State-Taint Analysis for Detecting Resource Bugs

Zhiwu Xu[1] Dongxiao Fan[1] Shengchao Qin[2]
[1] College of Computer Science and Software Engineering, Shenzhen University, China
[2] School of Computing, Teesside University, UK
Email: xuzhiwu@szu.edu.cn, fan123199@gmail.com, s.qin@tees.ac.uk

***Abstract*—To verify whether a program uses resources in a valid manner is vital for program correctness. A number of solutions have been proposed to ensure such a property for resource usage. But most of them are sophisticated to use for resource bugs detection in practice and do not concern about the issue that an opened resource should be used. This *open-but-not-used* problem can cause resource starvation in some case as well. In particular, resources of smartphones are not only scarce but also energy-hungry. The misuse of resources could not only cause the system to run out of resources but also lead to a shorter battery life. That is the so-call *energy leak* problem.**

Aiming to provide a lightweight method and to detect as many resource bugs as possible, we propose a *state-taint analysis* in this paper. First, take the *open-but-not-used* problem into account, we specify the appropriate usage of resources as resource protocols. Then we propose a taint-like analysis which takes resource protocols as a guide to detect resource bugs. As an application, we enrich the resource usage protocols by taking into account 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 find several energy leaks.

I. Introduction

Resource usage [1] is one of the most important characteristics of programs. To verify whether a program uses resources in a valid 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, and an opening camera (a popular resource for smartphones nowadays) should be closed eventually.

A number of static analyses have been proposed to analyse the correct usages of computer resources [1]–[7]. Most of them adopt a type-based method to ensure a *resource-safe* property. However, first, although sound, they either require rather complex program annotations to guide the analysis or are sophisticated to use for resource bugs detection in practice, since one needs to enhance a type system with resource usage information, which is not easy to follow. Second, few of them concern about that an opening resource should be *used* before its closing. While in some cases it may cause no harm or just minor problems to open/obtain a resource but then leave it unused/unattended, in other cases this may lead to more severe problems. For instance, when some resource is very limited and is not released timely, this issue can lead to a major problem, causing resource starvation and severe system slowdown or instability. Specifically, for mobile devices, such as tablets and smartphones, which have become hugely popular, the resource is rather limited, specially 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. The opening of these resources could be more expensive, as they would consume energy continuously to make a shorter battery life. 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. First, take the *open-but-not-used* problem into account, we specify the appropriate usage of resources as resource protocols according to the API documentation of resources. A resource protocol describes how a resource should be used or which action sequences are appropriate. Resource protocols can be viewed as a kind of typestate properties [2] , which can be represented as finite state automata. Different resources can have different specific automata. An action sequence that does not satisfy resource protocols is considered as a resource usage bug.

Intending to be used easily in practice and to find as many bugs as possible, we propose a static analysis, which takes a control flow graph (CFG) of a program as input, and guided by the resource protocols to track the resource actions among CFG. Following the idea of taint-analysis [9], our analysis propagates the states of resources among CFG. During the propagation, our analysis will check whether the resource actions confirm to the corresponding resource protocols (*i.e.* automata). In detail, our analysis will verify that (i) whether all the resource actions obey the resource protocols before the exit of CFG, and (ii) whether all the states of resources reaching the exit of CFG are accepted following the resource protocols. Our analysis is path-sensitive, so states for one resource from different paths may be different

and preserved.

Moreover, energy leak becomes a critical concern for smartphone applications. As an application, we use *state-taint analysis* to detect energy leaks for smartphone applications. Usually, a resource bug can cause an energy leak, if the bug keeps the resource open unnecessarily. We distinguish the action sequences that may cause energy leaks from the other inappropriate ones, and improve 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 improved protocols as a guide, we thus can use *state-taint analysis* to detect energy leaks in smartphone applications.

Finally, we also implement the analysis as a tool called *statedroid*. Using this tool, we conduct experiments on 100 real Android applications collected from F-Droid, and find that 18 applications have energy leaks.

The contributions of our work can be summarised as follows:

First, we specify the appropriate usage of resources as resource protocols to ensure that an opened resource should be used before closing and propose a static analysis called *state-taint analysis*, which takes resource protocols as a guide to detect as many resource bugs as possible. Compared with the existing works [1]–[7], which adopt type-based method, our analysis (i) is lightweight and easier to use, yet still sound, and (ii) helps detect the *open-but-not-used* problem.

Second, as an application, we use *state-taint analysis* to detect energy leaks in Android applications. we analyse which resource bugs could cause energy leaks, and improve the resource protocols with them to be a guide for the analysis.

Third, we implement the analysis as a tool called *statedroid*. Then we use it to conduct experiments on several real Android applications and find several energy leaks.

The rest of the paper is constructed as follows: Section II illustrates some examples that have potential resource bugs or energy leaks. Section III presents the main algorithms of our analysis. Section IV gives an application of using the analysis to detect energy leaks. Section V presents the selected implementation and experiments. Section VI reviews related work and Section VII concludes the paper.

II. Illustrated Examples

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

As an example, Figure 1 shows a code snippet about the file resource. This program first opens a file (Line 3). Then it writes (Line 4) and reads (Line 5) the file conditionally. Finally, it close the files conditionally as well (Line 6). The program is not resource-safe. First, it does not always close the opened file, as the *close_condition* (Line 6) may not hold. For instance, an I/O exception is generated by the read or write action and programmers

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

Fig. 1. Snippet Code of File

forget to close the file for that case. Second, if neither the *write_condition* (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 to cause resource wasting.

Considering energy consumption (*i.e.*, energy leaks), a resource bug can cause 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. First, if the *download_condition* (Line 6) is not met, for instance, a user does not click, then the program would not download any data. This indicates that HTTP connection is left open unnecessarily, in which case unnecessary waste of energy takes place. Second, the *use_condition* (Line 9) may not be met either. In that case, the downloaded data would not be used, signifying unnecessary energy consumption (for the unnecessary downloading). Moreover, if the *download_again* (Line 10) condition is met, then the variable *result* would point to the newly downloaded data, leaving the previously downloaded one inaccessible. So the former data is never used, leading to 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 waste energy, yielding 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 saving energy and according to Javadoc for *HttpURLConnection*, it should be closed eventually[1].

III. State-Taint Analysis

In this section, we present a static analysis called *state-taint analysis* to detect resource bugs. First, we specify the usage of resources as resource protocols, which describe how a resource should be used. Second, we propose a static analysis, which takes resource protocols as a guide, to detect resource usage bugs. Finally, we present examples to illustrate our analysis.

[1]There exists a discussion about this close question in *stackoverflow*. Interesting reader can refer to [10].

```
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 result = String.valueOf(out.read());
9        if(use_condition) tv.setText(result);
10       if(download_again) {
11         result = String.valueOf(out.read());
12       }
13     }
14   }
15 }
```

Fig. 2. Snippet Android Code of Network

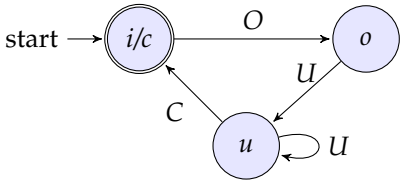

Fig. 3. Abstract FSA for General Resources

A. Resource Protocols

A resource usage protocol specifies how a resource should be used. Different resources can have different protocols. Action sequences that do not satisfy resource protocols may lead to resource bugs. Resource protocol 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 states, namely *i/c* (*i.e.*, the resource is in 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 shown in Figure 3, where *O*, *U* and *C* are abstract actions denoting the relevant opening (or acquiring), using and closing (or releasing) APIs of resources for short respectively. Usually, the appropriate action sequences for a resource should be in form of "*O*, *U*, ..., *U*, *C*", namely OU^+C in regular expression. Note that there is at least one *use* action between the *open* and *close* actions.

Depended on their usages and API documentation, different resources can have different specific automata generated from the abstract one in Figure 3. Take the file resource for example. There are two kinds of usage for a file: *read* and *write*. These two usages seem the same in some cases, so the automaton could be the one in Figure 3. While in other case, users would like to distinguish these two usages, thus the automaton in Figure 4 can be used instead, where State *u* is spitted into *rd* and *wr*, and Action *U* into *RD* (representing *read* action) and *WR* (representing *write* action). Moreover, the protocol

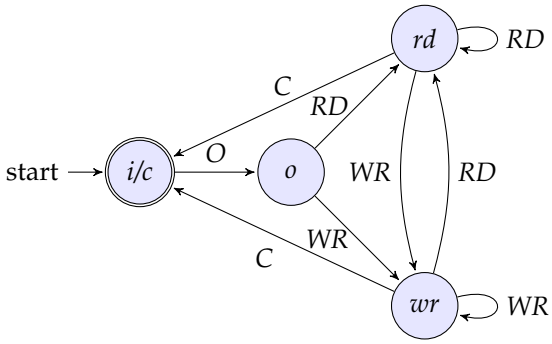

Fig. 4. Specific FSA for File

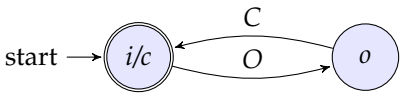

Fig. 5. Specific FSA for WakeLock

for WakeLock can be the specific automaton in Figure 5.

Let $action(r, s, A)$ denote the action function, which returns the state obtained by performing action A on State s according to Protocol r if the action succeeds, or *error* otherwise:

$$action(r, s, A) = \begin{cases} s' & s \xrightarrow{A} s' \in r \\ error & otherwise \end{cases}$$

We also generalise the action function *action* to a sequence of actions as

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

B. Main Analysis

Aiming to find out those inappropriate action sequences which are likely to cause resource bugs, we propose a taint-like analysis, which combines typestate checking [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 checking is performed meanwhile, that is, states will be checked and changed depending on the relevant action according to the corresponding automata. In short, we take resource protocols as a guide to perform a taint-like analysis on CFG.

Generally, different resources have different APIs, yet the same actions, namely, *open*, *use*, and *close*. So for simplicity, we focus on three abstract statements. Similar to [11], we also consider the assignments and the function calls, while the others are routine and omitted. Therefore, our analysis considers the following kinds of abstract statements:

p = open r	p = q
use r p	p = q.f
close r p	p.f = q
q = c.m($a_1, \dots, a_n$)	

where n is the number of parameters of method m in class c, and r represents any kind of resource.

Let R denote the set of the possible resources, r range over elements of R, S denote the set of the possible resource states (*i.e.*, $\{i, o, u\}$), s range over elements of S, V denote the set of the variables or fields in a program, and v range over subsets of V.

The analysis takes as input the control-flow graph (CFG) of a program and the automaton specifying the resource usage protocol. Besides states of the automaton, the analysis also tracks which kinds of resources are acquired and which variables and fields are "tainted". Formally, the data fact (or property) that the analysis tracks is a set of tripes (r, s, v), meaning that the resource r 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 unify them. Therefore, we do not need to impose a partial order on states, although it can be done easily. The main algorithm is essentially a classic data flow algorithm, which is shown in Algorithm 1.

This algorithm starts from the entry nodes of CFG with the empty data-fact mapping (Lines $1-3$). For each node t, the algorithm applies its corresponding transfer function, yielding a data fact d to its successors (Line 6). The transfer functions either check whether the resource action conforms to the corresponding resource protocol, or propagate the resource information, which are presented in Section III-C. If the data fact d is different from the original data fact $D(t')$ of a successor t' (Line 8), that is, some data facts are fresh, then the algorithm update $D(t')$ by unioning $D(t')$ and d (Line 9), and enqueue t' into the queue p (Line 11). The algorithm traverses over CFG until p is empty, that is, until the data-fact mapping is no longer updated (Lines $4-11$).

Since the states of the protocol automaton, the variables and fields are finite, (and the length of access paths is limited), the triples are finite as well. Therefore, the state-taint analysis algorithm terminates finally.

In addition to the *statecheck* of data facts during the traversal of CFG (see the transfer functions in the next subsection), 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 *r.accepts* returns the set of accepted states of automaton r. Note that, the global or static variables are considered as local variables of the *main* function.

Algorithm 1: State-Taint Analysis Algorithm

```
   Input: CFG, resource protocol (an automaton)
   Output: The data fact mapping D
1  for each node t ∈ CFG do
2      Let D(t) = ∅
3  enqueue each entry point into queue q
4  while q is nonempty do
5      t ← dequeue p
6      d ← apply the transfer function of t on D(t)
       w.r.t. the resource protocol
7      for each successor t' of t do
8          if d ≠ D(t') then
9              D(t') ← d ∪ D(t')
10             if t' ∉ q then
11                 enqueue t' into q
12 return D
```

Algorithm 2: Exit Checking Algorithm

```
   Input: The data fact mapping D
1  for each exit of each function f do
2      for each (r, s, v) ∈ D(exit) do
3          if s ∉ r.accepts ∧ v ⊆ local(f) then
4              statecheck (r, s, v)
```

C. Transfer Functions

Consider the transfer functions for intra-procedural analysis first, which are shown in Figure 6. For the *open* action p = open r, it first creates a new data fact $(r, o, \{p\})$ and kills the data facts that were managed formerly by p. Then it checks the newly generated data fact: if a resource is managed by no variables and its state is unaccepted, then a resource bug is reported. The *use* and *close* actions change the states of resources managed by p according to the resource protocol automaton respectively. After that, the states are checked: if it is an error state, then 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.

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

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

Function	Output
$remove(v, p)$	$v \setminus \{p.\pi \mid p.\pi \in v\}$, π is any access path
$kill(d, p)$	$\{(r, s, remove(v, p)) \mid (r, s, v) \in d\}$
$lift(d, p)$	$\{(r, s, v) \in d \mid \exists q \in v.\ alias(p, q)\}$*and report error when empty*
$killalias(d, p, f)$	$\{(r, s, remove(v, q.f)) \mid (r, s, v) \in d \wedge alias(p, q)\}$
$statecheck(d)$	$\{(r, s, v) \mid (r, s, v) \in d \wedge s \neq error \wedge v \neq \emptyset\}$ if $s = error \vee (v = \emptyset \wedge s \notin r.accepts)$ then report

Fig. 6. Transfer Functions for Intra-Procedural

r, then f_{call} will also transfer r to the callee's context by replacing c with *this*. The call flow function is:

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

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 shown as follows:

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

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

The alias analysis is triggered by the *use* or *close* actions and the assignments to heap variables. The alias analysis can be any alias analysis. 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.

D. Example

Consider the file example illustrated in Section II again. For convenience, we focus on the resource actions and represent them in our abstract statement presented above. Figure 7 gives the CFG of the file example, where d_i are data facts to be computed.

The *open* action generates $(f, o, \{file\})$ (*i.e.*, d_1), which flows to $d_2 - d_5$. While the *wirte* and *read* actions 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 4, the *close* action closes d_2 and d_3 normally except for d_1, since the state of d_1 is o, indicating that the open

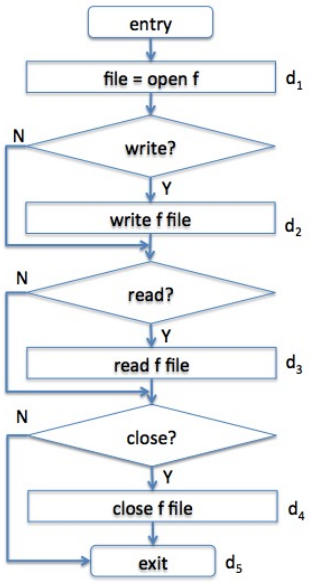

Fig. 7. Control Flow Graph of File Example

file has not been used yet. Finally, all the data facts above will 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 I.

TABLE I
DATA FACTS FOR FILE EXAMPLE

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

Let us check the data fact of the *exit* node. Only the

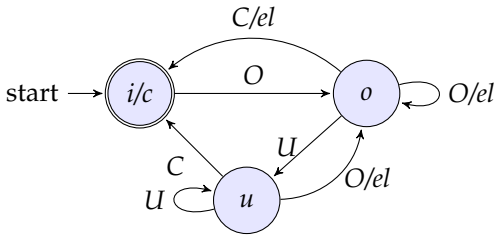

Fig. 8. Abstract FSA with Energy Leak

triple $(f, c, \{file\})$ is accepted. 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.*, *error*). These are the possible resource bugs that illustrated in Section II.

IV. Application: Energy Leak

Over the recent years, the popularity of smartphones has increased dramatically. This has led to a 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 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.

As discussed in Section I, a resource bug can cause energy leak, e.g. if a resource is left open to consume energy unnecessarily, as also illustrated in the Android code for network in Section II, where the HTTP connection is kept open. However, not necessarily all resource bugs lead to energy leaks, for instance, to use a closed resource is a resource bug but may not cause noticeable energy leak. Therefore, we distinguish actions that may cause energy leaks from other inappropriate actions, and enrich our resource usage protocol with such a distinction.

Figure 8 shows an abstract automaton for the improved protocol, where *el* denotes that the corresponding action may cause an energy leak. Generally, if an unaccepted state of a resource reaches the *exit* node, then an energy leak is caused by this resource, since it is not closed eventually. Moreover, a resource should not be opened again, since the former one is not yet used and may remain open to consume energy. An opened resource should be used before closing. Otherwise, to open it is useless and could cause energy waste.

Let us use these improved protocols to guide *state-taint analysis* to analyse the network example illustrated in Section II. Similarly, we focus on the resource actions and represent them in our abstract statement for convenience. Due to limited space, we just consider the data facts for the last action node (*i.e.* the last *download* action) and the *exit* node of CFG, which are shown in Table II, where *hc*, *is* and *dl* represent the HTTP connection, input stream and downloaded data respectively[2].

TABLE II
Partial Data Facts for Network Example

Node	Data Fact
result = open dl	$\{(hc, o, \{huc\}), (hc, u, \{huc\}), (is, u, \{out\}), (dl, o, \{result\})\} \rightarrow el_{(dl,o,\{result\})}$
exit	$\{(hc, o, \{huc\}), (hc, u, \{huc\}), (is, u, \{out\}), (dl, o, \{result\}), (dl, u, \{result\})\}$

First, the analysis will generate an energy leak for the *download* action, since the variable *result* may point to some downloaded but unused data. Next, let us check the data fact of the *exit* node. Only the triple $(dl, u, \{result\})$ is accepted. The data fact indicates that (i) the http connection may be opened but not used or closed in some situation; (ii) the input stream is not closed; (iii) the download data may not be used in some situation. These are the possible energy leaks illustrated in Section II earlier.

V. Implementation and Experiments

We have implemented our analysis as a tool called *statedroid* to detect energy leaks for Android applications, which consists of two parts: the front-end and the back-end. The front-end parses an apk file and then builds a control flow graph, which is passed to the back-end as input. We build our tools based on *flowdroid*, since its front-end almost does the same things 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 III. Our state-taint analysis is implemented upon the IFDS framework [14].

To evaluate our analysis, we have conducted a series of experiments on real Android applications by using *statedroid*. First, we collected 100 real Android applications from F-Droid, a famous free and open source Android application repository, since it is easy to obtain the source codes for open source applications. Then we performed our tool *statedroid* on each application, taking the resource protocols for http connection as a guide. Our experiments are run on a machine with Intel core I5 CPU and 4GB RAM, running Ubuntu 14.04.

Among these 100 applications, our tool reports that 18 applications have energy bugs. Table III summarises our findings[3] of these 100 applications, where *O* denotes

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

[3]We exclude the bugs caused by exceptions, as there are too many such kind of bugs and not all of them are interesting.

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, and *FP* denotes the resource bugs reported by our tool but cannot be rediscovered manually. The results shows that lots of bugs are due to the unclosed-ness (*i.e.*, *O* and *U*). The reason is that programmers are prone to forget to close the resource for every exit. Moreover, there are also many bugs due to the unused-ness (*i.e.*, *O* and *C*). This is 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.

We also performed manual analysis on the source codes of these applications where energy bugs are reported, and found that 13 energy bugs turn out to be false positives. There are several reasons for these 13 false positives. The first one is the null pointer checking, due to which 69.2% (9) 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. Our analysis can not catch this variable. Another reason is that a type checking is performed before closing, for example, *con instanceof HttpURLConnection* is performed before closing *con* in applicatioin *Mirrored*.

In addition, the running time of our tool ranges from 2.4s to 152.8s with an average of 46.4s, and the memory overhead is from 68.6MB to 1186.4MB with an average of 648.5MB, which indicates that our tool is lightweight and easy to use in practice.

TABLE III
RESULTS FOR SELECTED APPLICATIONS

App.	*O*	*U*	*C*	*T.*	*FP*	*LOC*	*Time*	*Mem.*
antennapod	1	4	0	5	0	28529	152.8	1010.5
bible	3	0	2	5	2	27099	122.0	800.5
coolreader	2	1	0	3	0	5326	58.1	307.1
cordova	12	2	2	16	0	38488	61.1	946.0
derbund	14	0	0	14	0	9981	9.4	718.8
gearshift	3	1	1	5	1	17504	131.0	609.4
giga	1	0	0	1	0	11636	9.4	436.8
goblim	1	1	0	2	0	9402	39.5	328.6
impeller	21	1	0	22	0	26426	46.0	651.5
kontalk	6	1	2	9	1	44480	128.6	1121.9
lico	2	0	0	2	0	3382	3.5	501.6
mirakel	2	2	0	4	0	37991	12.6	1091.6
Mirrored	1	2	1	4	4	2395	2.4	550.4
movement	2	0	0	2	1	991	13.8	68.6
muzei	2	0	0	2	2	3960	33.5	218.8
openmensa	0	3	0	3	0	7546	4.8	414.9
remote	0	1	0	1	0	3819	2.5	709.0
tether	1	1	0	2	2	9570	5.0	1186.4
Total	74	20	8	102	13	-	-	-
Aver.	-	-	-	-	-	16029.2	46.4	648.5

VI. 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 [11]. Our price is very low, while their price is 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 analysis 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 [11] 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. [15] 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 [16]–[20], while our analysis concerns about the correct usage of resource (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 [21] for more detail. Here we discuss those that adopt program analysis, and consider the root causes of energy leaks.

Pathak et al. [22] proposed a data flow analysis to check Wakelock API (*i.e.*, *on* and *off*) to find no sleep bugs. Chaorong Guo et al. [23] built a function call graph and then checked whether *require* and *release* actions are matched. These methods consider simper resource protocols, with only *open* and *close* states, than our analysis.

Zhang Lide et al. [8] presented a dynamic taint-tracking to detect energy leaks resulting from unnecessary network communication. Yepang Liu et al. [24] used Java Path Finder not only to monitor sensor and wake lock operations to detect missing deactivation of sensors and wake locks, but also tracked the utilization of sensory data. Compared to our analysis, these two methods considered partial usages of network, sensors and wake locks.

VII. Discussion and 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 used 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 embedded resources whereby a resource may contain or open another resource. We can improve the protocols and/or the analysis to take into account the relations between resources. We can also improve our analysis to deal with the null pointer checking to reduce false positives.

Acknowledgements. The authors would like to thank the anonymous reviewers for their helpful comments. This work was partially supported by the National Natural Science Foundation of China under Grants No. 61502308 and 61373033, Science and Technology Foundation of Shenzhen City under Grant No. JCYJ201418193546117.

References

[1] A. Igarashi and N. Kobayashi, "Resource usage analysis," in *POPL '02*, 2002, pp. 331–342.
[2] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, pp. 157–171, jan 1986.
[3] R. DeLine and M. Fähndrich, "Enforcing high-level protocols in low-level software," in *PLDI '01*, 2001, pp. 59–69.
[4] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," in *PLDI 02*, vol. 37, no. 5, 2002, pp. 1–12.
[5] K. Marriott, P. Stuckey, and M. Sulzmann, "Resource usage verification," in *APLAS 03*, vol. 2895, 2003, pp. 212–229.
[6] N. Kobayashi, "Time regions and effects for resource usage analysis," *Acm Sigplan Notices*, vol. 38, no. 3, pp. 50–61, 2003.
[7] H.-G. Kang, Y. Kim, T. Han, and H. Han, "A path sensitive type system for resource usage verification of c like languages," in *APLAS '05*, 2005, pp. 264–280.
[8] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang, "ADEL: An automatic detector of energy leaks for smartphone applications," in *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, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *PLDI '14*, 2014, pp. 259–269.
[10] *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.
[11] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *ICSE '10*, 2010, pp. 535–544.
[12] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *FASE 13*, vol. 7793, 2013, pp. 210–225.
[13] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "edoctor: Automatically diagnosing abnormal battery drain issues on smartphones," in *NSDI 13*, 2013, pp. 57–70.
[14] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL '95*, 1995, pp. 49–61.
[15] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," in *ISSTA 06*, 2006, pp. 133–144.
[16] W.-N. Chin, H. H. Nguyen, S. Qin, and M. Rinard, "Memory usage verification for oo programs," in *SAS 05*, 2005.
[17] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen, "Verifying safety policies with size properties and alias controls," in *ICSE 05*, 2005.
[18] G. He, S. Qin, C. Luo, and W.-N. Chin, "Memory usage verification using hip/sleek," in *ATVA 09*, 2009, pp. 166–181.
[19] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino, "Types and effects for resource usage analysis," in *FOSSACS 07*, 2007, p. 3247.
[20] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin, "Analysing memory resource bounds for low-level programs," in *ISMM'08*, 2008.
[21] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, profiling, and debugging the energy consumption of mobile devices," *ACM Comput. Surv.*, October 2015.
[22] A. Pathak, A. Jindal, Y. C. Hu, and 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.
[23] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *ASE 2013*, 2013, pp. 389–398.
[24] Y. Liu, C. Xu, S. Cheung, and J. Lu, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *TSE*, vol. 40, no. 9, pp. 911–940, 2014.