXL program, then organized the results in a histogram showing how many apps have a frequency of if-statements. Figure 6 shows the histogram frequency of if-statements in the dataset's apps. We noted 15 apps having over 50 if-statements and can be considered outliers. We excluded these 15 apps in figure 7 to show a better detail of the frequency distribution. In our tests we've seen that a program with more than 12 if-statements becomes too large to add path sensitivity in a practical way. And since the growth is exponential, with each if-statement making at least two possible branches, we want to optimize the process and try to deal with as little if-statments as possible.

One way of optimizing this is doing the transformations on all the methods at once. So, a program with two methods, each with one if-statement, would only produce two paths, instead of four. So the amount of paths generated would be

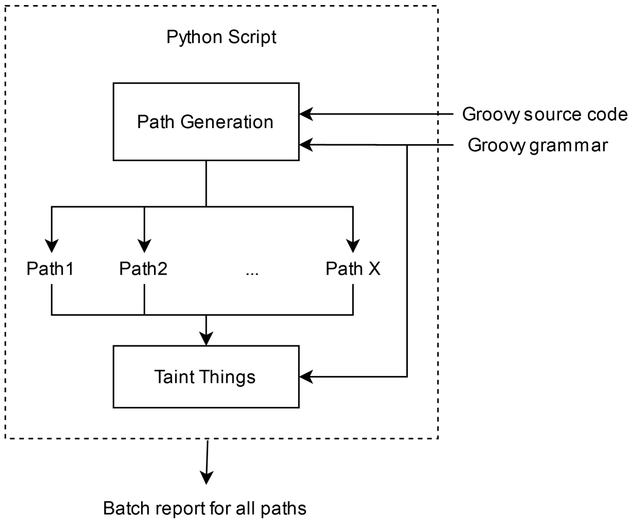


FIGURE 5. Taint-Things with path sensitivity.

the exponential of the maximum number of if-statements in one method rather than the total of if-statements in the source code. Figure 8 shows a histogram of the frequency of the maximum if-statement in one method in each app in the dataset. We can see this effectively lowering the amount of possible branches, with less apps having over 12 if-statements that we have to deal with, but this can come on the expense of context-sensitivity and sometimes ignores certain possible paths that affects a flow through multiple methods; this can happen if two methods, dependant on each other, containing if-statements that affect the same data flow.

Another thing to note is that not all paths generated necessarily affect the dataflow, so some programs will have multiple paths with the same dataflows reported on different paths. One way to overcome this and provide better optimization is to only consider if-statements that affect the flow. To do that, we can perform a path-insensitive analyses to mark the sinks and backtrace, marking statements in the data flow, then performing the path sensitivity analyses where we generate paths only from if-statements containing the marked statements, and finally performing the analyses again on the paths to generate results. Adding two analysis steps is considerably cheaper than the exponential cost of performing the analyses on all if-statements that don't effect the flow. Figure 9 shows the frequency of if-statements that actually affect the flow in the apps dataset's app. It shows that it can significantly reduce the required amount of if-statements to process in most apps.

C. ADDING CONTEXT-SENSITIVE ANALYSIS

The previous two subsections, we looked into how the flow can affected through variable reassignment and how conditionals can affect that. But sometimes variable reassignment is done through a method calls. Handling method calls and their return values with precision can also pose a challenge. In this section we look into this case where the flow goes

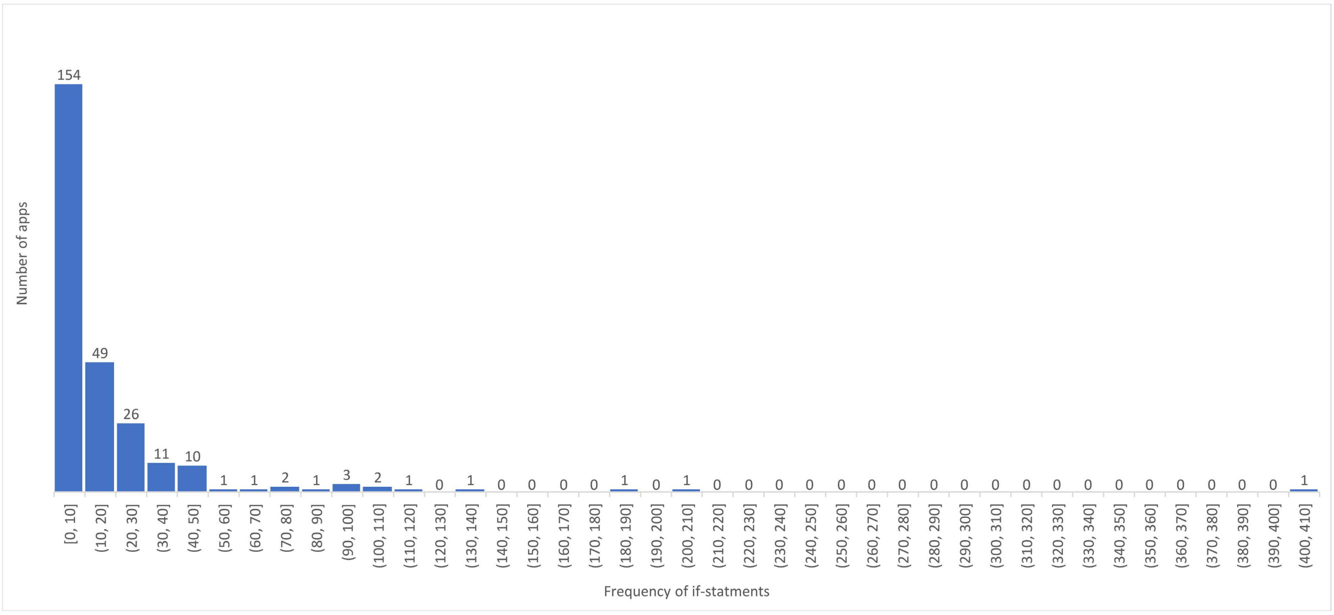


FIGURE 6. If-statement frequency in the dataset apps.

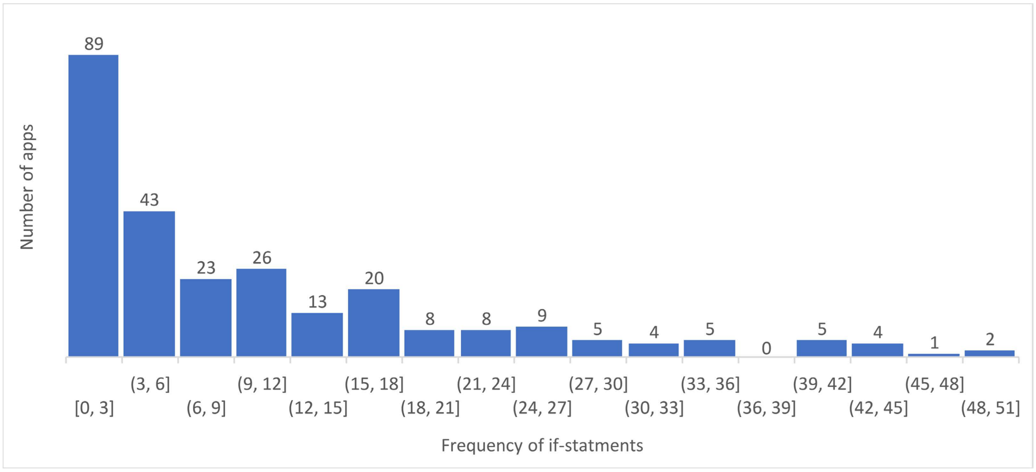


FIGURE 7. If-statement frequency without outlier values.

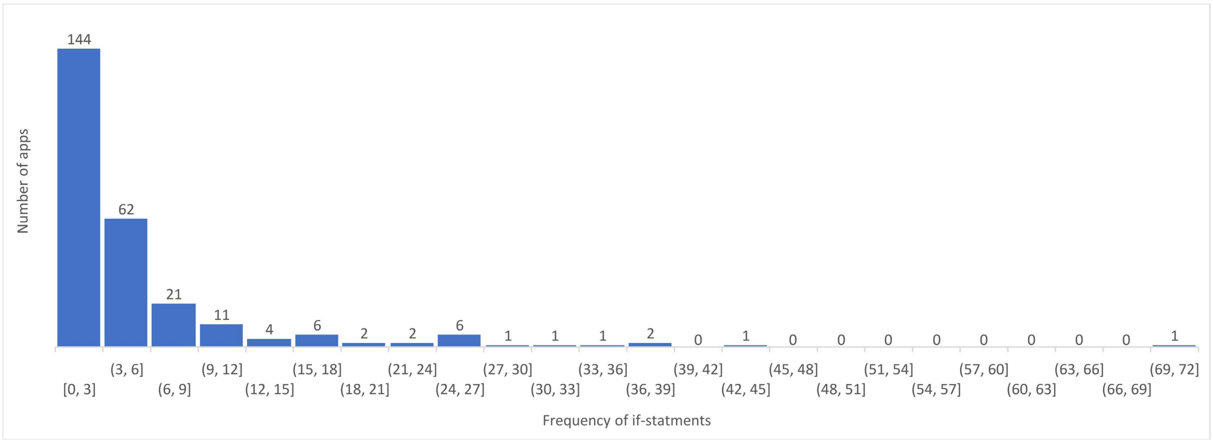


FIGURE 8. If-statement frequency with method-centered optimization.

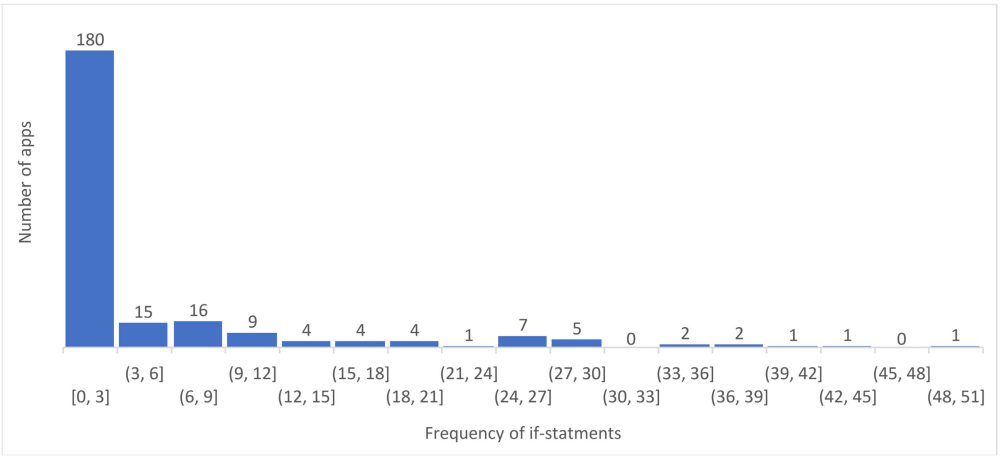


FIGURE 9. If-statement frequency with method-centered optimization.

```
//$sensitiveData defined
def takeAction() {
   def message = "This contains $sensitiveData";
   def firstCall = returnMessage(message);
   def secondCall = returnMessage("no sensitive data");
   sendSms(secondCall);
}
private returnMessage(message) {
  return message;
}
```

Listing. 19. Context sensitivity example.

through method calls and how to handle the context of each call.

A context sensitive analysis takes the context of each method call in consideration while a context insensitive approach might confuse them and mark every call to a method as tainted if at one point there were one tainted call. If we take listing 5 where a method gets called twice, once with a tainted flow passing through it and another with a benign flow. In a context insensitive analysis, both calls on lines 4 and 5 might be conflated; the method is marked tainted after *firstCall*, so *secondCall* will also be considered tainted and the flow tainted. A context sensitive approach on the other hand, doesn't confuse method calls and distinguishes each call site, so it won't mark the flow to *sendSms* tainted.

One way to add the sensitivity is through method cloning. Where, similar to the SSA approach, we assign each method call a unique call name and clone the function with it. This assures that each call sight is treated uniquely and no conflation happens. This can be costly if there is a lot of nesting in the methods and won't work properly in the case of recursive functions. Another disadvantage of this approach is that it affects the line numbering of the data flow; depending on where the cloned functions are placed in the source code, the data flow will go through them rather than the original methods. If we apply that to the previous listing, the result will be like listing 20.

We wrote a TXL program that can be used to do method cloning in the program, by identifying any method call, assigning a unique method name to it and cloning the method

Algorithm 3 Rename Method Calls in a Program and Make Clones of the Original Method With the New Name

Input: A program's source code
Output: Program's source code with function cloning

```
1  for each method M do
2      $C(M) \leftarrow 0$
3      get method name $MN$ from $M$
4  end
5  for each statement MC in program X do
6      if MC is a method call and MC equals MN then
7          $i \leftarrow C(M)$
8          $C(M) \leftarrow i + 1$
9          replace $MC$ with new $MCi$
10         copy $M$ body and replace $MN$ with $MCi$
11     end
12 end
```

```
//$sensitiveData defined
def takeAction() {
   def message = "This contains $sensitiveData";
   def firstCall = returnMessage1(message);
   def secondCall = returnMessage2("no sensitive data");
   sendSms(secondCall);
}

private returnMessage(message) {
   return message;
}
private returnMessage1(message) {
   return message;
}
private returnMessage2(message) {
   return message;
}
```

Listing. 20. Function cloning example.

definition using the new name. This is done by parsing the inputted program and iterating through defined methods. Extracting the method's name, parsing the program, and for any expression with a method call that matches the method name, constructing a new unique method name. It then constructs a clone of the defined method but assigns the unique method name to it, and replaces the call to the old method

TABLE 2. Comparative analysis of SainT and Taint-Things.

App Category	Taint-Things		SAINT	
	Benign	Leaking	Benign	Leaking
Community (Total: 117)	68	49	70	47
Forum (Total: 41)	16	25	17	24
Market (Total: 36)	11	25	13	23
Benchmark (Total: 19)	1	18	1	18

* A leaking app is an app with at least one tainted flow is detected

with a call to the new name. Finally, it appends the program with the newly constructed cloned method

V. EVALUATION

A. EVALUATING TAINT-THINGS

In this experiment we try to answer the following two research questions:

- RQ1: How does our static taint analysis approach compares with other available approaches in terms of accuracy and performance?
- RQ2: Whether our approach added modules for flow, path and context sensitivity minimize false positives?

To answer *RQ1*, and to measure Taint-Things's performance in terms of correctly detecting apps containing potential leakage and the speed of the process, we have conducted a comparative analysis experiment with SAINT tool [7]. We collected SAINT's analysis reports on our dataset and then manually compared their findings with ours. The dataset included 264 applications; 42 official SmartThings Marketplace apps, 144 official apps provided by the community, 59 third-party apps collected from the forums and 19 apps specifically developed by the SAINT team to include common vulnerabilities available online under IoTBench test suite [18].

A thing to note in our comparative study is that some applications would timeout on each of the tools, whether it's due big size or convoluted function calls. SAINT's web portal having around 42 cases while our tool had 18. For this part of the comparative analysis, we excluded all cases of timeouts.

Table 2 presents the summary of our comparative analysis with SAINT. The table presents the number of apps that were reported malicious, due to potential leakage of sensitive information in the form of at least one tainted data flow, vs. those that were identified benign. Our results matches SAINT's results in terms of finding which apps have malicious flows except in 5 apps where we report them containing potential tainted flows while SAINT reports them as benign and doesn't report any potential leakage. The criteria for this comparison was looking at what sinks SAINT was reporting and checking if it reported a potential flow being passed to those sinks.

In addition to comparing our results with SAINT, we verified that our findings were accurate by checking the source code and seeing that the reports matched it. For the cases with mismatched results, we found that the the reason for this is due to lack of field sensitivity in our tool and how we handle state variables. In our tool, we considered any state variable to be a potential source and so mark the flow

TABLE 3. Comparison of the benchmark set between Taint-Things and SAINT.

	Taint-Things	SAINT
call_by_reflection_1	Warning followed by 10 tainted flows to 4 sinks	5 flows, 4 sinks and 2 findings
call_by_reflection_2	2 tainted flows to 2 sinks	3 flows, 2 sinks and 2 findings
call_by_reflection_3	7 tainted flows to 3 sinks	7 flows, 2 sinks and 7 findings
explicit	6 tainted flows to 3 sinks	5 flows, 3 sinks and 3 findings
global_variable_1	7 tainted flows to 3 sinks	5 flows, 3 sinks and 4 findings
implicit_1	3 tainted flows to 2 sinks	5 flows, 4 sinks and 4 findings
implicit_2	4 tainted flows to 3 sinks	6 flows, 3 sinks and 4 findings
implicit_explicit	9 tainted flows to 3 sinks	6 flows, 3 sinks and 8 findings
leaking_via_closures	5 tainted flows to 3 sinks	5 flows, 3 sinks and 6 findings
multiple_devices_1	2 tainted flows to 2 sinks	7 flows, 3 sinks and 5 findings
multiple_devices_2	2 tainted flows to 1 sink	7 flows, 3 sinks and 6 findings
multiple_devices_3	2 tainted flows to 1 sink	5 flows, 3 sinks and 5 findings
multiple_entrypoint_1	11 tainted flows to 3 sinks	7 flows, 3 sinks and 9 findings
multiple_entrypoint_2	4 tainted flows to 3 sinks	5 flows, 3 sinks and 7 findings
multiple_leakage_1	10 tainted flows to 4 sinks	7 flows, 5 sinks and 5 findings
multiple_leakage_2	6 tainted flows to 5 sinks	11 flows, 5 sinks and 9 findings
multiple_leakage_3	5 tainted flows to 5 sinks	37 flows, 7 sinks and 16 finding
side_channel_1	No leakage	No leakage
side_channel_2	3 tainted flows to 3 sinks	6 flows, 3 sinks and 2 findings

from it as tainted. This can be addressed on the grammar level and how the program parses state atomic variables. For this case our tool has a more generalized approach when detecting flows from state variables, resulting in the apps that our tool marked and SAINT did not. When looking in the detailed results and comparing the flows reported in each app we've also found that SAINT can report some hard-coded strings as potential leakage sources if they resemble a number, where our app considers hard-coded strings as benign. This can be added by using regex to detect strings that contain phone number patterns or utilize natural language processing to detect strings that can act as sensitive data sources, but at this point, this is beyond the scope of our research.

Table 3 shows the reports provided by the two tools, Taint-Things and SaINT when running the benchmark set. Here, we provided the reports as is, but it should be noted that the way the two tools report their results are different, hence the need for manually checking each case. For example, SAINT's report starts with all the detected sinks and all flows, regardless if they were tainted or not, and reports the findings based on the source variables, while Taint-Things only reports tainted flows in the summary and bases the report on the line numbers. When manually checked, we found that the actual tainted flows detected generally match between the two tools with the exceptions mentioned previously, which are state variables and hard coded strings.

The answer to our *RQ1* is: The core module of Taint-Things accurately detects the same tainted flows that SAINT detected while showing significant improvement in speed. Celik *et al.* [6] reported on the SAINT's results on a 230 dataset, using a 2.6GHz 2-core Intel i5 processor and 8GB RAM took around 16 minutes to evaluate the batch, while an individual app took 23 ± 5 seconds on average. On the other hand, our tool achieved significant improvement in performance with at least 4 folds. In addition, our tool was able to analyse apps that SAINT times out or fails on startup. detailed results of performance analysis of our tool is presented in Table 4. This improvement is mainly because our approach computes dependency chains directly form the code, using the inductive transformation paradigm, while

TABLE 4. Performance analysis.

App Category	Including Warning-producing Apps	Excluding Warning-producing Apps	Timeout\Warning Taint-Things	Timeout\Failed SAINT
Community	1m25s (Total apps: 143)	0m54s (Total apps: 138)	6	24
Forum	2m57s (Total apps: 56)	0m25s (Total apps: 48)	11	12
Market	0m27s (Total apps: 42)	0m14s (Total apps: 41)	1	6
Benchmarks	0m9s (Total apps: 19)	0m9s (Total apps: 19)	0	0
Total	**4m4s (Total apps: 260)**	**1m26s (Total apps: 246)**	**18**	**42**

TABLE 5. SSA test results.

	Taint-Things		SSA		SAINT	
	Benign	Leaking	Benign	Leaking	Benign	Leaking
Community	67	55	67	55	69	53
Forum	16	28	16	28	17	27
Marketplace	11	24	11	24	13	22
Benchmark	01	18	01	18	01	18

TABLE 6. Detailed results when running Taint-Things with and without SSA form on the benchmarks set.

	Taint-Things	Adding SSA
call_by_reflection_1	Warning followed by 10 flows to 4 sinks	Warning followed by 4 flows to 4 sinks
call_by_reflection_2	2 flows to 2 sinks	2 flows to 2 sinks
call_by_reflection_3	7 flows to 3 sinks	7 flows to 3 sinks
explicit	6 flows to 3 sinks	6 flows to 3 sinks
global_variable_1	7 flows to 3 sinks	7 flows to 3 sinks
implicit_1	3 flows to 2 sinks	3 flows to 2 sinks
implicit_2	4 flows to 3 sinks	4 flows to 3 sinks
implicit_explicit	9 flows to 3 sinks	9 flows to 3 sinks
leaking_via_closures	5 flows to 3 sinks	5 flows to 3 sinks
multiple_devices_1	2 flows to 2 sinks	2 flows to 2 sinks
multiple_devices_2	2 flows to 1 sink	2 flows to 1 sink
multiple_devices_3	2 flows to 1 sink	2 flows to 1 sink
multiple_entrypoint_1	11 flows to 3 sinks	11 flows to 3 sinks
multiple_entrypoint_2	4 flows to 3 sinks	4 flows to 3 sinks
multiple_leakage_1	10 flows to 4 sinks	10 flows to 4 sinks
multiple_leakage_2	6 flows to 5 sinks	6 flows to 5 sinks
multiple_leakage_3	5 flows to 5 sinks	5 flows to 5 sinks
side_channel_1	No leakage	No leakage
side_channel_2	3 flows to 3 sinks	3 flows to 3 sinks

TABLE 7. Runtime performance when using SSA.

	Without SSA	With SSA
Real time	6m3.505s	4m14.684s
User time	0m1.745s	0m5.365s
System time	0m6.717s	0m17.959s

TABLE 8. SSA test results.

	Taint-Things		SSA		SAINT	
	Benign	Leaking	Benign	Leaking	Benign	Leaking
Community	67	55	67	55	69	53
Forum	16	28	16	28	17	27
Marketplace	11	24	11	24	13	22
Benchmark	01	18	01	18	01	18

TABLE 9. Flow sensitivity mutation test.

	#Apps	Without SSA	With SSA
Leaking	263	263 Detected as Tainted (True positive)	263 Detected as Tainted (True positive)
Benign	263	263 Detected as Tainted (False positive)	263 Accurately detected as benign (True negative)

SAINT builds a dependency graph for the program and then use graph algorithms to reduce it to the tainted flow, which is mapped back to source statements afterward.

B. EVALUATING FLOW-SENSITIVE ANALYSIS

To answer (RQ2), evaluate the results of our implementation with flow-sensitive analysis, we compared the results of our tool with and without the application of SSA. We used the previous dataset, which includes 260 IoT apps gathered from different sources. For the sake of consistency we have excluded programs that gave errors in the final results; this included 4 programs without SSA, and another 4 when adding due to the SSA TXL transformation failing on some apps. As well as 35 apps using SAINT web app at the time of the dataset collection. Table 5 shows the results of this test. The criteria is based on the previous comparison done in the previous section. We see that Adding SSA does not affect the final result which shows that no false positives were caused from the lack of flow sensitivity.

When looking in detail at the results of our tool when adding SSA form on this dataset and comparing them to the original results, we saw that the results are identical with few exceptions. This shows us that on a real dataset, most programs do not get affected by adding flow sensitive analyses and there aren't many cases of variables reassignment in a way that changes the dataflow. We have included a detailed look at the results on the benchmark apps from the dataset in Table 6. We can see that the results are identical with the exception of *call_by_reflection_1* where the SSA form actually helps in reducing the reported flows; in this case, without SSA a variable was being conflated with a possible source and thus flows from it were reported, while the SSA assigns the variable a unique name avoiding this mixup.

Performance wise, this functionality creates relatively little overhead. To measure that, we compare the time Taint-Things takes to run on the dataset, once with the SSA transformation chained to it and another without it. Table 7 shows that the SSA transformation had an increase in CPU time but this translated to having no significant effect on the real time.

Since we want to provide a definitive answer to *RQ2*, we want to accurately measure precision and recall. To achieve that, we used an independently generated dataset through a mutation framework that was developed by Alalfi *et al.* [1], [25]. The framework generates mutants targeting the evaluation of flow analysis by altering the statements' order in benign flows to generate leaking ones. The mutants covered multiple patterns where variable reassignment happen to make the flow tainted or benign and to be leaked in multiple sink types such as leakage through the internet, notifications and messages. To compute precision

TABLE 10. **Optimized path sensitivity results.**

	Taint-Things		SSA		Path Sensitivity		SAINT	
	Benign	Leaking	Benign	Leaking	Benign	Leaking	Benign	Leaking
Community	63	47	63	47	63	47	65	45
Forum	15	21	15	21	15	21	13	23
Marketplace	11	21	11	21	11	21	11	21
Benchmark	01	16	01	16	01	16	01	16

TABLE 11. **Detailed results of optimized path generation and analyses for the benchmarks Set.**

	#Paths	Benign Paths	Leaky Paths	#Flows
call_by_reflection_1	Error	Error	Error	Error
call_by_reflection_2	12	10	2 (2 unique)	2
call_by_reflection_3	3	0	3 (2 unique)	7
explicit	9	2	7 (5 unique)	5
global_variable_1	2	0	2 (2 unique)	4
implicit_1	2	0	2 (2 unique)	3
implicit_2	3	0	3 (2 unique)	4
implicit_explicit	3	0	3 (2 unique)	7
leaking_via_closures	2	0	2 (2 unique)	4
multiple_devices_1	2	1	1 (1 unique)	1
multiple_devices_2	2	1	1 (1 unique)	2
multiple_devices_3	3	0	3 (2 unique)	2
multiple_entrypoint_1	3	0	3 (2 unique)	9
multiple_entrypoint_2	3	0	3 (2 unique)	4
multiple_leakage_1	3	0	3 (3 unique)	6
multiple_leakage_2	3	0	3 (3 unique)	5
multiple_leakage_3	3	0	3 (3 unique)	4
side_channel_1	1	1	0	0
side_channel_2	Error	Error	Error	Error

and recall we used the following two equations:

$$precision = \frac{TruePositives}{TruePositives + FalsePositives}$$
$$recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

This dataset included 526 mutants, 263 benign and 263 leaking, adding SSA makes it possible to accurately report benign apps, while without it, it will be falsely reported as leaky. On the full dataset, Taint-Things achieves 100% recall with all the 263 leaking apps detected and 50% precision with 263 true positives out of the 526 reported leaking apps, but when adding SSA it achieves 100% recall and 100% precision with 263 true positives out of the 263 apps.

Overall we can see that adding flow sensitivity though SSA is inexpensive and doesn't create an overhead. While the real dataset didn't have cases where adding flow sensitivity makes a significant effect, this functionality can still help in avoiding the false positive case where a variable can be reassigned a a benign value and cases where variables can be conflated. This can be very useful if we apply mitigation to the dataflow, where we want it to be correctly marked as benign when applied.

One thing to note when transforming code to SSA form is the challenge of dealing with the scope. In groovy, variables are declared global by default. To make a local variable, it uses the *def* keyword and then any reference to it would be local. Also variables declared locally in a method will still be accessed by block statements in that method. This makes it hard to generalize a static-analysis approach for dealing with scopes. In our SSA TXL transformation, we put precedence

```
def initialize () {
  message1 = returnMethod ("benign")
  message2 = returnMethod ("$sensitiveData")
  sendSms (message1)
}

def returnMethod (x) {
  return x
}
```

Listing. 21. **Context sensitivity example.**

on the local scope by taking the structure of the inputted program into consideration. One problematic aspect of this approach is that some block statements, such as control flow statements can modify variables outside their scope. Flow sensitivity by itself ignores that, which can result into false negatives, as it either ignores variable modification inside these blocks, or in the case of a conditional statement, it'll only deal with one branch of execution. To deal with this problem path-sensitivity can be used.

C. EVALUATING PATH-SENSITIVE ANALYSIS

To answer *RQ2* and evaluate our approach for path sensitivity, we have run the program on the original dataset. We have gathered the results and compared them to the previous findings of our tool and SAINT's.

Table 10 shows a comparison between the results generated by Taint-Things with and without path sensitivity and SAINT. We used the optimization method described earlier, but it should be noted that there were still few apps with more than 12 if-statements that we had to skip, since they end with a large number of paths that would exceed the ability of our testing device to analyze in a practical manner. And for the sake of consistency we also skipped the apps the introduced errors just like previous comparisons. Since the path tool actually generates multiple cases for each program, the criteria we followed for the classification was if at least one leaking path was detected, the program is flagged leaking as such. Table 10 shows our findings. We saw that on a real dataset, there wasn't any miss-match related to path sensitivity when it comes to determining whether a program is benign or leaking. This shows that there were no false positives related to path sensitivity on the original test. For a more detailed look in the results, we've added the detailed results on the benchmark part of the set in Table 11 where we examine how many paths were generated and how many of them are benign, denoting a possibility of running the program in a way that doesn't leak data. We also looked how many of the leaky paths contained unique new flows, as apposed to having the same flow reporting in each of them

TABLE 12. Detailed results when running Taint-Things with and without method cloning form on the benchmarks set.

	Taint-Things	Adding Cloning
call_by_reflection_1	Warning followed by 10 flows to 4 sinks	Warning followed by 23 flows to 10 sinks
call_by_reflection_2	2 flows to 2 sinks	4 flows to 2 sinks
call_by_reflection_3	7 flows to 3 sinks	4 flows to 3 sinks
explicit	6 flows to 3 sinks	8 flows to 6 sinks
global_variable_1	7 flows to 3 sinks	17 flows to 7 sinks
implicit_1	3 flows to 2 sinks	3 flows to 2 sinks
implicit_2	4 flows to 3 sinks	6 flows to 6 sinks
implicit_explicit	9 flows to 3 sinks	12 flows to 6 sinks
leaking_via_closures	5 flows to 3 sinks	12 flows to 6 sinks
multiple_devices_1	2 flows to 2 sinks	6 flows to 6 sinks
multiple_devices_2	2 flows to 1 sink	6 flows to 3 sink
multiple_devices_3	2 flows to 1 sink	2 flows to 2 sink
multiple_entrypoint_1	11 flows to 3 sinks	16 flows to 6 sinks
multiple_entrypoint_2	4 flows to 3 sinks	6 flows to 6 sinks
multiple_leakage_1	10 flows to 4 sinks	24 flows to 8 sinks
multiple_leakage_2	6 flows to 5 sinks	14 flows to 10 sinks
multiple_leakage_3	5 flows to 5 sinks	10 flows to 10 sinks
side_channel_1	No leakage	No leakage
side_channel_2	3 flows to 3 sinks	Warning followed by 7 flows to 7 sinks

and finally the total number of flows reported in all paths. Generally the flows reported matches the ones in the SSA form previously, but the path sensitivity gives more detail to the cases where each can happen.

In the original version of the path-sensitivity approach was also independently tested on a set of mutations that were based on the original dataset to include path-sensitivity related attacks. The set included 440 mutations with different leakages of different types; through messages, posting on the internet and notifications. When tested, it was found that our tool managed to cover all the possible flows for all the mutations. From this set, our tool managed to catch all the tainted flows while avoiding false positives. The results on this set showed a high level of precision and recall; the precision is 100% since there are no false positives reported in that set, and recall is calculated as 100%.

1) EVALUATING CONTEXT-SENSITIVE ANALYSIS

To answer *RQ2* and evaluate our approach for context sensitivity, we first wanted to make test case exemplifying the case and how it is handled by Taint-Things. Listing 21 shows an example where *message1* calls *returnMehtod* with a benign string and then gets sent through a sink. The expected result is that no tainted flow is to be reported. On SAINT, it correctly reports that there is no potential leakgage. But in Taint-Things we get a flow from the lines: 3 7 8 2 10. This shows a context insensitive behavior, where *returnMehtod* got tagged as tainted because of *message2*. And even though the context of the method call in *message1* is different, it still gets conflated and the flow is considered tainted because of that. But after we run the function cloning transformation on this example, Taint-Things doesn't report a tainted flow, which is the correct context-sensitive behaviour.

We've also ran a test on the dataset to see how it affects the results. Like the previous comparisons, it didn't have a big difference on the dataset in terms of labeling apps as tainted or benign, thus showing that there were no false positives done due to context sensitivity in the dataset. We found that there were changes done on how many unique flows and sinks were detected as well as changes on the line numbers reported from the original source code. This is due to the flow running through the multiple newly made clone methods. Performance wise, the process of function cloning isn't heavy in itself, but as the program gets bigger and with more nested function calls, analyzing the results can be more costly.

Additionally, we've tested the tool on a dataset of mutations that adds patterns requiring context sensitive analysis. With the exception of one mutation operator, our tool was able to detect all the tainted flows while avoiding false negatives. For that operator, due to its complexity, the tool had a problem parsing method calls and ended up conflating them. Overall it achieved 100% precision and 96.8% recall.

In the mutation testing, if we give equal weight to each of the three categories of mutators that address the three levels of sensitivity analysis, we can average their results to get an estimated overall precision and recall for each of the tools. When one file is counted as one mutant and the correctness of the path sensitivity results are considered SaINT has 100% recall and 56.8% precision, Taint-Things has 99% recall and 100% precision.

For Taint-Things, it can distinguish the change from the created base file to the generated mutant. But it failed to identify the mutants generated from one app that contained an extensive usage of state variable which marked it aggressively as a potential source. It failed for all the benign equivalent mutants generated from one source app when we only had sixteen source apps.

VI. RELATED WORK

IoT is still new technology, yet diverse, and it poses many security challenges. To get an overview of the field, we look into the previous research done on IoT security. Furthermore, to get a better understanding of the techniques and approaches, we have to look outside of IoT research and into different fields such as android apps security, which shares some similar features, but had more time to mature.

A. IoT SECURITY

Since the field of program analysis for the IoT is still in its infancy, there is only few related work on this area. Fernandes *et al.* [13] presents an approach for exposing vulnerabilities in SmartTings IoT apps. They concluded that many of the existing applications have vulnerabilities, mainly in the form of over-privilege. This study opened the field for later research to investigate the security aspects from a program analysis point of view, trying to provide potential solutions or ways to detect these problems.

Tian *et al.* [31] proposed a semantic based approach, with the objective of better representing applications' functionality and privileges to users. While Wang *et al.* [34] dealt with the logging problem and interconnectivity. With attention on the privacy aspect of IoT security, Celik *et al.* [6] tried to programmatically detect sensitive information used in apps where breaches might happen. Their research introduced SAINT, which is a static analysis tool that tries to detect tainted flow through IoT apps' code, which could lead to sensitive data leakage. SAINT uses Groovy AST API to help recover an intermediate representation (IR) where taint sinks and sources are identified. They proposed using IR as means of abstracting the code, focusing on the important parts which might make the analysis easier. Sensitive data flow is then detected and reported if it is a feasible flow; meaning, the code can execute the sink function and leak data through it.

Celik *et al.* [9] also used this approach and applied it on abuse prevention, the safety and security aspects of IoT apps. They introduced SOTERIA which performs static analysis check to find potential vulnerabilities in apps where it is tested against safety, security and functionality properties. This could be used both on single apps or in multi-app environments.

We focus in our research on the issue of privacy leaks using static analyses, so we looked more into the literature dealing with IoT app analysis. A literature review by Celik *et al.* [8] in 2018, surveyed six available tools that does privacy and security analyses, one of them, Saint, does static analysis for data leaks detection.

This comparison looked at multiple features concerning IoT specific issues, handling of app idiosyncrasy and analysis sensitivity. Specific issues include: Multi-app analysis, trigger-action platform support, proactive defense, lack of runtime prompts. Idiosyncrasies include: RESTful APIs, Closures and calls by reflection. Analysis sensitivities includes flow, context, field, path, and provenance tracking.

According to the report Saint provides analysis sensitivity for all types and handles the mentioned idiosyncrasies. As well as having no runtime prompts and providing proactive defense. The criteria for path and context sensitivity that was considered is that the tool does not run infeasible paths. Saint achieves that by pruning these paths in the IR using a work list approach. A more detailed criteria would be considered.

Similar to SAINT, we offer another take on the problem of detecting potential privacy leaks in the source code, with the goal of adding more efficiency and exploring ways to improve the precision. SAINT and SOTERIA implement their algorithms on the AST of a SmartThings app because of the constraints on Groovy language and proprietary back-end libraries. However, in our approach we compute dependency chains directly from the SmartThings App's code, using an inductive transformation paradigm. Our experiments shows that our approach produces equivalent results with significant improvement in performance, in speed and memory usage.

B. STATIC ANALYSIS IN ANDROID APPS

We wanted to look more in depth in applications of static analysis and how sensitivity can be applied to it, so we looked into tools that run on Android apps. One thing to note in Android apps is that they have a well-defined IR. So, analyses can be done directly on that. Soot [30] and WALA [33] are tools commonly used to convert the source code to its IR.

This approach is what an IFDS framework [26] does; the code is considered as an inter-procedural, finite, distributive subset, where it can be represented as a collection of flow graphs, with the statements and procedures as nodes. The analyses, then, is treated a problem of graph reachability. So in this case, an IR can be used to construct call graphs and used for the analyses.

We started by looking into FlowDroid [4] which is the first static analysis tool which provides full context, field, object, and flow sensitivity. It tries to solve problems that were not handled in previous tools. They are coarse grained and sometimes over or under approximate. Such problems happen when the life-cycle is not faithfully modeled, so the tool misses flows.

It is uses the Soot framework for representing the Java code and IR. It works by analyzes the byte-code and configuration file, making a dummy main method and constructing the call graph to emulate the life-cycle. It then preforms the analysis on the call graph.

The analysis is on-demand, based on IFDS framework, which adds context sensitivity, and is inspired by another tool, Andromeda, but adds more precision. For example, Andromeda can sometimes lack flow sensitivity, whereas FlowDroid adds that by using activations statements. One thing to note is that the IFDS framework and FlowDroid in extension are not path sensitive. It instead joins analysis results immediately at any control-flow merge point. Adding path sensitivity considered expensive.

In literature review of static analysis tools for android apps [20], the authors looked in multiple tools and their precision. One of their findings is that path sensitivity was often overlooked, with only 5 out of 30 of the surveyed tools provided it: Woodpecker [16], Apparecium [32], Anadroid [21], THRESHER [5] and ContentScope [36].

Apparecium detects arbitrary data flows. Avoiding entry point analysis, it directly uses the sinks and sources. It uses textual representation, smali, for the code which is used to generate the class hierarchy. It then uses backward slicing for variables that can be assigned to sinks, followed by forward slicing for variables that can contain a source, and then combine these. It uses a data flow graph for its representation and can add paths to it.

Woodpecker detects capability leaks in apps, which are cases where an app gains permissions without requesting. Builds a control flow graph from the byte code. It adds refines it path sensitivity using symbolic path simulation. Anadroid detects malware in android apps. And uses higher-order pushdown analyses and entry point saturation. ContentScope specifically tries to detect two vulnerabilities, passive content leak and content pollution and uses the analyses to determine their prevalence in Android markets. For detection of passive content leaks, it generates call graphs and tests for reachability. Thresher deals with the tries to detect heap reachability using static analysis.

These studies show different cases where static analysis is used in phone apps, specifically in Android, to detect taint flows, capability leaks, specific vulnerabilities or arbitrary flows. While there is some similarities in the concepts used in Android apps and IoT apps, like both being event driven models, there are differences and challenges specific to IoT.One problem is that while Groovy gets compiled to Java byte code, using the Android approaches is challenging due to Groovy's dynamic nature which makes it difficult to perform binary analysis with tools such as Soot. Other differences include that Android apps can have a straightforward IR, where IoT platforms on the other hand use different programming languages, each with its own features and quirks that should be taken in consideration in the analyses. Naturally, this makes approaches based on tools specifically tailored for Android, such as Soot and WALA, not applicable to IoT. And while SAINT proposed introducing an IR for SmartThings IoT apps for the analysis, we propose doing the analysis directly on the source code. This is because SmartThings apps are generally less complicated, usually smaller in size, contained in one file and have a simple structure, due to the use of a sandbox environment, where the programming language features are limited; for example you cannot define your own classes when writing SmartThings apps.

However, all the surveyed taint flow analysis techniques for Android did not provide path sensitivity analysis as it is an expensive analysis. While we do provide a light weight path sensitivity analysis without compromising performance. Our analysis approach can be adapted to other platforms such as Android. Other members from the team already adapted the core taint analyser to analyse android apps [22].Their analysis provided a more accurate with improvement to performance when compared to FlowDroid [4]. However, that approach is not yet expanded to enable flow-, path- and context- sensitivity analysis.

VII. DISCUSSION

In this paper We present a tainted flow static analysis approach for the identification and reporting of information leakage in Smarthings IoT apps. For RQ1, we show that implementing the core analysis directly on the source code with less preprocessing can achieve 4 fold the speed while achieving the same results as the available tools. This is possible due to the simple and defined structure of SmartThings app.

Our approach automatically transform the source into SSA form to make the analysis flow sensitive. This approach avoids false positives that happen due to conflation of variables in the case of reassigning their values. We provide a framework for path generation to make the analysis path sensitive and for it to consider the branching in code execution and explored ways of optimizing the process to make it more applicable. We also explored function cloning as a way to make the analysis context sensitive and to avoid the conflation of method calls.

When compared to SaINT [7]which parses SmartThings apps, transforms them into an intermediate representation and then constructs CFG to reason about taint flow analysis. Our approach applies an inductive transformation paradigm on the abstract syntax tree (AST) of the original source code. The AST is produced as part of the source transformation stage and is very much influenced by the grammar definition and overrides. There is no four step process like Saint (Parse, transform to IR, build CFG, then perform taint flow analysis). Instead our taint analysis is applied on the original source codes' AST. This is true for the core analysis. Providing more sound analysis requires some pre-transformations, such as SSA form for flows-sensitivity analysis, methods cloning for context sensitivity analysis, and path exploration module for path sensitivity analysis.

Furthermore, for RQ2, we show that flow, path and context sensitivity can be added as extra modules before the core analysis and would improve the precision of the tool by avoiding false positives. We found that the real dataset didn't contain major cases that generated false positives due to the lack of sensitive analysis, but provide them as a way of increasing the precision and avoiding potential false positives especially in the cases where a mitigation is introduced.

To confirm our findings, we reference an extensive study conducted by Alalfi *et al.* [25] to evaluate taint analysis tools for IoT applications using a mutation-based framework. The analysis evaluated Taint-Things with another two tools, SaINT and FlowsMiner. This study provides a clearer assessment of the tools' accuracy. It also tests the consistency of the tool's results over a large number of test cases.

In their flow-sensitive mutation tests they found that SaINT and Taint-Things have a recall rate of 100% for this calculation, but SaINT's precision rate dropped to 50% where Taint-Things' precision remains 100%, indicating that SaINT was not able to avoid false positives that were due to low sensitivity.

For testing the impact path-sensitivity mutators, they checked the correctness of the tool's reports. SaINT was able to identify the tainted results, giving a recall of 100% but had its precision drop by ignoring the potential benign paths if another tainted path existed and reported a false positive in mutants that had two benign paths. Taint-Things on the other hand is able to give a detailed report of all the potential paths.

When it comes to context-sensitivity mutators, the results confirmed that the tools are using context-sensitive analysis up to a certain level. For SaINT, even after getting the context, it failed to differentiate benign from malicious when using a specific sink.

If we give equal weight to each of the three categories of mutators that address the three levels of sensitivity analysis, we can average their results to get an estimated overall precision and recall for each of the tools. When one file is counted as one mutant and the correctness of the path sensitivity results are considered SaINT has 100% recall and 56.8% precision, Taint-Things has 99% recall and 100% precision. For Taint-Things, it can distinguish the change from the created base file to the generated mutant. But it failed to identify the mutants generated from one app that contained an extensive usage of state variable which marked it aggressively as a potential source. It failed for all the benign equivalent mutants generated from one source app when we only had sixteen source apps.

In general, we see that SmartThings real cases are not complex. From our tests, we have seen that all the apps from our datasets were not affected by the inclusion of the sensitive analyses. That is because most of them are simple one page apps written in the SmartThings sandbox [28]. This also allows our approach to avoid dealing with class hierarchy beyond the syntax analyses done on the TXL grammar side, since so users can not define their own classes. Tainted flows exists in the events and functions and our analysis focuses on tracking them in the source code. Similarly we do not deal with pointer analysis. But it should be noted, while real cases do not exhibit complexity that gets affected by the sensitivities, their inclusion provides an important addition not only to minimize false positives but also to avoid false negatives, especially in the case of mitigating leaking apps.

VIII. CONCLUSION

In this paper we present a tainted flow static analysis approach for the identification and reporting of informationleakage in Smarthings IoT apps. Our approach provides a quicker core analyzer as well as more accurate analysis by adding flow- path- and context sensitivity analysis as added modules. When compared to existing tools addressing the same problem, our analysis provides more accurate results with a considerable higher performance gain. One aspect that may have improved the performance over other existing techniques is that the approach performs the analysis directly on the source. In addition, our approach is unique in providing a lightweight path sensitivity analysis, and innovative way to implement context sensitivity analysis using methods cloning and source transformation.

We have deployed a version of Taint-Things that accounts for flow sensitivity analysis online as well. Taint-Things is currently available, for testing, at: http://taint-things.scs.ryerson.ca/.

IX. FUTURE WORK

Future work includes extending our analysis approach to other Smart Home platforms, such as OpenHab. OpenHab uses Java, which already have a TXL grammar [12], but it also utilizes a domain specific language for handling its rules. The syntax for this is shared with Xtend. We can use its documentation [35] to build a TXL grammar to handle it. OpenHab also uses a different structure than SmartThings apps and will have different definitions for sources and sinks. A study of how the sources and sinks appear in it, as well as modification of Taint-Things' transformation rules will be required to accommodate that.

While our approach examines potential leakage in each app by itself, further study can be done on potential leaks happening through multiple apps when they communicate with each other. SmartThings apps are usually self contained in one file and TXL can handle that easily as an input, but extending the program to handle multiple files and analyze their inter connectivity can be challenging. Nonetheless we can explore similar methods to what were used in adding flow and path sensitivity analysis, where we either chain and redirect the analysis results and use a script to work on multiple inputs.

```
/**
 * Copyright 2015 SmartThings
 *
 * Licensed under the Apache License, Version 2.0 (the "
     License"); you may not use this file except
 * in compliance with the License. You may obtain a copy
     of the License at:
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in
     writing, software distributed under the License is
     distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS
     OF ANY KIND, either express or implied. See the
     License
 * for the specific language governing permissions and
     limitations under the License.
 *
 * Unlock It When I Arrive
 *
 * Author: SmartThings
 * Date: 2013-02-11
 */

definition(
   name: "Unlock It When I Arrive",
   namespace: "smartthings",
   author: "SmartThings",
   description: "Unlocks the door when you arrive at your
        location.",
   category: "Safety & Security",
   iconUrl: "https://s3.amazonaws.com/smartapp-icons/
       Convenience/Cat-Convenience.png",
   iconX2Url: "https://s3.amazonaws.com/smartapp-icons/
       Convenience/Cat-Convenience%402x.png",
   oauth: true
)

preferences {
   section("When I arrive..."){
      input "presence1", "capability.presenceSensor",
           title: "Who?", multiple: true
   }
   section("Unlock the lock..."){
      input "lock1", "capability.lock", multiple: true
   }
}

def installed()
{
   subscribe(presence1, "presence.present", presence)
}

def updated()
{
   unsubscribe()
   subscribe(presence1, "presence.present", presence)
}

def presence(evt)
{
   def anyLocked = lock1.count{it.currentLock == "
       unlocked"} != lock1.size()
   if (anyLocked) {
      sendPush "Unlocked door due to arrival of $evt.
           displayName"
      lock1.unlock()
   }
}
```

Listing. 22. Real app example.

```
preferences {
   section (" When I arrive...") {
      input " presence1 ", " capability.presenceSensor ",
            title : " Who ? ", multiple : true
   }
   section (" Unlock the lock...") {
      input " lock1 ", " capability.lock ", multiple :
           true
   }
}

def presence (evt)
{
   if (anyLocked) {
      < sink > sendPush " Unlocked door due to arrival of
            $evt.displayName " < / >
   }
}
```

Listing. 23. Real app example mark sinks step.

```
preferences {
   section (" When I arrive...") {
      input " presence1 ", " capability.presenceSensor ",
            title : " Who ? ", multiple : true
   }
   section (" Unlock the lock...") {
      input " lock1 ", " capability.lock ", multiple :
           true
   }
}

def installed ()
{
   < 50 source > subscribe (presence1, " presence.present
         ", presence) < / >
}

def updated ()
{
   < 50 source > subscribe (presence1, " presence.present
         ", presence) < / >
}

def presence (< > evt < / >)
{
   if (anyLocked) {
      < sink > sendPush " Unlocked door due to arrival of
            $evt.displayName " < / >
   }
}
```

Listing. 24. Real app example trace backward step.

More ways to optimize and utilize precision can be explored. When adding flow sensitivity analysis, different ways to determine the variable scopes and using them in making the SSA form. This can be done through TXL and would require a rewriting of the script we currently use by altering the priorities of how variables get parsed and renamed. Ways for making path-sensitivity analysis less intensive and for context-sensitivity analysis to be achieved while maintaining the structure and line numbers of the original source code should also be studied. A compromise approach can be used where instead of doing full path or context sensitivity, we can use a partial approach. This can be done similarly to the optimized path sensitivity, where we only provide the path and context sensitivity only for some method calls and if-statements.

The next step after detecting tainted flow would be offering ways of mitigation or suggestions to good coding patterns and practices. This would require studying and mapping mitigation methods to each type of leakage. We can replace non-secure patterns with secure ones using TXL. And finally,

```
41 50 54
47 50 54



preferences {
   section (" When I arrive...") {
      input " presence1 ", " capability.presenceSensor ",
            title : " Who ? ", multiple : true
   }
   section (" Unlock the lock...") {
      input " lock1 ", " capability.lock ", multiple :
           true
   }
}

def installed ()
{
   < 50 source > subscribe (presence1, " presence.present
         ", presence) < / >
}

def updated ()
{
   < 50 source > subscribe (presence1, " presence.present
         ", presence) < / >
}

def presence (< 54 > evt < / >)
{
   if (anyLocked) {
      < sink > sendPush " Unlocked door due to arrival of
            $evt.displayName " < / >
   }
}
```

Listing. 25. Real app example final result.

we can explore ways for deploying the app in a more user friendly manner such as linking it to an IDE.

APPENDIX

The following Listing 22 is a real example of a SmartThings app that examines a common tainted flow pattern through a sendPush sink. The sensitive information here is the location of the user and his presence. It should be noted that this tainted flow is not necessarily malicious, but it presents a potential pattern where sensitive information could leak, which requires more scrutiny in the review, in case the data is not sanitized or going through an insecure API.

The following listings show the output of the analysis on the provided app. Listing 23 shows the sink marking step. In this case the line containing the sendPush is marked as such. Listing 24 shows the backward tracing step, where the variable passed to the sink are traced through the program, line containing the variables are tagged with the line numbers they pass the variable to. Listing 25 shows the final results when the app is run through the analyzers. The line numbers where the flows exists are printing with the relevant lines are marked.

REFERENCES

[1] M. H. Alalfi, S. Parveen, and B. Nazzal, "A mutation framework for evaluating security analysis tools in IoT applications," *CoRR*, vol. abs/2110.05562, pp. 1–36, Oct. 2021.

[2] G. Antlr, "Groovy parser and lex files," Tech. Rep. Accessed: Jun. 26, 2020.

[3] *Apple Homekit*, Apple, Cupertino, CA, USA. Accessed: Feb. 1, 2019.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. 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.

[5] S. Blackshear, B.-Y.-E. Chang, and M. Sridharan, "Thresher: Precise refutations for heap reachability," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 275–286, Jun. 2013.
[6] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity IoT," in *Proc. 27th USENIX Conf. Secur. Symp. (SEC)*. Berkeley, CA, USA: USENIX Association, 2018, pp. 1687–1704.
[7] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Saint analysis tool for smartthing apps," Tech. Rep., Last accessed: Feb. 1, 2019.
[8] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. McDaniel, "Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1–30, Jul. 2020.
[9] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*. Berkeley, CA, USA: USENIX Association, 2018, pp. 147–158.
[10] *Vera Control'S Vera3*, Vera Control, Clifton, NJ, USA. Accessed: Feb. 1, 2019.
[11] J. R. Cordy, "The TXL programming language," Tech. Rep. Accessed: Feb. 1, 2019.
[12] J. R. Cordy, "Java TXL grammar," Tech. Rep. Accessed: Jun. 26, 2020.
[13] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*. San Jose, CA, USA: IEEE Computer Society, May 2016, pp. 636–654.
[14] *Open Connectivity Foundation's Alljoyn*, Open Connectivity Found., Beaverton, OR, USA. Accessed: Feb. 1, 2019.
[15] *Google's Weave/Brillo*, Google, Mountain View, CA, USA. Accessed: Feb. 1, 2019.
[16] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. San Diego, CA, USA: The Internet Society, Feb. 2012, pp. 1–15.
[17] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 229–243, Apr. 2004.
[18] *IoTBench-Test-Suite*, IoTBench, Bengaluru, India. Accessed: Feb. 1, 2019.
[19] J. R. Cordy, "Excerpts from the TXL cookbook," in *Proc. 3rd Int. Summer School Conf. Generative Transformational Techn. Softw. Eng. (GTTSE)*. Berlin, Germany: Springer-Verlag, 2009, pp. 27–91.
[20] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of Android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, Aug. 2017.
[21] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn, "Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation," in *Proc. 3rd ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2013, pp. 21–32.
[22] A. Moiz and M. H. Alalfi, "An approach for the identification of information leakage in automotive infotainment systems," in *Proc. IEEE 20th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2020, pp. 110–114.
[23] *OpenHAB*, OpenHAB. Accessed: Feb. 1, 2019.
[24] *OWASP Top 10 Internet of Things 2018*, OWASP, Wakefield, MA, USA. Accessed: Jun. 26, 2020.
[25] S. Parveen and M. H. Alalfi, "A mutation framework for evaluating security analysis tools in IoT applications," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M.-E. Fokaefs, and M. Zhou, Eds., London, ON, Canada, Feb. 2020, pp. 587–591.
[26] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. 22nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1995, pp. 49–61.
[27] *Samsung SmartThings*, Samsung, Suwon-si, South Korea. Accessed: Feb. 1, 2019.
[28] *SmartThings Documentation*, Samsung, Suwon-si, South Korea. Accessed: Oct. 12, 2022.
[29] F. Schmeidl, B. Nazzal, and M. H. Alalfi, "Security analysis for SmartThings IoT applications," in *Proc. IEEE/ACM 6th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*. Piscataway, NJ, USA: IEEE Press, May 2019, pp. 25–29.
[30] *Soot*, Soot. Accessed: Jun. 26, 2020.
[31] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "SmartAuth: User-centered authorization for the Internet of Things," in *Proc. USENIX Secur. Symp.*, 2017, pp. 361–378.
[32] D. Titze and J. Schutte, "Apparecium: Revealing data flows in Android applications," in *Proc. IEEE 29th Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, Mar. 2015, pp. 579–586.
[33] *Wala*, Wala. Accessed: Jun. 26, 2020.
[34] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the Internet of Things," in *Proc. 22nd Netw. Distrib. Secur. Symp. (NDSS)*, 2018, pp. 1–16.
[35] *Xtend Documentation*, Eclipse Xtend. Accessed: Jun. 26, 2020.
[36] Z. Yajin and J. Xuxian, "Detecting passive content leaks and pollution in Android applications," *Proc. 20th Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2013, pp. 1–16.

BARA' NAZZAL (Member, IEEE) received the B.Sc. degree in computer science from the University of Jordan and the M.Sc. degree in computing from Toronto Metropolitan University (formerly, Ryerson University). He is currently pursuing the Ph.D. degree with the School of Computing, Queen's University. His research interests include cyber-security and software engineering with a focus on application analysis.

MANAR H. ALALFI (Member, IEEE) is currently an Associate Professor at the Computer Science Department, Toronto Metropolitan University (formerly known as Ryerson's University), and an Adjunct Assistant Professor at the Software Technology Laboratory, Queen's School of Computing, Canada. She is the Director of the Creative Research in Security and Software Engineering Technology (CRESSET) Laboratory. Her team conducts internationally recognized research in the area of software quality assurance, software security and vulnerability analysis, software analytics and big data, and model driven engineering.