Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

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

ARTICLE INFO

Article history:
Received 3 January 2017
Received in revised form 20 June 2017
Accepted 25 June 2017
Available online xxxx

Keywords:
Resource bug
Static analysis
Energy leak
Android

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.

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 [2,3,1,4–7]. 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

* Corresponding author at: College of Computer Science and Software Engineering, Shenzhen University, China.
E-mail addresses: xuzhiwu@szu.edu.cn (Z. Xu), 2150230509@szu.edu.cn (C. Wen), shengchao.qin@gmail.com (S. Qin).

http://dx.doi.org/10.1016/j.scico.2017.06.010
0167-6423/

involves several usage constructors. When files are accessed through the invocation of a function closure, the type annotation 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 [2,3,1,4–7], 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 also 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 generalise the analysis of energy leaks in [10] to be an extension of the proposed state-taint analysis (see Section 5), which we called as resource behaviour analysis, so as to analyse 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.

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

Please cite this article in press as: Z. Xu et al., State-taint analysis for detecting resource bugs, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.06.010