State-Taint Analysis for Detecting Resource Bugs

Zhiwu XU[a], Cheng WEN[a], Shengchao QIN[b,a,*]

[a] *College of Computer Science and Software Engineering, Shenzhen University, China*
[b] *School of Computing, Teesside University, UK*

Abstract

To ensure that a program uses its resources in an appropriate manner is vital for program correctness. A number of solutions have been proposed to check that programs meet such a property on resource usage. But many of them are sophisticated to use for resource bug detection in practice and do not take into account the expectation that a resource should be used once it is opened or required. This *open-but-not-used* problem can cause resource starvation in some cases, for example, smartphones or other mobile devices where resources are not only scarce but also energy-hungry, hence inappropriate resource usage can not only cause the system to run out of resources but also lead to much shorter battery life between battery recharge. That is the so-call *energy leak* problem.

In this paper, we propose a static analysis called *state-taint analysis* to detect resource bugs. Taking the *open-but-not-used* problem into account, we specify the appropriate usage of resources in terms of resource protocols. We then propose a taint-like analysis which employs resource protocols to guide resource bug detection. As an extension and an application, we enrich the protocols with the inappropriate behaviours that may cause energy leaks, and use the refined protocols to guide the analysis for energy leak detection. We implement the analysis as a prototype tool called *statedroid*. Using this tool, we conduct experiments on several real Android applications and test datasets from Relda and GreenDroid. The experimental results show that our tool is precise, helpful and suitable in practice, and can detect more energy leak patterns.

Keywords: Resource Bug, Static Analysis, Energy Leak, Android

1. Introduction

Resource usage [1] is one of the most important characteristics of programs. To ensure that a program uses its resources in an appropriate manner is vital for program correctness. For example, a memory cell that has been allocated should be eventually deallocated (otherwise it may cause *resource leak* or *memory leak*), a file should be opened before reading or writing (otherwise it may cause program errors), and an opening camera (a popular resource for smartphones nowadays) should be closed eventually (or it will drain battery unnecessarily if it remains open after use).

A number of static analyses have been proposed to analyse the correct usages of computer resources [2, 3, 1, 4, 5, 6, 7]. Most of them adopt a type-based method to ensure a *resource-safe* property. However, although sound, they either require rather complex program annotations to guide the analysis or are rather sophisticated to use for resource bugs detection in practice, since one needs to enhance a type system with resource usage information, which may not be an easy task for users to follow. Taking the file resource as an example, a possible type annotation is $\mu\alpha.\ (0\&((l_{Read}\&l_{Write});\alpha)));l_{Close}$ [1], which involves several usage constructors. When files are accessed through the invocation of a function closure, the type annotation

*Corresponding author
Email addresses: xuzhiwu@szu.edu.cn (Zhiwu XU), 2150230509@szu.edu.cn (Cheng WEN), shengchao.qin@gmail.com (Shengchao QIN)

could be more complex, for example, $\mu\alpha.\ (0\&((l_{Read}\&l_{Write});\alpha)));l_{Close}\otimes U$, where U is the usages for the other resources. Moreover, few of them concern about that an opening (or required) resource should be *used* before it's closed (or released). While in some cases it may be just a minor problem or even cause no harm for an opened (obtained) resource to be left unused/unattended, in other cases this may lead to more severe problems. For instance, when some resource is very limited but is not released timely, it may lead to a major problem, causing resource starvation, severe system slowdown or instability. Specifically, for mobile devices, such as tablets and smartphones, which are ubiquitous and hugely popular nowadays, some resource can be rather limited, an example being their power energy (*i.e.*, the battery capacity). Most resources of mobile devices are not only scarce but also *energy-hungry*, such as GPS, WiFi, camera, and so on. Leaving these resources continuously open could be more expensive, as they would consume energy continuously leading to a shorter battery life (before a recharge). This is the so-call energy leak problem [8], that is, energy consumed that never influences the outputs of a computer system.

In this paper, we propose a static analysis called *state-taint analysis* to detect resource bugs, which is easy to use in practice and helps detect the *open-but-not-used* problem. Taking the *open-but-not-used* problem into account, we specify the appropriate usage of resources as resource protocols. A resource protocol describes how a resource should be used or which behaviour sequences (on resource usage) are appropriate. Resource protocols can be viewed as a kind of typestate properties [2], which can be represented as finite state automata. According to the API documentation of resources, different resources have different specific automata. A behaviour sequence that does not satisfy its corresponding protocol is considered as a resource usage bug.

Our proposed static analysis is a combination of typestate analysis [2] and taint analysis [9]. The analysis takes a control flow graph (CFG) of a program as input, and is guided by the resource protocols to track the resource behaviours among CFG. Following the idea of taint analysis, our analysis propagates the states of resources among CFG. During the propagation, our analysis will check whether the resource behaviours confirm to the corresponding resource protocols (*i.e.*, typestate checking). In detail, our analysis will verify that (i) whether all the resource behaviours obey the resource protocols before the exit of CFG, and (ii) whether the states of resources at the exit of CFG are all accepting ones of their corresponding protocols. Our analysis is flow-sensitive, so states for one resource from different paths may be different and are preserved.[1] Compared with the existing works [2, 3, 1, 4, 5, 6, 7], which adopt type-based approaches, our analysis (i) is easier to use (*i.e.*, without the need for complex type annotations), and (ii) helps to detect the *open-but-not-used* problem.

Furthermore, we enrich the resource protocols with extra information, namely, inappropriate behaviours that may cause a specific program bug, and we extend our *state-taint analysis* to detect these inappropriate behaviours. As an application, we use the extended behaviour analysis to detect energy leaks for smartphone applications, since the energy leak problem becomes a critical concern for smartphone applications. Usually, a resource bug can cause an energy leak, if the bug keeps the resource open unnecessarily. We distinguish the behaviour sequences that may cause energy leaks from the other inappropriate ones, and enrich the protocol with them. For example, to open an unneeded resource will cause an energy leak, while to use a closed resource will not. With the enriched protocols as a guide, we thus can use the extended behaviour analysis to detect energy leaks for smartphone applications. Moreover, sensitive data is essentially a resource. Our analysis can aslo be used to detect information leaks. In that case, our analysis degenerates into taint analysis. The extension of protocols with guards is also discussed.

We have also implemented the proposed analysis in a tool called *statedroid*, and conducted some experiments on many Android applications collected from F-Droid, and test datasets from Relda and GreenDroid. The experimental results show that our tool is precise and viable in practice. The results also demonstrate that, compared with Relda and GreenDroid, our tool can detect more energy leak patterns.

This paper is an extension of [10], and further contains the core language with resource usage, the proof of correctness, some possible extensions of the analysis, more experiments and several recent related work. Please note that, in this extended version, we generalize the analysis of energy leaks in [10] to be

[1] For simplicity, we do not consider the path conditions, so an infeasible path may lead to a false positive.

an extension of the proposed state-taint analysis (see Section 5), which we called as resource behaviour analysis, so as to analyze the behaviours that may lead to a special program bug such as energy leaks and information leaks. Therefore, instead of being a direct extension of the previous energy leak analysis in [10], our new energy leak analysis is an application of the extended state-taint analysis (i.e., resource behaviour analysis).

The rest of the paper is constructed as follows: Section 2 gives some backgrounds of taint analysis and typestate analysis and the notation list used in the paper. Section 3 illustrates some examples that have potential resource bugs or energy leaks. Section 4 presents the main algorithms of our analysis. Section 5 extends our analysis to detect some inappropriate behaviours and gives an application of using the extended analysis to detect energy leaks. Section 6 and Section 7 present the selected implementations and experiments, respectively. Section 8 reviews related work and Section 9 concludes the paper.

2. Background

Since our analysis is a combination of typestate analysis and taint analysis, we introduce them briefly in this section.

Taint analysis [9], one of the most well-known data-flow analyses, consists of tracking all the variables either which are predefined as taint sources such as the ones containing user supplied data (*i.e.*, a "taint") or whose computations depend on some taint sources (*i.e.*, a "taint" propagation) and preventing those variables from being used (*i.e.*, a "sink") until they have been sanitized. This technique is often applied on malware analysis, input filter generation, test case generation, vulnerability discovery, and so on.

Typestate analysis [2] is a form of program analysis, which is most commonly applied to object-oriented languages. Typestates define valid sequences of operations that can be performed upon an instance of a given type. Typestates associate state information with variables of that type, which is used to determine at compile-time which operations are valid to be invoked upon an instance of the type. Operations performed on an object that would usually only be executed at run-time are performed upon the type state information which is modified to be compatible with the new state of the object.

The notations used in the paper are listed in Table 1.

Table 1: Notation List

Notation	Descriptions	Notation	Descriptions
P	Programs	S	Statements
p, q	Variables	f, g	Filed names
r	Resources	rid	Resource instances
c	Classes	m	Methods
A	Actions in automata	$accepts$	Accepting states of automata
s	States in automata	v	Variable sets
d	Data facts	D	Data fact mapping
M	Memory environment, mapping variables to resource instances		
B	Behaviour environment, mapping resource instances to behaviour traces		
$L(rid)$	The set of appropriate behaviour sequences of rid		
MD	Method table, mapping method names to their definitions		
PD	Protocol table, mapping resource instances to their protocols		
$action(rid, s, A)$	Returns the state by performing A on s w.r.t. rid		
$state(rid, d)$	Returns the set of states of rid in d		

3. Illustrated Examples

In this section, we illustrate some examples that may have potential resource bugs or energy leaks.

```
1 public class Test {
2   public static void main(string[] args) {
3     f = new RandomAccessFile("file", "rw");
4     if(write_cond) f.write("text");
5     if(read_cond) str = f.read();
6     if(close_cond) f.close();
7   }
8 }
```

Figure 1: Snippet Code of File

```
1  public class TestActivity extends Activity {
2    protected void onCreate(Bundle b) {
3      URL url = new URL("http://www.android.com");
4      HttpURLConnection huc =
5        (HttpURLConnection) url.openConnection();
6      if(download_condition) {
7        InputStream out = huc.getInputStream();
8        String str = String.valueOf(out.read());
9        if(use_condition) tv.setText(str);
10       if(download_again) {
11         str = String.valueOf(out.read());
12       }
13     }
14   }
15 }
```

Figure 2: Snippet Android Code of Network

As an example, Figure 1 shows a code snippet about the file resource. This program first opens a file (Line 3). It writes (Line 4) and then reads (Line 5) the file under certain conditions. It closes the file when the *close_cond* condition is met (Line 6). The program is not resource-safe. Firstly, it does not always close the opened file, as *close_cond* (Line 6) may not hold. For instance, an I/O exception is generated by the read or write behaviour and programmers forget to close the file for that case. Secondly, if neither the *write_cond* (Line 4) nor *read_condition* (Line 5) is met, then the opened file will not be used at all. In that case, an unneeded file is created and left open causing resources to be wasted.

With respect to energy consumption (*i.e.*, energy leaks), a resource bug may cause an energy leak. Let us consider the snippet Android code of network in Figure 2, which has some potential energy leaks caused by resource bugs.

This program seems correct, but there are several situations that may cause energy leaks. The first scenario is when the *download_condition* (Line 6) is not met, for instance, a user does not click, in which case the program would not download any data. This indicates that HTTP connection is left open unnecessarily, in which case unnecessary consumption of energy takes place. The second scenario is when the *use_condition* (Line 9) is not met. In that case, the downloaded data would not be used, signifying unnecessary energy consumption (for the unnecessary download). Moreover, if the *download_again* (Line 10) condition is met, the variable *str* would point to the newly downloaded data, leaving the previously downloaded one inaccessible. So the former data is never used, leading to an energy leak. Even worse, if the connection and the input stream are used only to download the unwanted data, then it is clearly unnecessary to open the connection and the input stream. In other words, the program may open the unneeded HTTP connection and input stream to cause unnecessary energy consumption and hence an energy leak. Finally, even they are needed, the connection and the input stream are not closed at last. Thus it remains open to consume energy until the exit of the application. For the benefit of saving energy and according to Javadoc for

HttpURLConnection, it should be closed eventually[2].

4. State-Taint Analysis

In this section, we present a static analysis called *state-taint analysis* to help detect resource bugs. We first give a core language with (explicit) resource usage commands, and then specify resource usage behaviours in terms of resource protocols which depict how resources should be used. After that, we present our state-taint analysis, which takes resource protocols as a guide, to detect resource usage bugs. We also prove the correctness of the proposed analysis and illustrate the analysis via an example.

4.1. A Core Language with Resource Usage

To characterise the resource usages in the examples illustrated in Section 3, we consider a core language with resource usage. While different resources may have different APIs, for example, *openConnection* and *disconnect* are APIs for HTTP connections, *Open*, *Write*, *Read* and *Close* are APIs for Files. These APIs share the same behaviour patterns, namely, *open*, *use*, and *close*. For simplicity, we shall focus on these abstract behaviours. Indeed, we can consider that there is a mapping Label for each resource that maps an API to an (abstract) behaviour. Therefore, the core language with resource usage consists of the following (abstract) statements:

$$\begin{array}{rcl} S & ::= & p = open\ r \mid use\ r\ p \mid close\ r\ p \mid p = q \mid p = q.f \mid p.f = q \\ & \mid & if\ cond\ S\ else\ S \mid while\ cond\ S \mid S;\ S \mid q = c.m(a_1, \ldots, a_n) \end{array}$$

where $p, q \in \mathsf{V}$ are variables, $f \in \mathsf{F}$ is a field name, $r \in \mathsf{R}$ represents a kind of resource, *cond* is a boolean expression, n is the number of parameters of method m in class c, and a_i is an argument of function m. A function definition has the following syntax:

$$fun\ c.m(p_1, \ldots, p_n)\ \{S;\ return\ res\}$$

where p_i is a parameter of function m in class c and *res* is a special variable *res* that holds the return value of the function. A program P consists of a finite number of function definitions and a sequence of statements.

An evaluation environment is a pair (M,B), where M is a memory environment that maps variables to resource instances, and B is a behaviour environment that maps resource instances to behaviour traces. The semantics is defined via the evaluation rule $(M, B) \vdash S \rightarrow (M', B')$, where (M, B) is an evaluation environment before the execution of the statement S, and (M', B') is the evaluation environment after the execution of S. The evaluation rules are shown in Figure 3. Rules (E-Open), (E-Use) and (E-Close) capture the semantics of the behaviours of resources, collecting the resource behaviours into sequences by order: Rule (E-Open) creates a new instance with a single behaviour *open*, and (E-Use) and (E-Close) separately append their behaviours into the behaviour environments of their corresponding resource instances. Similar to [1], we assume that opening a resource always succeeds and returns a new instance. This is because (1) it is not easy to capture the result of opening a resource by static analysis: a simple solution is to add side conditions to specify success or failure into the semantics [7], which may generate too many cases and require guards to be added into our protocols (see Section 5.3); (2) our analysis focuses on the behaviour sequences of resources rather than the behaviour effect (see resource protocols in Section 4.2), although a failed opening can lead to a false positive. Consequently, a behaviour sequence, which may open a unique resource several times, is divided into several ones such that each one starts from *open*. Note that this assumption is not too strong for our analysis, since if these divided behaviour sequences cause some resource bugs, so would the original one, and vice verse (regardless of the failed opening). The (E-Ass i) (i=1,2,3) rules capture the semantics of the assignments. For simplicity, our semantics does not differentiate between stack variables and heap ones. That is, a heap variable such as a field $q.f$ acts like a stack one, except that the heap one $q.f$ may have some aliases, which are represented by $alias(q.f)$. The next five rules deal with the control-flow statements, which are routine and thus are not elaborated here. Finally, considering

[2]There exists a discussion about this close question in *stackoverflow*. Interesting reader can refer to [11].

$$(M,B) \vdash p = open\ r \rightarrow (M[p \mapsto rid], B[rid \mapsto open]), \text{where } rid \text{ is fresh} \quad \text{(E-Open)}$$

$$\frac{M(p) = rid}{(M,B) \vdash use\ r\ p \rightarrow (M, B[rid \mapsto B(rid).use])} \quad \text{(E-Use)}$$

$$\frac{M(p) = rid}{(M,B) \vdash close\ r\ p \rightarrow (M, B[rid \mapsto B(rid).close])} \quad \text{(E-Close)}$$

$$(M,B) \vdash p = q \rightarrow (M[p \mapsto M(q)], B) \quad \text{(E-Ass1)}$$

$$(M,B) \vdash p = q.f \rightarrow (M[p \mapsto M(q.f)], B) \quad \text{(E-Ass2)}$$

$$(M,B) \vdash p.f = q \rightarrow (M[p' \mapsto M(q)], B), \text{where } p' \in alias(p.f) \quad \text{(E-Ass3)}$$

$$\frac{cond \rightarrow true \quad (M,B) \vdash S_1 \rightarrow (M_1, B_1)}{(M,B) \vdash if\ cond\ S_1\ else\ S_2 \rightarrow (M_1, B_1)} \quad \text{(E-If1)}$$

$$\frac{cond \rightarrow false \quad (M,B) \vdash S_2 \rightarrow (M_2, B_2)}{(M,B) \vdash if\ cond\ S_1\ else\ S_2 \rightarrow (M_2, B_2)} \quad \text{(E-If2)}$$

$$\frac{cond \rightarrow true \quad (M,B) \vdash S \rightarrow (M_1, B_1) \quad (M_1, B_1) \vdash while\ cond\ S \rightarrow (M_2, B_2)}{(M,B) \vdash while\ cond\ S \rightarrow (M_2, B_2)} \quad \text{(E-While1)}$$

$$\frac{cond \rightarrow false}{(M,B) \vdash while\ cond\ S \rightarrow (M, B)} \quad \text{(E-While2)}$$

$$\frac{(M,B) \vdash S_1 \rightarrow (M_1, B_1) \quad (M_1, B_1) \vdash S_2 \rightarrow (M_2, B_2)}{(M,B) \vdash S_1;\ S_2 \rightarrow (M_2, B_2)} \quad \text{(E-Seq)}$$

$$\frac{(M[p_i \mapsto a_i], B) \vdash MD(c.m) \rightarrow (M', B')}{(M,B) \vdash p = c.m(a_1, \ldots, a_n) \rightarrow (M'[p \mapsto M'(res)], B')} \quad \text{(E-Fun)}$$

Figure 3: Evaluation rules for statements

the semantics of the function calls, we assume that function definitions are stored in a table *MD* indexed by function names. The evaluation rule for function calls is given by (E-Fun), where $p_1, .., p_n$s are the parameters of function *c.m*. (E-Fun) first evaluates the function body of *c.m*, found from the function table *MD*, and then passes to the variable *p* the result value, held by the special variable *res*, yielding the final environment.

4.2. Resource Protocols

A resource usage protocol specifies how a resource should be used or which behaviour sequences are appropriate. Resource protocols can be viewed as a kind of typestate properties [2], which can be represented as finite state automata. Concerning the *open-but-not-used* problem, a resource intuitively has (at least) three possible states, namely *i/c* (*i.e.*, the resource is on the initial state or closed), *o* (*i.e.*, the resource is opened or required), and *u* (*i.e.*, the resource is used). An abstract automaton for general resource protocols is given in Figure 4, where *O*, *U* and *C* are abstract behaviours denoting the relevant opening (or acquiring), using and closing (or releasing) APIs of resources for short respectively. Behaviour sequences that do not satisfy resource protocols lead to resource bugs. Generally, the appropriate behaviour sequences for a resource should be in form of "*O*, *U*, ..., *U*, *C*", namely OU^+C in regular expressions. Note that there is at least one *use* behaviour between the *open* and *close* behaviours.

Depending on their usages and API documentation, different resources may have different specific automata[3] generated from the abstract one in Figure 4. Take the file resource for example. There are

[3]In implementation, we use a configure file to define the protocols, so a user can define the protocols as they wish.

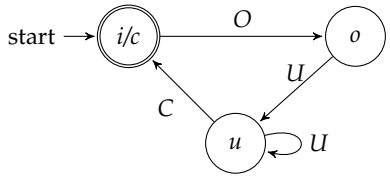
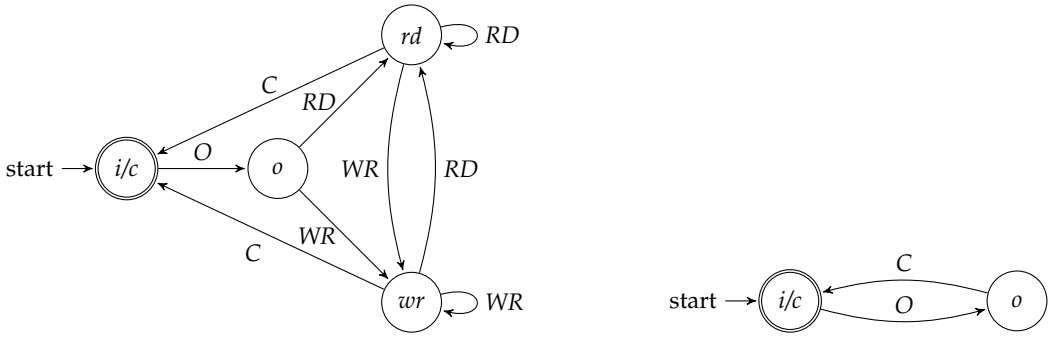


Figure 4: Abstract FSA for General Resources



(a) Specific FSA for File

(b) Specific FSA for WakeLock

Figure 5: Specific FSA for Different Resources

two kinds of usage for a file: *read* and *write*. In some cases, these two behaviours may be regarded as indistinguishable, so the protocol could stay the same as the one in Figure 4. In some other cases, one may want to distinguish between these two usage behaviours, thus the protocol in Figure 5a can be used instead, where the state u is spitted into *rd* and *wr*, and the behaviour U into RD (representing *read*) and WR (representing *write*). Another example is WakeLock, which has only two behaviours, namely, *open* and *close*. Figure 5b gives a specific protocol for WakeLock, which contains only two states.

Assume that there exists a protocol definition function PD which maps resource instances to their corresponding protocols. Let $action(rid, s, A)$ denote the action function of the resource instance rid, which returns the state obtained by performing the behaviour A on the state s if the behaviour succeeds according to the protocol of rid, or bug otherwise:

$$action(rid, s, A) = \begin{cases} s' & \text{if there exists } s' \cdot s \xrightarrow{A} s' \in PD(rid) \\ bug & \text{otherwise} \end{cases}$$

We generalize the action function $action$ to a sequence of behaviours as follows.

$$action(rid, s, A_1 A_2 \dots A_n) = action(rid, action(rid, s, A_1), A_2 \dots A_n)$$

Let $L(PD(rid))$ denote the set of all appropriate behaviour sequences of the resource instance rid. We say a program is resource usage correct if all the behaviour sequences of all the resources satisfy their corresponding protocols.

Definition 4.1. *Let P be a program. If $(\emptyset, \emptyset) \vdash P \to (M, B)$ and $\forall rid \in dom(B).\ B(rid) \in L(PD(rid))$, then P is resource usage correct.*

4.3. Main Analysis

Aiming to detect those inappropriate behaviour sequences that likely lead to resource bugs, we propose a taint-like analysis, consisting of a state-taint analysis algorithm and an exit checking algorithm. Figure 6 shows the framework of our analysis.

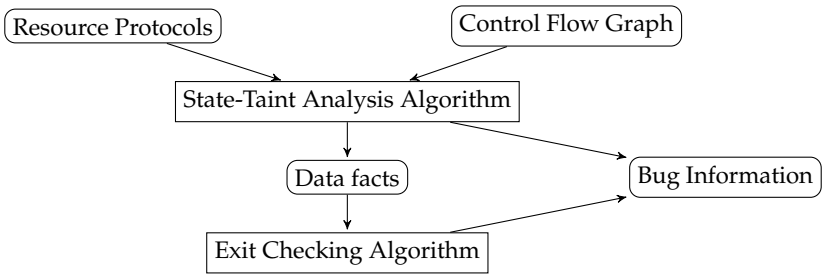


Figure 6: Analysis Framework

The state-taint analysis algorithm combines typestate analysis [2] and taint analysis [9]. Different from taint analysis, our analysis propagates the states of resource protocols among the control flow graph (CFG) of a program instead of simple source tags. During the propagation, typestate analysis is performed meanwhile, that is, states of resource instances will be checked and changed depending on the relevant behaviour according to the corresponding automata. In short, we take resource protocols as a guide to perform a taint-like analysis on CFG.

Algorithm 1 State-Taint Analysis Algorithm

Input: CFG, resource protocols (automata)
Output: the data fact mapping D

for each node $t \in$ CFG **do**
  Let $D(t) = \emptyset$
end for
enqueue each entry point into queue q
while q is nonempty **do**
  $t \leftarrow$ dequeue q
  $d \leftarrow$ apply the transfer function of t on $D(t)$ w.r.t. resource protocols
  for each successor t' of t **do**
    if $d \neq D(t')$ **then**
      $D(t') \leftarrow d \cup D(t')$
      if $t' \notin q$ **then**
        enqueue t' into q
      end if
    end if
  end for
end while
return D

The state-taint analysis algorithm takes as input the control-flow graph (CFG) of a program and the automata specifying the resource usage protocols, and returns a mapping D that maps each node to its data fact. Besides states of the automata, the algorithm also tracks which resource instances are handled and which variables and fields are "tainted". Formally, the data fact (or property) that the analysis tracks is a set of tripes (rid, s, v), meaning that the resource instance rid is at the state s, and may be managed by the variable set v. We represent variables and fields as access paths [12] up to a fixed length. An access path is an expression comprised of a variable followed by a (possibly empty) sequence of field names. For instance, $p.f.g$ represents an access path of length 2. Besides, different paths to a node of CFG can obtain different states for the same resource. As we would like to identify in which path a resource could cause resource bugs, we include different states for the same resource into a set rather than unifying them. That is to say, our analysis is flow-sensitive. Therefore, unlike typestate checking, we do not need to impose a partial order on states, although it can be done easily. The state-taint analysis algorithm is essentially a

classic data flow algorithm, which is shown in Algorithm 1.

The algorithm starts from the entry nodes of CFG with the empty data-fact mapping (Lines $1-4$). For each node t, the algorithm applies its corresponding transfer function, yielding a data fact d to its successors (Line 7). The transfer functions either check whether the resource behaviour conforms to the corresponding resource protocol, or propagate the resource information, which are presented in Section 4.3.1. If the data fact d is different from the original data fact $D(t')$ of a successor t' (Line 9), that is, some data facts are fresh, then the algorithm update $D(t')$ by unioning $D(t')$ and d (Line 10), and enqueue t' into the queue q (Line 12). The algorithm traverses over CFG until q is empty, that is, until the data-fact mapping is no longer updated (Lines $5-15$).

After the data fact mapping is computed, we also check the data fact at the *exit* node of each function to ensure that the resources managed by the local variables are released. The exit checking algorithm is shown in Algorithm 2, where $local(f)$ returns all the local variables of the function f and $PD(rid).accepts$ denotes the set of accepting states of the protocol automaton of the resource instance rid. Note that, the global or static variables are considered as local variables of the *main* function. The function $statecheck(d)$ (defined in Table 2) checks whether the bug state occurs in d, and whether a resource instance at a non-accepting state and not being accessed by any variables exists in d.

Algorithm 2 Exit Checking Algorithm

```
1: for each exit of each function f do
2:   for each (rid, s, v) ∈ D(exit) do
3:     if s ∉ PD(rid).accepts ∧ v ⊆ local(f) then
4:       statecheck (rid, s, v)
5:     end if
6:   end for
7: end for
```

Due to the fresh instance creating of an $open$ behaviour, for an $open$ behaviour occurring in a loop (*e.g.*, while-statement), the algorithm will generate a fresh data fact for this behaviour at each loop, which would cause the algorithm to visit the loop again. However, thanks to the regularity of the behaviour sequences, it suffices to visit the $open$ behaviour in a loop twice for an identity data fact: one is for the *break* case and the other for the *continue* case. Let us consider the statement: $S_1; while\ cond\ \{S_3; p = open\ r; S_4\}; S_2$ and let b_i be the behaviours of resource r in S_i. The possible behaviour sequence of resource r is described by the regular expression $b_1(b_3.open.b_4)^* b_2$. Divided by $open$, the possible behaviour sequences contain b_1b_2 (no loops), b_1b_3 (in the first loop), $open.b_4b_2$ (the *break* case), and $open.b_4b_3$ (the *continue* case). There may be several different resource instances of resource r with the same behaviour $open.b_4b_3$. Since they belong to the same resource r, it is suffice to consider just one instance.

In addition, since the states of the protocol automata, the variables and fields are finite, (and the length of access paths is capped), the triples are finite as well. Therefore, the state-taint analysis always terminates.

4.3.1. Transfer Functions

Prior to presenting the transfer functions, we introduce some auxiliary functions, which are listed in Table 2, where $remove(v, p)$ removes the access paths starting with p in variable set v; $kill(d, p)$ deletes the access paths starting with p in data fact d; $lift(d, p)$ selects the data tripes in d that are accessed by p or an alias of p, and checks whether there are no resource instances accessed by p in some cases; $killalias(d, p, f)$ deletes the access paths in d starting with $p.f$ or the alias of $p.f$; and $statecheck(d)$ checks whether the bug state occurs in d, and whether a resource instance at a non-accepting state and not being accessed by any variables exists in d.

Consider the transfer functions for intra-procedural analysis first, which are listed in Table 3. For the *open* behaviour `p = open r`, it creates a new data tripe $(rid, o, \{p\})$ and kills the data tripes that were managed formerly by p, where rid is fresh. It also checks the newly generated data fact: if a resource is managed by no variables and its state is not an accepting one, then a resource bug is reported. The *use* and *close*

Table 2: Auxiliary Functions

Function	Output
$remove(v, p)$	$v \setminus \{p.\pi \mid p.\pi \in v\}$, π is any access path
$kill(d, p)$	$\{(rid, s, remove(v, p)) \mid (rid, s, v) \in d\}$
$lift(d, p)$	$\{(rid, s, v) \in d \mid alias(p) \cap v \neq \emptyset\}$ if $lift(d, p) = \emptyset$ or $\exists rid.(rid, s_1, v_2) \in lift(d, p) \wedge (rid, s_2, v_2) \in d \setminus lift(d, p)$ then report
$killalias(d, p, f)$	$\{(rid, s, remove(v, q.f)) \mid (rid, s, v) \in d \wedge q \in alias(p)\}$
$statecheck(d)$	$\{(rid, s, v) \mid (rid, s, v) \in d \wedge s \neq bug \wedge v \neq \emptyset\}$ if $s = bug \vee (v = \emptyset \wedge s \notin PD(rid).accepts)$ then report bug

Table 3: Transfer Functions for Intra-Procedural Analysis

Statement	Transfer Function $f(d)$
p = open r	$\{(rid, o, \{p\})\} \cup statecheck(kill(d, p))$
use r p	$statecheck(\{(rid, action(r, s, U), v) \mid (rid, s, v) \in lift(d, p)\}) \cup (d \setminus lift(d, p))$
close r p	$statecheck(\{(rid, action(r, s, C), v) \mid (rid, s, v) \in lift(d, p)\}) \cup (d \setminus lift(d, p))$
p = q	$\{(rid, s, (v \cup \{p.\pi\}) \mid (rid, s, v) \in d \wedge q.\pi \in v\} \cup statecheck(kill(d, p))$
p = q.f	$\{(rid, s, (v \cup \{p.\pi\}) \mid (rid, s, v) \in d \wedge q.f.\pi \in v\} \cup statecheck(kill(d, p))$
p.f = q	$\{(rid, s, (v \cup \{p.f.\pi\}) \mid (rid, s, v) \in d \wedge q.\pi \in v\} \cup statecheck(killalias(d, p, f))$
others	d

behaviours change the states of resources managed by p respectively according to the resource protocol automata. After that, the states are checked: if it is a bug state, report it. The assign statement p = q kills the data fact managed by p, and shares the resource currently managed by q to p (*i.e.*, if q is "tainted", then p is "tainted" as well). Similarly to p = q.f and p.f = q. Note that the alias analysis is triggered by the *use* or *close* behaviours and the assignments to heap variables.

For inter-procedural analysis, we assume that there is a *call* edge from a call statement to each of the possible callees, and there is a *return* edge from the exit of a callee to each of the call statements that could have invoked it. Consider a call statement q = c.m($a_1, \ldots, a_n$). Generally, the call flow function f_{call} will transfer a resource r, managed by an argument a_i, to its corresponding formal parameter p_i. If the caller's context contains a resource r, then f_{call} will also transfer r to the callee's context by replacing c with *this*. The call flow function is:

$$f_{call}(d) = \cup \begin{cases} \{(rid, s, this.\pi) \mid (rid, s, c.\pi) \in d\} \\ \{(rid, s, p_i.\pi) \mid (rid, s, a_i.\pi) \in d\} \end{cases}$$

The return flow function f_{ret} will do the opposite thing instead. Besides, if the return value x is "tainted", then the assignment to q makes it "tainted" as well. The return flow function is given as follows:

$$f_{ret}(d) = \cup \begin{cases} \{(rid, s, c.\pi) \mid (rid, s, this.\pi) \in d\} \\ \{(rid, s, a_i.\pi) \mid (rid, s, p_i.\pi) \in d \wedge \neg immut(a_i)\} \\ \{(rid, s, q.\pi) \mid (rid, s, x.\pi) \in d\} \end{cases}$$

where $immut(a)$ returns *true* iff a is a primitive or immutable data, such as Int, Sting, etc.

Generally, any alias analysis can be adopted here. The more precise the alias analysis is, the more precise result we can get. For instance, with a must-alias analysis fewer but preciser result can be obtained than with a may-alias one. Here we use the on-demand alias analysis adopted by *flowdroid* [9], since it is efficient and context-sensitive.

4.4. Correctness

Since we focus on the resource usage analysis, assume that the transformation from programs to CFGs is correct and the alias analysis is correct[4]. Under that assumption, we prove the correctness of our analysis: if our analysis on a program reports no bug information, then the program is resource usage correct.

We use $analysis(P) \rightsquigarrow_E D$ to denote the whole analysis on a program P, yielding the data fact mapping D and the bug information E, and use $state(d, rid)$ to denote the set of states of the resource instance rid in the data fact d, that is $state(d, rid) = \{s \mid \exists v.\ (rid, s, v) \in d\}$.

Definition 4.2. *Given a behaviour environment B and a data fact d, we say d entails B, denoted as $d \vDash B$, if $action(rid, i, B(rid)) \in state(d, rid)$ for all $rid \in dom(B)$, where i is the initial state of the protocol automaton of rid.*

Lemma 4.1. *Let S_1^n be a sequence of statements $S_1; \ldots S_n$. If $(\emptyset, \emptyset) \vdash S_1^n \rightarrow (M, B)$ and $analysis(S_1^n) \rightsquigarrow_\emptyset D$, then $D(S_n) \vDash B$.*

Proof. By induction on n. It is trivial when $n = 0$. Assume that $n > 0$. In that case, we have $S_1^n = S_1^{n-1}; S_n$. According to the semantics, we have that $(\emptyset, \emptyset) \vdash S_1^{n-1} \rightarrow (M', B')$ and $(M', B') \vdash S_n \rightarrow (M, B)$ for some M' and B'. By induction on S_1^{n-1}, we have $D(S_{n-1}) \vDash B'$. Let us consider S_n by cases.

- p=open r: By the semantics, $B = B'[rid \mapsto open]$, where rid is fresh. So we only need to consider the resource instance rid. According to the transfer functions, $(rid, o, \{p\}) \in D(S_n)$ and for any other rid', $state(D(S_n), rid') = state(D(S_{n-1}), rid')$ (since no bug information is reported). It is clear that $action(rid, i, open) = o$. Hence $D(S_n) \vDash B$.
- use r p: Since $(M', B') \vdash S_n \rightarrow (M, B)$, then there exists rid such that $M'(p) = rid$ and $B = B'[rid \mapsto B'(rid).use]$. So we just consider the resource instance rid. Since $D(S_{n-1}) \vDash B'$, we have $action(rid, i, B'(rid)) \in state(D(S_{n-1}), rid)$. According to the transfer functions, a behaviour use is performed on rid. As no bug information is reported, rid is accessed by p for all cases and this succeeds. Thus $action(rid, i, B'(rid).use) \in state(D(S_n), rid)$, that is, $action(rid, i, B(rid)) \in state(D(S_n), rid)$.
- close r p: Similar to the case of use r p.
- assignments: It is trivial for the stack variables. For the heap variables, the alias information is ensured by the correctness of the alias analysis $alias()$.
- others: Trivial.

□

Theorem 4.2. *Let P be a program. If $analysis(P) \rightsquigarrow_\emptyset D$, then P is resource usage correct.*

Proof. Let the statements in P be S_1^n. Assume $(\emptyset, \emptyset) \vdash S_1^n \rightarrow (M, B)$. By Lemma 4.1, we have $D(S_n) \vDash B$. As no bug information is reported, according to the exit algorithm, all the states in $D(S_n)$ are accepting ones, that is, for all $rid \in dom(B)$, $action(rid, i, B(rid)) \in PD(rid).accepts$. Therefore, for all $rid \in dom(B)$. $B(rid) \in L(PD(rid))$. □

4.5. Example

Consider the file example illustrated in Section 3 again, where the protocol for the file resource is the one in Figure 5a. For convenience, we focus on the resource behaviours and represent them in our abstract statements presented above. Figure 7 gives the CFG of the file example, where d_is are data facts to be computed.

The *open* behaviour generates $(File, o, \{f\})$ (*i.e.*, d_1), which flows to $d_2 - d_5$. While the *write* and *read* behaviours change any state to w and r respectively. For d_4, there are three sources: d_1, d_2 and d_3. According to the protocol in Figure 5a, the *close* behaviour closes d_2 and d_3 normally except for d_1, since the state of

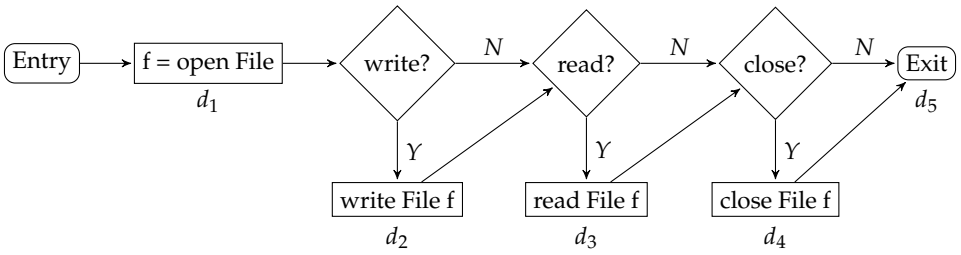


Figure 7: Control Flow Graph of File Example

Table 4: Data Facts for File Example

d_i	**Data Fact**
d_1	$\{(File, o, \{f\})\}$
d_2	$\{(File, w, \{f\})\}$
d_3	$\{(File, r, \{f\})\}$
d_4	$\{(File, c, \{f\}), (File, bug, \{f\})\}$
d_5	$\{(File, o, \{f\}), (File, w, \{f\}), (File, r, \{f\}), (File, c, \{f\}), (File, bug, \{f\})\}$

d_1 is o, indicating that the open file has not been used yet. Finally, all the data facts above flow to the *exit* node, thus d_5 is the union of $d_1 - d_4$, that is, $d_5 = d_1 \cup d_2 \cup d_3 \cup d_4$. All the data facts are shown in Table 4.

Let us check the data fact of the *exit* node. Only the triple $(File, c, \{f\})$ is accepting. The data fact indicates that (i) the file may be opened but not used or closed in some situation (*i.e.*, at the state o); (ii) the file may be opened and then used to read or write, but not closed eventually (*i.e.*, at the state r or w); (iii) the file may be opened and closed correctly without using (*i.e.*, *bug*). These are the possible resource bugs illustrated in Section 3.

In addition, assuming that a type-based method is adopted, one needs to give the type annotations explicitly to all the resource related statements in the program. For this example, a type annotation for files can be $\mu\alpha.(0\&((l_R\&l_W);\alpha));l_C$ [1], which involves several usage constructors and is not easy to follow. Type annotations could be more complex, when a resource is accessed through the invocation of a function closure. Even worse, one may need to understand the whole type system and/or the complex usage constructors to write the annotations correctly.

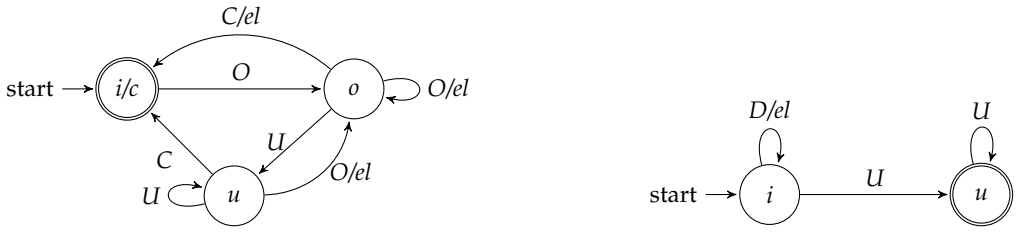
5. Extension: Resource Behaviour Analysis

The resource protocols presented in Section 4 capture the appropriated behaviour sequences. Any sequence not accepted by the protocols would be considered as a potential resource bug. However, different resource bugs may lead to different program bugs. For example, to use an unopened resource may lead to a program crash while reopening an open resource may not. In this section, by enriching the resource protocols with some inappropriate behaviours, we extend our analysis to analyze these behaviours that may lead to special program bugs, for example, energy leaks and information leaks.

5.1. Application: Energy Leaks

Over the last decade, the popularity of smartphones has increased dramatically. This has led to widespread availability of smartphone applications. Since energy is a *scarce* resource for smartphones, mobile applications should be energy efficient. However, many applications are energy inefficient and can suffer from

[4]Many resource instances are managed by stack variables in practice.



(a) Abstract FSA for General Resources

(b) Specific FSA for Downloaded Data

Figure 8: Enriched FSA with Energy Leaks

energy leaks. Existing studies show that eliminating energy leaks can result in a good reduction in energy consumption [8, 13]. Therefore, to detect energy leaks is a meaningful task for smartphone applications. As an application, we show our proposed *state-taint analysis* can be easily adapted to detect energy leaks for smartphone applications.

Enriched Protocols with Behaviours. As discussed in Section 1, a resource bug can cause an energy leak, for example, a resource is left open to consume energy unnecessarily. As also illustrated in the Android code for network in Section 3, the HTTP connection is kept open to consume unnecessary energy. But not all resource bugs lead to energy leaks, for instance, to use a closed resource is a resource bug but may not cause a noticeable energy leak. Therefore, we shall distinguish behaviours that may cause energy leaks from other inappropriate behaviours, and enrich our resource usage protocol with such a distinction.

Figure 8a shows an abstract automaton for the enriched protocol, where *el* denotes there may be an energy leak caused by the corresponding behaviour. Generally, if a non-accepting state of a resource reaches the *exit* node, an energy leak is then caused by this resource, since it is not closed eventually. Moreover, an (already opened) resource should not be opened again, since it is currently not yet used and may remain open to consume energy. Finally, an opened resource should be used before closing. Otherwise, it is an energy waste to leave it open but not used.

In particular, we also consider the usage of downloaded data, since 70% of energy consumption of network resources is due to downloads [8]. The protocol of downloaded data can be represented as a two-states FSA in Figure 8b, where there are no states corresponding to a *close* behaviour and D is the abstract behaviour denoting the download APIs for short.

We extend the action function with energy leaks as follows:

$$action(rid, s, A) = \begin{cases} s' & \text{if there exists } s' \cdot s \xrightarrow{A} s' \in PD(rid) \\ s'/el & \text{if there exists } s' \cdot s \xrightarrow{A/el} s' \in PD(rid) \\ bug & \text{otherwise} \end{cases}$$

Analysis Algorithms. The proposed algorithms would still work for the enriched protocols, with a minor change: the extended action function is used instead and whenever there is an energy leak, it gets reported. Indeed, the extended analysis is exactly the same as the original one, except that the extended one reports more specific bugs than the original one does. That is, if a resource bug is reported by the original analysis, then the extended analysis reports the same bug or a more specific one (e.g., energy leaks). Moreover, if there are no bugs reported by the original one, so does the extended one. This leads to a conclusion that the extended analysis is correct as well.

Theorem 5.1. *If the extended analysis on a program P reports no bug information, then there are no specific bugs (e.g., energy leaks) in P.*

Proof. Similarly to Theorem 4.2, we can prove P is resource usage correct. Hence there are no specific bugs in P. □

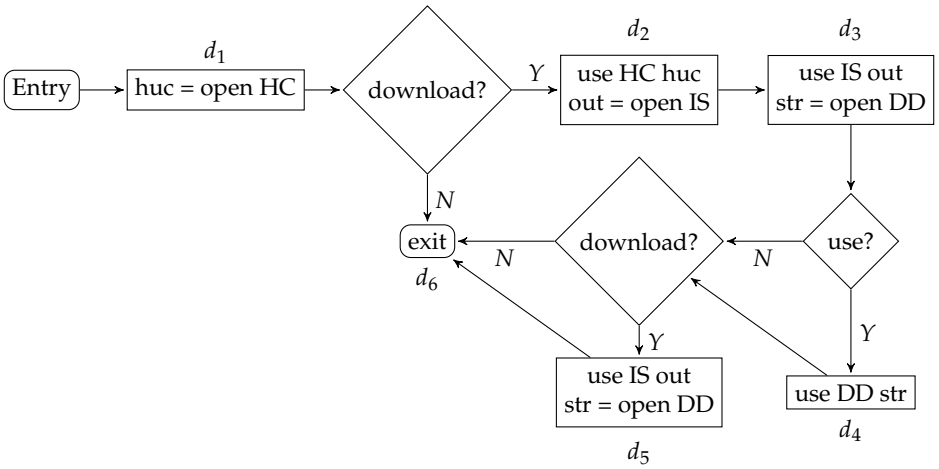


Figure 9: Control Flow Graph of Network Example

Example. Let us apply the resource behaviour analysis to detect energy leaks of the network example illustrated in Section 3, where the enriched protocols for HTTP connection (HC for short) and input stream (IS for short)[5] are in Figure 8a and the one for downloaded data (DD for short) is in Figure 8b. Similarly, we focus on the resource behaviours and represent them in our abstract statements for convenience. Figure 9 gives the CFG of the network example, where d_i are data facts corresponding to each node and are shown in Table 5. Note that the statement $\mathsf{out} = \mathsf{huc.getInputStream()}$ (respectively $\mathsf{str} = \mathsf{out.read()}$) is a use behaviour for HTTP connection (respectively input stream) as well as an open behaviour for input stream (respectively downloaded data).

Consider the data in Table 5. First, the analysis will generate an energy leak for the second *download* behaviour, since the variable *str* may point to some downloaded but unused data. Next, let us check the data fact d_6 at the *exit* node, where only the triple $(DD, u, \{str\})$ is accepting. This data fact d_6 indicates that (i) the HTTP connection may be opened but not used $((HC, o, \{huc\}))$ or closed $((HC, u, \{huc\}))$ in some situation; (ii) the input stream is not closed $((IS, u, \{out\}))$; (iii) the download data may not be used in some situation $((DD, o, \{str\}), (DD', o, \{str\})$, where two different instances correspond to two different download behaviours in the program). All these are the possible energy leaks illustrated in Section 3 earlier.

Table 5: Data Facts for Network Example

Node	**Data Fact**
d_1	$\{(HC, o, \{huc\})\}$
d_2	$\{(HC, u, \{huc\}), (IS, o, \{out\})\}$
d_3	$\{(HC, u, \{huc\}), (IS, u, \{out\}), (DD, o, \{str\})\}$
d_4	$\{(HC, u, \{huc\}), (IS, u, \{out\}), (DD, u, \{str\})\}$
d_5	$\{(HC, u, \{huc\}), (IS, u, \{out\}), el_{(DD,o,\{str\})}, (DD', o, \{str\})\}$
d_6	$\{(HC, o, \{huc\}), (HC, u, \{huc\}), (IS, u, \{out\}),$ $(DD, o, \{str\}), (DD, u, \{str\}), (DD', o, \{str\})\}$

5.2. Application: Information Leaks

Another issue for smartphone applications is the information leak problem: apks leak sensitive user data without the user's permission. According to existing studies, there are dozens of popular Android

[5]By present, we consider each resource separately. We left the embedded resources for future work.

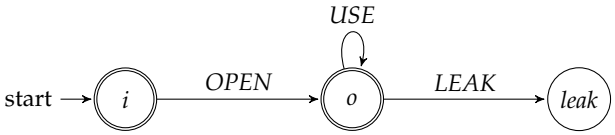


Figure 10: Protocol for Information Leaks

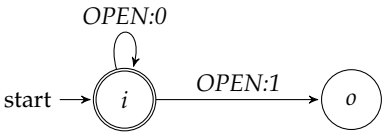


Figure 11: Partial File Protocol with Effects

apps that leak sensitive user data. So to detect information leak is also a meaningful task for smartphone applications.

Indeed, sensitive data is essentially a resource, whose protocol is shown in Figure 10, where *OPEN* is an abstract behaviour that gets the sensitive data (*e.g.*, getIMEI) and *LEAK* is an abstract behaviour that may leak sensitive data (*e.g.*, sendSMS). The protocol states that once there is a *LEAK* behaviour following an *OPEN* behaviour, there may be an information leak.

According to the protocol, to detect information leaks is reduced to find whether there is a path between *OPEN* and *LEAK* in CFG. If we treat *OPEN* behaviours as sources and *LEAK* behaviours as sinks, our analysis becomes the taint analysis.

5.3. Protocols with Guards

The semantics presented in Section 4.1 does not take the effects of the behaviours into account. We should consider the effects for the semantics. For example, if an attempt to open a file fails, the file remains closed, and thus the *read* or *write* behaviour cannot be performed. On the other hand, some behaviours can only be performed conditionally, an example being that the sensitive data can be assessed only by the authorized users. Therefore, another possible extension is to enrich the resource protocols with the performed conditions or the effects of the behaviours, which are described as *guards*. For example, Figure 11 gives a partial protocol for file that takes the effect of $open$ behaviour into account, where 0 and 1 denote failure and success respectively.

As a result, during the analysis, there are two cases for an *open* behaviour: the success case and the failure case. Accordingly, two data triples are generated by $x = open\ r$: $(rid, o, \{x\}, success)$ and $(rid, i, \emptyset, failure)$, where both *success* and $failure$ are guards describing the possible results of $open$. A more precise solution would take into account the satisfiability of guards in the analysis, which is left as future work.

6. Implementation

We have implemented our analysis as a tool called *statedroid* and used it to detect energy leaks for Android applications. The tool consists of a front-end and a back-end, as shown in Figure 12. The front-end parses an apk file and then builds a control flow graph, which is passed to the back-end as input. Our tool's front-end is very similar to that of *flowdroid*, whose front-end provides almost the same functionalities as ours. Interesting readers can refer to [9] for more details. While the back-end takes as input the control flow graph built by the front-end and the resource protocols, and performs state-taint analysis and exit checking presented in Section 4. Our state-taint analysis is implemented upon the IFDS framework [14].

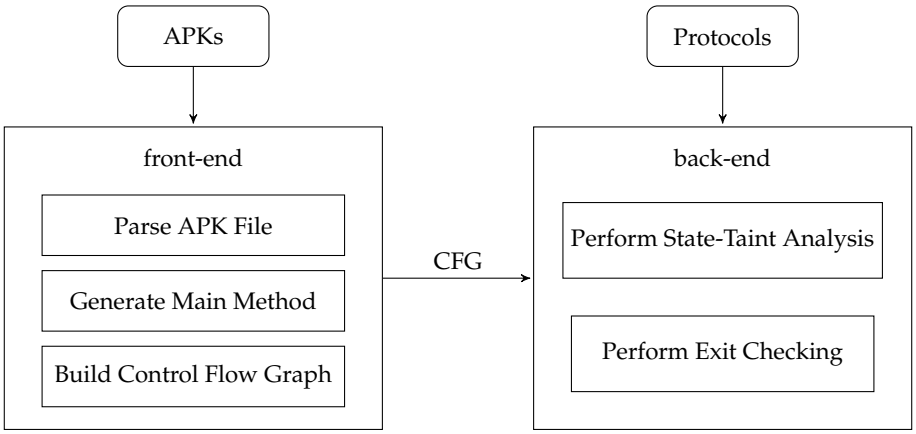


Figure 12: Architecture of *statedroid*

7. Experimental Evaluation

To evaluate our analysis, we have conducted a series of experiments on real Android applications to detect energy leaks by using *statedroid*, where the applications are collected from F-Droid, a test dataset of Relda [15] and a test dataset of GreenDroid [16]. All the experiments have been conducted on a machine with Intel core I5 CPU and 4GB RAM, running Ubuntu 14.04.

7.1. Energy Leaks from F-Droid

The first experiment is to address how precise our tool is. For that, we have collected 100 real Android applications from F-Droid, a well-known open source Android application repository. We have used our tool *statedroid* to detect energy leaks for each application, taking as a guide the resource protocols for HTTP connection (Figure 8a), WakeLock (Figure 5b) and Media Player (Figure 8a) respectively. We have also performed manual analysis on the high-level source codes of these applications where energy bugs are reported. Please note that, since we use the soot framework to analyse Android applications in our implementation, there exist too many bugs caused by exceptions. By now, as they are not easy to check by hand, we decide not to take exceptions into account at the current stage but leave it to future work.

Table 6 summarises our findings of these 100 applications, where *O* denotes an opened resource which is neither used nor closed, *U* denotes an opened resource which is used but not closed, *C* denotes a resource that is closed eventually without using, *T* denotes the total resource bugs reported by our tool, *FP* denotes the resource bugs reported by our tool but cannot be rediscovered manually, *R* denotes the resource kind that causes the energy leak, and *HC*,*WL* and *MP* represent HTTP connection, WakeLock and Media Player for short respectively. Note that there are no use behaviours for WakeLock.

The results show that among these 100 applications, our tool reports that (1) 18 applications have 102 energy leaks caused by HTTP connection; (2) 6 applications have 6 energy leaks caused by WakeLock; (3) 6 applications have 7 energy leaks caused by Media Player; and (4) the precision rating of our tool is about 83.5%. From the results, we also can see that lots of bugs are due to the unclosed-ness (*i.e.*, *O* and *U*). We believe the main reason is that programmers are prone to forget to close/release the resource for every exit. For example, the application *hydromemo* creates a Medie Player but does not release it finally, while the application *betterwifionoff* creates two WakeLocks (*i.e.*, one for wifi and the other for screen) but only releases the one for screen. Moreover, there are also many bugs due to the unused-ness (*i.e.*, *O* and *C*). This is mainly because programers are likely to open the resource in advance and close the resource at the last minute without considering it is in need or not. For example, the application *bib* opens the HTTP connection just to check the connection without using it.

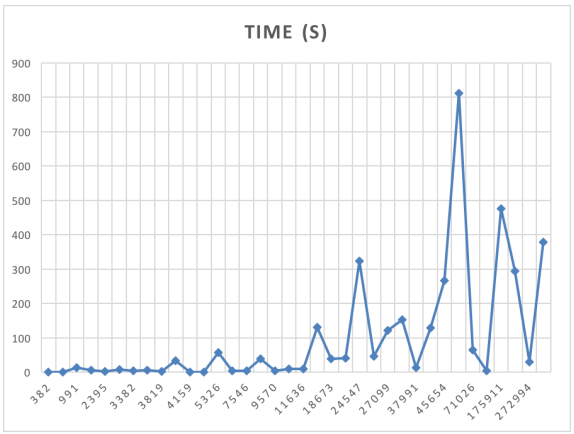
Through the manual analysis on the high-level source codes of these applications where energy bugs are reported, we found that 19 energy bugs turn out to be false positives. There are several reasons for these

Table 6: Results for Selected Apks from F-Droid

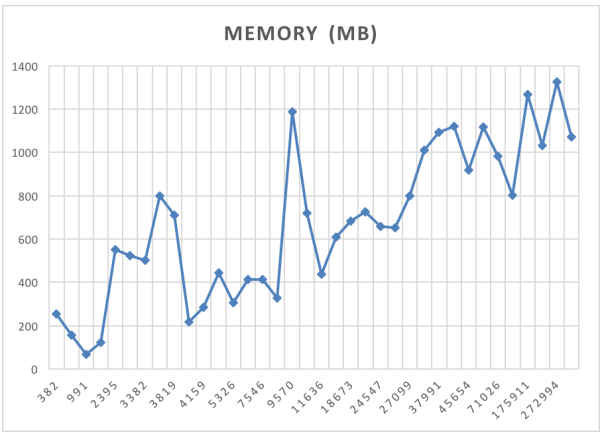
APK	*O*	*U*	*C*	*T*	*FP*	*R*	*APK*	*O*	*U*	*C*	*T*	*FP*	*R*
antennapod	1	4	0	5	0	HC	bible	3	0	2	5	2	HC
coolreader	2	1	0	3	0	HC	cordova	12	2	2	16	0	HC
derbund	14	0	0	14	0	HC	gearshift	3	1	1	5	1	HC
giga	1	0	0	1	0	HC	goblim	1	1	0	2	0	HC
impeller	21	1	0	22	0	HC	kontalk	6	1	2	9	1	HC
lico	2	0	0	2	0	HC	mirakel	2	2	0	4	0	HC
mirrored	1	2	1	4	4	HC	movement	2	0	0	2	1	HC
muzei	2	0	0	2	2	HC	openmensa	0	3	0	3	0	HC
remote	0	1	0	1	0	HC	tether	1	1	0	2	2	HC
adbwireless	1	-	-	1	1	WL	androbe	1	-	-	1	1	WL
betterwifi	1	-	-	1	0	WL	fbreader	1	-	-	1	1	WL
smarterwifi	1	-	-	1	1	WL	ttrssreader	1	-	-	1	0	WL
hydromemo	1	0	0	1	0	MP	impeller	1	0	0	1	0	MP
tasks	1	0	0	1	0	MP	smspopup	1	0	0	1	1	MP
babymonitor	0	1	0	1	1	MP	imcktg	1	1	0	2	0	MP
Total	85	22	8	115	19	-	Ratio	73.9%	19.1%	7.0%	1	16.5%	-

19 false positives. The first one is the null pointer checking, due to which 57.9% (*i.e.*, 11) false positives are generated. However, since we maintain some relations between resources and variables, we can improve our analysis to check the null pointer for some variables by these relations, which would reduce the false positives and is left for future work. The second reason is that a variable such as *state* is used to simulate the status of resources, and different actions are allowed to depend on this variable, such as the applications *adbwireless* and *fbReader*. Our analysis does not catch this variable in its current form. The third reason is that a type checking is performed before closing, for example, *con instanceof HttpURLConnection* is performed before closing *con* in application *mirrored*. Another reason is due to the lifecycle of Android. For example, in the application *babymonitor*, the callback function *onCompletion*, which will release the Media Player, is invoked when the Media Player finishes the playing, but our tool can not catch this callback relation at the current stage.

Moreover, we have investigated the precision rates of several static analysises on energy leaks from their experimental studies: (1) the precision rate of the work of Pathak *et al.* on no-sleep bugs [17] is about 76.4% (55 no-sleep bugs with 13 false positives found in 86 apks), which is lower than ours. (2) the precision rate of Relda [15] is about 88.0% (92 energy leaks with 81 true positives found in 43 apks), which is slightly higher than ours. But Relda is flow-insensitive. (3) Relda2 [18] supports both flow-sensitive analysis and flow-insensitive analysis. Their precision rates are about 68.1% (69 energy leaks with 47 true positives found in 76 apks) and about 55.4% (121 energy leaks with 67 true positives found in 76 apks) respectively, both of which are lower than ours. We also compare our tool and Relda2 on a same dataset, and the result shows that our tool is more precise than Relda2 (see Section 7.3 for detail). (4) The work of Wu *et al.* [19] summarizes two precise patterns for run-time GUI-related energy-drain behaviours and performs their analysis on 15 apks which are known to suffer from energy leaks, thus it has a high precision: about 90.9% for the first pattern (11 energy leaks with 10 true positives found in 15 apks) and almost 100% for the second pattern (17 real energy leaks found in 15 apks). Some of 15 apks are from GreenDroid, which can be detected by our tool as well (see Section 7.3). Although our tool seems to be less precise than this one, we consider more usage patterns. In a nutshell, our tool is reasonably precise and can detect more usage patterns, and thus would be very helpful in practice.



(a) Time for Different Apks



(b) Memory for Different Apks

Figure 13: Overhead for Different Apks

7.2. Performance

The second experiment is to address the performance of our tool. To do that, we have selected applications whose lines of code (LOC) range from 300 to 300000 from F-droid and performed our tool *statedroid* to analyze each application, taking as a guide the resource protocols for HTTP connection. During the analysis, we calculate the running time in seconds and the memory overhead in MBs that are needed by each application.

Figure 13 shows the overhead only for those applications reported to suffer from energy leaks. The running time for these applications ranges from 0.69s to 811.71s with an average of 98.10s, and the memory overhead is from 68.6MB to 1325.41MB with an average of 675.25MB. It signifies that our tool does not cost too much time or memory, which indicates that our tool is quite suitable in practice. The figure also shows the overall trend of the overhead: the larger the application, the more the time and memory overhead. Note that there are some anomalous cases, due to fewer resource behaviours occurring in the applications. Indeed, if an application contains no resource-related behaviours, the tool returns with no bugs immediately.

7.3. Experiments on datasets from Relda and GreenDroid

Both Relda [15, 18] and GreenDroid [16], two recent tools for detecting energy leaks for Android applications, consider partial usages of resources, hence, they cannot detect the energy leaks caused by some other usages like *open-but-not-used* (such energy leaks are denoted by *C* in Table 6). In the following, we perform our tool on the datasets from Relda and GreenDroid under the same resource usage patterns used by Relda and GreenDroid to compare them more.

Relda. We have performed our tool and Relda2 (with flow-sensitive setting) on a data set from Relda, the set of 55 applications from an Android application development book [20], taking as a guide the same resources as Relda2, where the *use* behaviours of resources are omitted. Both Relda2 and our tool report the same applications which may possibly suffer from energy leaks[6]. The detailed results are shown in Figure 7, where *E* denotes the number of the energy leaks reported by our tool or Relda2, *BT*, *CM*, *LM*, *MR* and *WM* represent Bluetooch, Camera, LocationManager, MediaRecorder and WifiManager for short respectively, the other notations are the same as Table 6. The results show that: (1) our tool is more precise than Relda2: through the manual analysis, there are 4 false positives for Relda2, while our tool has only 1. (2) our tool detects two more energy leaks in *ase*, one of which is caused by WakeLock and the other

[6]Relda1 only detects four energy leaks from this dataset [15], while our tool can detect two more real energy leaks than Relda1 under the resources that Relda1 detects.

by Bluetooth. Note that Relda2 also detects two energy leaks caused by MediaRecoder in *ase*, but they are false positives. (3) Relda2 seems to report more bugs than our tool does, but indeed these bugs are almost the same. This is due to the different measurements: Relda2 counts the bugs by resource behaviours (*i.e.*, APIs), while our tool counts by paths, which contain as many behaviours as possible.

Table 7: Comparison with Relda2

APK	*statedroid*			*Relda2*		
	E	*FP*	*R*	*E*	*FP*	*R*
ase	2	0	WL, BT	2	2	MR
chess	1	0	MP	3	0	MP
myAudioManger	1	0	MP	2	0	MP
myBluetooth	1	0	BT	1	0	BT
myCamera	1	0	CM	4	0	CM
myLocation	1	0	LM	1	0	LM
myMusicPlayer	1	0	MP	2	0	MP
myMusicPlayOnline	1	0	MP	3	0	MP
myRecord	1	1	MR	2	2	MR
mySdcardMusicPlay	1	0	MP	7	0	MP
mySdcardVideoPlay	1	0	MP	3	0	MP
mySeekPlay	1	0	MP	3	0	MP
myVideoMake	1	0	MR	2	0	MR
myVideoPlayOnline	1	0	MP	3	0	MP
myWifi	1	0	WM	1	0	WM
Total	16	1	-	39	4	-

In addition, taking several *use* behaviours of resources into account, our tool can detect two more real energy leaks: one is from *ase*, due to that Bluetooth is opened and used, but not closed (which can be detected by Relda); the other is from *mySeekPlay*, due to that MediaPlayer is opened and later closed, but has never been used (which cannot be detected by Relda).

GreenDroid. We have also performed our tool on the Android applications that suffer from energy leaks due to Lock and sensors detected by GreenDroid[7] [21]. The results are shown in Table 8, where the notations are the same as Table 7. Our tool can detect almost all the energy leaks that are detected by GreenDroid, except for the one in *ecycle-locator*, which is because the class *MapView* can not be resolved by the soot framework used in our tool. This is an implementation problem.

Table 8: Result on Dataset from GreenDroid

APK	*R*	*E*	*APK*	*R*	*E*
DroidAR	LM	1	ecycle-locator	LM	0
Sofia Public Trasport Nav	LM	1	Ushahidi	LM	1
EbookDroid	WL	1	AndTweet	WL	1
BabbleSink	WL	1	CWAC-Wakeful	WL	1

GreenDroid adopts a dynamic analysis, and thus it can detect the underutilization problem quite well. While our tool adopts a static analysis instead so that it cannot capture this underutilization problem.

[7]We have asked for GreenDroid but are not able to run it on our dataset.

However, our tool considers more resource usages than GreenDroid, consequently it can detect more energy leaks caused by inappropriate resource usages, for example, the energy leaks denoted by *C* in Table 6 and the one due to the unused MediaPlayer in *mySeekPlay*.

8. Related Work

There are many related works about resource usage analysis, such as resource management, API usages, and energy bugs. Here we discuss some of the most related ones.

Resource Usage Analysis. DeLine and Fähndrich [3] proposed a type system to keep track of the state (called a *key*) of a resource. The state of a resource determines what operations can be performed on the resource, and the state changes after operations are performed. Therefore, keys in their type system roughly correspond to the states of protocols in our *state-taint analysis*. However, their keys did not consider the *open-but-not-used* issue, that is, an opened resource should be used. Moreover, their type system requires programmers to provide explicit type annotations (including keys) to guide an analysis, which is not so easy to give.

Igarashi and Kobayashi [1] formalized a general problem of how each resource is accessed as a resource usage analysis problem, and proposed a type-based method to check whether the inferred resource usage matches the programmer's intention. As a follow-up, Kobayashi refined their type-based analysis by introducing a new notion of *time regions* [6] , which can express how often an action can perform in a region of time. Their type system is powerful and can deal with concurrent access to a resource. However, the type system is too complex to use in practice for detecting resource bugs or energy leaks.

Kang *et al.* [7] presented a path sensitive type system for resource usage verification. They introduced typing rules for conditions in branches to determine the boolean value of condition as possible. In contrast, our analysis just union branches simply, thus is approximated. For example, `if (f != null) then f.close()` is correct for their type system, but is considered problematic in our analysis. Nevertheless, we can improve our analysis to handle some *branches* with the consistent checking in [22]. Our price is very low, while their price is that they have to put lots of information into types.

Marriott *et al.* [5] specified the resource usage policy as an automaton and the program as a context-free grammar, and then checked whether the language of the grammar is contained in the automaton. This is very similar to our analysis. But our approach can analyse several resources at the same time while their analysis does one resource at a time. Besides, they did not consider the *open-but-not-used* issue.

Torlak and Chandra [22] presented an effective data analysis to detect resource leaks. Their analysis is very close to our analysis, but our analysis can detect not only resource leaks, but also other resource bugs and energy leaks.

Fink *et al.* [23] proposed an effective typestate verification in the presence of aliasing to check correct API usage for various Java standard libraries. Their verification tracks a must-alias set, a may-alias set and a must-not-alias set meanwhile, thus it can handle aliasing very well. In contrast, our approach considers just one must or may alias set. Although less precise, ours incurs lower cost and is simpler to use for detecting resource bugs. Moreover, their verification did not ensure that a close API should be called for a resource eventually nor that a use API should be called for an opened resource.

Besides, there are some other works that analyse or verify the bound usage or size property of resources [24, 25, 26, 27, 28], while our analysis concerns about the correct usage of resource (*e.g.*, to avoid energy leaks).

Energy Bugs. There are many solutions proposed to detect energy bugs for applications. Most of them rely on energy profilers to record resource usage and relevant events or codes. Although they can identify some energy leaks and energy hots, they may not find the root causes for the bugs. Interesting readers can refer to [29] for more detail. Here we discuss several latest solutions that adopt program analysis, and consider the root causes of energy leaks.

Pathak *et al.* [17] proposed a data flow analysis to check Wakelock API (*i.e.*, *on* and *off*) to find no sleep bugs. Guo *et al.* [15] built a function call graph and then checked whether *require* and *release* actions are matched, which is extended by Wu *et al.* [18, 30] to an inter-procedural and callback-aware one. The

resource protocols considered by these methods are simper than our analysis, with only *open* and *close* states.

Zhang *et al.* [8] presented a dynamic taint-tracking to detect energy leaks resulting from unnecessary network communication. Liu *et al.* [16] used Java Path Finder not only to monitor sensor and wakelock operations to detect missing deactivation of sensors and wakelocks, but also tracked the utilization of sensory data. Compared to our analysis, these methods considered partial usages of network, sensors and wakelocks. Wu *et al.* [19] defined two precise patterns of run-time energy-drain behaviours (*i.e.*, lifetime containtment and long-wait state) and presented a static analysis to detect these two patterns. The first pattern indeed is a require-but-not-released behaviours and thus can be encoded by our protocols, while the second one concerns the lifecycle and cannot be encoded by our protocols.

9. Conclusion

We have proposed a *state-taint analysis*, guided by user-specified resource usage protocols, for easy detection of resource usage bugs. The analysis is general as it can work with different customised resource usage protocols. As an application, we have shown the proposed *state-taint analysis* can be readily extended to detect energy leaks for Android applications. To demonstrate the viability of the approach, we have implemented the analysis in a prototype tool and carried out some interesting experiments.

As for future work, we may consider the null pointer checking to reduce false positives. We can improve the protocols and/or the analysis to take into account the guards and the relations between resources. We can also improve the current analysis by introducing partial orders on states to merge the data fact for same resource but different states, which may benefit the analysis for large applications.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving the final version of the paper. They would also like to thank the editors for their generous comments and support during the review process. This work was partially supported by the National Natural Science Foundation of China under Grants No. 61502308 and 61373033.

References

[1] A. Igarashi, N. Kobayashi, Resource usage analysis, in: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '02, 2002, pp. 331–342.

[2] R. E. Strom, S. Yemini, Typestate: A programming language concept for enhancing software reliability, IEEE Transactions on Software Engineering 12 (1) (1986) 157–171.

[3] R. DeLine, M. Fähndrich, Enforcing high-level protocols in low-level software, in: Proceedings of ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01, 2001, pp. 59–69.

[4] J. S. Foster, T. Terauchi, A. Aiken, Flow-sensitive type qualifiers, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Vol. 37 of PLDI '02, 2002, pp. 1–12.

[5] K. Marriott, P. Stuckey, M. Sulzmann, Resource usage verification, in: Proceedings of the 1st Asian Conference on Programming Languages and Systems, Vol. 2895 of APLAS '03, 2003, pp. 212–229.

[6] N. Kobayashi, Time regions and effects for resource usage analysis, Acm Sigplan Notices 38 (3) (2003) 50–61.

[7] H.-G. Kang, Y. Kim, T. Han, H. Han, A path sensitive type system for resource usage verification of c like languages, in: Proceedings of the Third Asian Conference on Programming Languages and Systems, APLAS '05, 2005, pp. 264–280.

[8] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, L. Yang, ADEL: An automatic detector of energy leaks for smartphone applications, in: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '12, 2012, pp. 363–372.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, 2014, pp. 259–269.

[10] Z. Xu, D. Fan, S. Qin, State-taint analysis for detecting resource bugs, in: Proceeding of the10th International Symposium on Theoretical Aspects of Software Engineering, TASE '16, 2016, pp. 168–175.

[11] Do I need to call HttpURLConnection.disconnect after finish using it, `http://stackoverflow.com/questions/11056088/do-i-need-to-call-httpurlconnection-disconnect-after-finish-using-it`. Jan 15, 2016.

[12] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, S. Guarnieri, Andromeda: Accurate and scalable security analysis of web applications, in: Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering, Vol. 7793 of FASE 13, 2013, pp. 210–225.

[13] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, G. M. Voelker, edoctor: Automatically diagnosing abnormal battery drain issues on smartphones, in: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 13, 2013, pp. 57–70.

[14] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, 1995, pp. 49–61.

[15] C. Guo, J. Zhang, J. Yan, Z. Zhang, Y. Zhang, Characterizing and detecting resource leaks in android applications, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, 2013, pp. 389–398.

[16] Y. Liu, C. Xu, S. Cheung, J. Lu, Greendroid: Automated diagnosis of energy inefficiency for smartphone applications, IEEE Transactions on Software Engineering 40 (9) (2014) 911–940.

[17] A. Pathak, A. Jindal, Y. C. Hu, S. P. Midkiff, What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps, in: MobiSys '12, 2012, pp. 267–280.

[18] T. Wu, J. Liu, X. Deng, J. Yan, J. Zhang, Relda2: An effective static analysis tool for resource leak detection in android apps, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, ACM, New York, NY, USA, 2016, pp. 762–767.

[19] H. Wu, S. Yang, A. Rountev, Static detection of energy defect patterns in android applications, in: Proceedings of the 25th International Conference on Compiler Construction, CC 2016, ACM, New York, NY, USA, 2016, pp. 185–195.

[20] S. Liu, Source code materials of Android mobile application development from beginner to the master (in Chinese), `http://www.tdpress.com/zyzx/tsscflwj/268575.shtml`. Dec 12, 2015.

[21] Android Applications detected by GreenDroid to Suffer Energy Leaks, `http://sccpu2.cse.ust.hk/greendroid/detected_problems.html`. Apr. 18, 2017.

[22] E. Torlak, S. Chandra, Effective interprocedural resource leak detection, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, 2010, pp. 535–544.

[23] S. Fink, E. Yahav, N. Dor, G. Ramalingam, E. Geay, Effective typestate verification in the presence of aliasing, in: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA 06, 2006, pp. 133–144.

[24] W.-N. Chin, H. H. Nguyen, S. Qin, M. Rinard, Memory usage verification for oo programs, in: Proceedings of the 12th International Conference on Static Analysis, SAS '05, 2005, pp. 70–86.

[25] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, H. H. Nguyen, Verifying safety policies with size properties and alias controls, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, 2005, pp. 186–195.

[26] G. He, S. Qin, C. Luo, W.-N. Chin, Memory usage verification using hip/sleek, in: Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis, ATVA '09, 2009, pp. 166–181.

[27] M. Bartoletti, P. Degano, G. L. Ferrari, R. Zunino, Types and effects for resource usage analysis, in: Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 07, 2007, p. 3247.

[28] W.-N. Chin, H. H. Nguyen, C. Popeea, S. Qin, Analysing memory resource bounds for low-level programs, in: Proceedings of the 7th International Symposium on Memory Management, ISMM '08, 2008, pp. 151–160.

[29] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, S. Tarkoma, Modeling, profiling, and debugging the energy consumption of mobile devices, ACM Comput. Surv.

[30] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, J. Zhang, Light-weight, inter-procedural and callback-aware resource leak detection for android apps, IEEE Transactions on Software Engineering 42 (11) (2016) 1054–1076.