Understanding and Detecting Performance and Security Bugs in IOT OSes

Hongliang Liang, Qian Zhao, Yuying Wang
College of Computer Science
Beijing University of Posts and Telecommunications
Beijing, China

Haifeng Liu
Beijing Information Security Test and Evaluation Center,
Beijing, China

***Abstract*—Operating system (OS) plays an important role in the efficiency and security of service in Internet of Things (IOT). Considering the limited storage resources and power utilization mechanism in IOT OS, we explore the performance bugs and security bugs of three open source IOT OS (Contiki, TinyOS and RIOT OS), including the features of bugs cause, bugs mitigation, bugs detection rules and bugs fixing in them. We present Rulede, a tool built on LLVM compiler framework to find bugs. Experimental results show that Rulede can effectively detect performance bugs and security bugs in target IOT OSes.**

Keywords—software performance bugs; software security bugs; Internet of things; Operating system; static analysis

I. INTRODUCTION

A. Motivation

With the wide adaption of smart terminals and sensors, there are now more applications for Internet of Thing. The early research IOT OS comes from wireless sensor OS represented by TinyOS [1] and Contiki [2]. Recently RIOT OS [3] is a friendly IOT OS because it allows application programming with the programming languages C and C++, and provides full multithreading and real-time abilities. Besides, more and more IT companies are exploiting the IOT market, for example, Google'sAndroid Home [4] and Nest Lads [5], Apple's HomeKit [6], Microsoft's Windows IOT [7], HuaWei's LiteOS [8].

Limited to storage, volume and portability of IOT OS, the quality of its software should be paid close attention. While there are still many kinds of bugs which greatly affect the power of hardware and may cause the security events, Performance bugs and security bugs occupy a big portion among all the software bugs. From May.2003 to Aug.2010, there are totally 4239 performance bugs, 847 security bugs in Firefox [9] bug database. Besides, Until Nov. 2015, 72606 and 67667 security bugs/vulnerabilities are recorded in NVD [11] and CNNVD [12] respectively.

Performance bugs may cause reduced throughput, wasted resources and increased latency. Security bugs/vulnerabilities may be exploited by the attackers, the results would be serious or even a disaster. Because of the specificity of platform and scene, the hidden danger brought by these two bugs is higher in IOT OS. Due to the limitation of hardware speed and resource consumption and the complexity of software, it is more difficult to repair both kinds of bugs than others, more experienced developers are needed and the repair process may also affect other software attributes [10]. However, there is currently little specific research for the performance bug and security bug of IOT OS. This paper aims to research on both kinds of bugs in IOT OSes.

B. Contribution 1: Study on the Characteristics of Bugs In IOT OSes

This paper samples 23 performance bugs and security bugs from 3 representative IOT OSes (Contiki, TinyOS and RIOT OS) by searching their bug database with related keywords(such as memory, buffer, performance, security, throughput, time, etc.). Combining with the features of each IOT OS, we explore bug classification, bug cause, bug mitigation in them. Our study makes the following findings:

- Understanding of performance bugs and security bugs in IOT OS: since limited resource and low power are two important features in IOT OS, performance problems should be paid more attention. Security in IOT OS is fundamental and may affect the security of the whole IOT system. IOT as a promising industry which are getting wide applications in market, it is necessary to take precautions to ensure efficiency and security of IOT.
- Guidance for bug detection: our study can be regarded as a contribution to accuracy and coverage of bug detection tool. We summarized some reusable efficiency rules in the bugs we study. These rules are helpful to detect bugs in IOT OS.
- Guidance for bug fixing: By investigating the patches of the 23 bugs, we find that these patches are useful to fix bugs with the similar structure patterns.

C. Contribution 2: A Tool to Detect Bugs In IOT OSes

Based on the study of 23 bugs from three target IOT OSes (Contiki, TinyOS, and RIOT OS), we get 4 general rules, including 2 performance bug detection rules and 2 security bug detection rules. What's more, all of the 4 rules are easy to understand and all are described with clear violation condition. We built a rule-based bug detection tool (named Rulede)

This work is partially supported by the National Science Foundation of China (NSFC) under grant No. 91418206, National High-tech R&D Program of China (863 Program) under grant No. 2013AA014701.

978-1-5090-2239-7/16/$31.00 copyright 2016 IEEE
SNPD 2016, May 30-June 1, 2016, Shanghai, China

which can detect the bugs that violate those four rules in target IOT OS. For the sake of expandability, design and implementation of Rulede is based on LLVM [13] compiler framework.

Rulede has detected 45 potential bugs in the target OSes, including 42 performance bugs and 3 security bugs. 2 out of the 45 are already known (they were reported in the past), and others are undetermined. Besides, one of three security bugs was verified by the official developers, and most of the undetermined performance bugs have attracted attention of developers.

II. STUDY ON THE CHARACTERISTICS OF BUGS IN IOT OSES

Contiki [14], TinyOS [15] and RIOT OS [16] are widely used open source IOT OS. They all have long development cycle, mature code management mechanism and bug tracking system. For above reasons, we choose them as our research target OSes.

A. Classifying Bugs In IOT OSes

By searching bug database of Contiki, TinyOS and RIOT OS using a set of performance and security related keywords, such as memory, buffer, performance, security, throughput, time, etc. We samples 23 fixed bugs, including 11 from Contiki, 6 from TinyOS and 6 from RIOT OS.

TABLE I. CLASSIFICATION OF BUGS IN TARGET OSES

OS	Sec. bugs of abusing storage	Perf. bugs of abusing storage	Perf. bugs of abusing CPU time	Sum.
Contiki	4(bugID:710, 759,796,1090)	5(bugID:217,81 2,150,857,890)	2(bugID:414,64 8)	11
TinyOS	0	3(bugID:120,26 6,288)	3(bugID:252,26 3,296)	6
RIOT OS	2(bugID:1782 ,3316)	2(bugID:288,31 99)	2(bugID:3431,3 439)	6
Sum.	6	10	7	23

Based on the study of storage resources and power utilization mechanism in target OSes. We classify the collected 23 bugs into 3 classifications:

- *Security bugs of abusing storage*: the security bugs caused by coding errors lead to memory overflow [17] or memory over-read [18], etc.
- *Performance bugs of abusing storage*: the performance bugs caused by coding errors lead to storage resource waste.
- *Performance bugs of abusing CPU time*: the performance bugs caused by coding errors lead to CPU time waste.

The number and bugID of every classification of 23 bugs we collected are shown in table 1.

B. Analyzing Bugs Causes

There are many reasons that cause performance bugs and security bugs in IOT OS. According to the 23 bugs in target OSes we study, the bug causes are divided into four categories:

- *API misuse*: Contiki supports C program language and uses C standard library API. In terms of memory or string related APIs (such as memcpy, sprintf and strcat), misusing of these APIs may cause memory overflow, memory over-read and memory leak and so on. TinyOS doesn't support C, but it uses a fragment pool buffer to manage memory and provides specified APIs for allocation and free of the pool. Memory leak will occur when programmers forget to free the pool. RIOT supports both C and C++, so above dangerous APIs should be carefully used in RIOT OS.
- *Lacking memory management*: Contiki supports dynamic memory allocation, so the heap space allocated dynamically may conflict with the stack space, then a memory overflow may occur in this case. TinyOS conducts static memory allocation in compiling time, which results in little memory bugs. RIOT OS allows multithreading and dynamic memory allocation thoroughly. So its developers should pay more attention on memory management issue. We find they should not ignore some details to save storage space, for example, the default size of static allocation should be set to a minimum suitably.
- *Code redundancy*: there are code redundancy problems both in core codes and application codes of Contiki and TinyOS, which leads to serious storage abusing in IOT OS, then performance loss thereby arises.
- *Logic defects*: there are useless and error operations in three IOT OSes, for example, two variables with inconsistent types or bytes are dealt with assignments and comparison operations. These bugs may lead to CPU time wasted. Ultimately, it is necessary for developers to have correct and meticulous programming in IOT OS.

C. Advice to Mitigate Bug

After the research on the 23 bugs in three IOT OSes, we provide mitigation advices for developers according to the bug cause, as shown in table 2. We hope these advices can help developers understand and mitigate performance and security bugs in IOT OS to some extends.

III. DESIGN AND IMPLEMENTATION OF RULEDE

A. Workflow And Architecture

IOT OSes are developed with different program languages and they run on various target platforms. In order to detect bugs in them in a simple manner, we decide to build our analysis tool (Rulede) based on LLVM which uses

TABLE II. MITIGATION ADVICES OF CONTIKI, TINYOS AND RIOT OS BUGS

Bug causes	Contiki	TinyOS	RTOS	Mitigation advices
API misuse	4	0	0	Pay attention to memory and string operation API, especially length check of written data, such as memcpy, strcat , sprintf and so on.
Lacking memory management	2	1	3	1) Enhance management of stack space and control of buffer flow in Contiki. 2) The default size of static allocation should be set more reasonable in RIOT OS. 3) Memory pool allocated should be freed after related operations in TinyOS.
Code redundancy	2	2	0	The repetitive codes should be written as a function.
Logic defects	3	3	3	Enhance function testing or verify the codes formally.

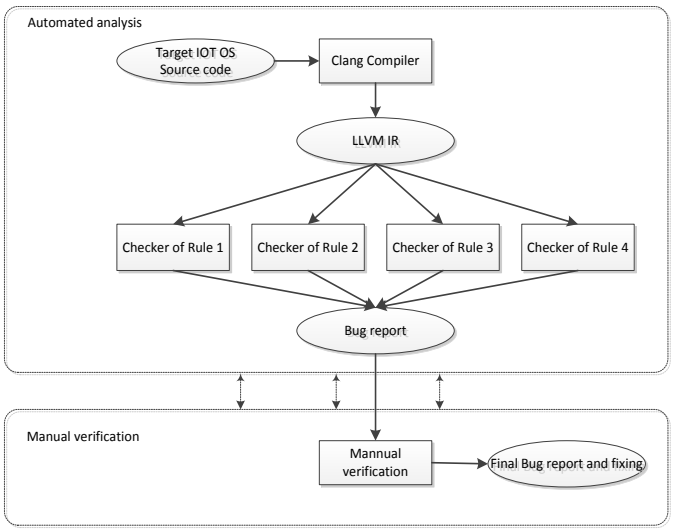

Fig. 1. Workflow and architecture of Rulede. Dotted rectangles represent work phases. Ellipses represent data, and rectangles represent phases of workflow. Full lines represent data flows. Dotted lines represent workflows in different phases

Intermediate Representation (IR) that is both front-ends and back-ends independent.

The workflow and architecture of Rulede is shown as figure 1. Firstly, the source files of target OSes (Contiki, TinyOS and RIOT OS) are transformed by clang into LLVM IR files. Secondly, these IR files are checked by rule-based static checker, then a preliminary bug report is generated. Finally, the bug report will be verified manually. With the study of source codes and documents of target OSes, then final bug report and bug fixing schemes are obtained. In other words, Rulede has two work phases, one is automated analysis phase, and the other is manual verification phase. The manual verification phase is important because it can help improve accuracy and coverage of the former phase.

Automated analysis phase consists of language process module and rule-based bug detection module. Both modules and manual verification phase are described in next sections B, C, D.

B. *Language Process Module*

This module transforms source codes into LLVM IR by using Clang [19] which is a compiler front-ends in LLVM. Though source files of Contiki and TinyOS can be compiled by Clang directly, those of TinyOS written by nesC [20] need an extra transformation, then can be compiled by Clang. This module transform the nesC codes of TinyOS into C codes firstly, then the rest process is just same as Contiki and RIOT OS.

In general, if we want to use Clang to compile applications built by GCC, we just need to modify its build files by changing GCC to Clang and adding some compiler options. However, IOT applications are just a small part of their source codes. We can't transform all the source codes of IOT OS using the method mentioned above.

In order to perform a complete transformation of source codes, each of the three IOT OSs source codes is divided into two parts to be processed. One part is the codes of kinds of applications, which are treated as follows:

- Replace the specified compiler with Clang in project's "Makefile" file;
- Add "-emit-llvm" and "-g" options to compiler, and deal with the conflict between new options and old options;
- Write shell scripts to transform all applications in the three IOT OSes, and collect the generated LLVM IR.

Another part is the codes of system components, which are treated by a shell script whose main task is to add suitable compile options to Clang. Though these compile options are slightly different in every IOT OS, specify locations of CPU information files and platform files should generally be given. Also we define some macro used in the compiled files. Finally, we collect the LLVM IR files produced in above steps.

C. *Rule-Based Bug Detection Module*

There are four static checkers in this module according to four bug detection rules. These checkers are implemented on LLVM compiler framework, by using its intra-procedural control-flow analysis and data-flow analysis API. Detection and fixing advice of each bug classification are detailed in next sections a, b, c, d.

a. Detection and Fixing Advice: Classification 1

Storage resource is one of the most important resource in IOT OS, However, our study shows that code redundancy is a serious issue, 2 out of 11 bugs in Contiki and 2 out of 8 bugs in TinyOS are caused by this kind of bug. For example, there are code redundancy [21] in "webbrower" and "webserver" applications in contiki. Besides, the codes to determine whether a queue is full are used at several locations [22] In TinyOS. CPU time is another important resource in IOT OS. Based on the study of code redundancy bugs, together with repeating and inefficient loop structure in target IOT OS, we find that it is inefficient when processing element of a string in two situations. 1) The step in a loop structure is so small that it may decrease the run speed of program when the string is very long. 2) The same operation to the same string may occur more than once in the file(s). So we conclude the first rule of classification:

Bug matching condition: In a loop structure, the loop condition is one equal/unequal predicate or AND operation of two equal/unequal predicates, as well as one of the predicates is a string element comparing with a constant.

Bug fixing scheme: 1) For saving storage: the loop structure used several times should be written as a function; 2) For saving CPU time: according to the context of loop structure, the loop step should be set to a greater value. For example, if the loop step in the loop that matches this condition is 8 bits, and the string to be handled is very long, the value of step can be set to 16 bits, 32 bits or 64 bits.

b. Detection and Fixing Advice: Classification 2

Memory over-read [17] is a special case of violation of memory safety that overrun buffer's boundary and read (or try to read) adjacent memory. This issue may happen when using memory or string operation APIs, for example, it may cause a memory over-read when a "memcpy" API is used carelessly in Contiki-bug710 [23]. In the case of this bug, the third argument of "memcpy" is a constant, the second argument is a pointer type. So memory over-read may occur if there isn't suitable length check before this function call. The better choice for developers is to replace "memcpy" with "strncopy" to avoid the problem. Then we concluded the second rule of classification:

Bug matching condition: when calling the "memcpy" API, its third argument's value is a constant, and the type of the second argument is a pointer.

Bug fixing scheme: replace the "memcpy" API with the "strncopy" API. For example, "memcpy(dest,src,length)" should be modified to "strncopy(dest,src,length)".

c. Detection and Fixing Advice: Classification 3

The "sprintf" API is used to write formatted data into a destination string buffer. One of its application scenarios is that a fixed number of data is converted to another format and written to a destination. We can replace the "sprintf" statement to a special arithmetic in this case so as to make the program run faster. Then we concluded the third rule of classification:

Bug matching condition: when calling sprint API, the second argument is a fixed constant string (for example "%02x%02x%02x%02x") and the converted numbers of data are greater than 4 in general.

Bug fixing scheme: Replace the "sprintf" API with another special arithmetic according to application scenarios.

For example, in the below code segment, the "sprintf" API is used to convert 16 elements of a string into "%02x" format. "src[]" is a character array to be converted.

```
sprintf(dest,"%02x%02x%02x%02x%02x%02x%02x%02x%02x
%02x%02x%02x%02x%02x%02x%02x",src[0],src[1],src[2]
,src[3],src[4],src[5],src[6],src[7],src[8],src[9],
src[10],src[11],src[12],src[13],src[14],src[15]);
```

The above code segment can be replaced with the below codes to save CPU time. The execution speed will be improved significantly.

```
static char hexmap[]={'0','1','2','3','4',
'5','6','7','8','9','a','b','c','d','e','f'};
int i;
char *p=dest;
for(i=0;i<=15;i++){
  (*p++)=hexmap[src[i]>>4];
  (*p++)=hexmap[src[i]&15];
}
```

d. Detection and Fixing Advice: Classification 4

The third parameter of "strncat" API is designed to prevent buffer overflow [18]. However, it still exists the case that its third parameter is assigned an inappropriate value. For example, programmers sometimes ignore that there is a null-terminator in C-style string in Contiki [24], right value of the third parameter should be "sizeof(dest)-strlen(dest)-1" when the "dest"(the first parameter of "strncat") is a C-style string. Then we concluded the fourth rule of classification:

Bug matching condition: when calling "strncat", the value of the third parameter is "sizeof(dest)-strlen(dest)", and "dest" is the first parameter of "strncat".

Bug fixing scheme: replace the third parameter with "sizeof(dest)-strlen(dest)-1".

D. Manual Verification

In this phase, the main work is to conduct manual review and verification on the bugs report produced by rule-based bug detection module. Not only source codes and documents of target OSes are needed to verify those reported bugs. In final, we acquire the final bug report and determine the fixing scheme by the types of bugs. Furthermore, the power and precision of rule-based bug detection module can be improved by analyzing false positives found in this phase.

IV. EVALUATION

We choose different versions of Contiki, RIOT OS and TinyOS to evaluate the capability of Rulede. The versions and SLOC of the target OSes are shown in table 3. 45 potential bugs are found in these OS, including 42 performance bugs and 3 security bugs. 2 of the 3 security bugs are known bugs, and the left one is unknown which has been verified by its

developers. Most of undetermined performance bugs has attracted attention of developers, but not verified as quickly as security bugs. An important reason is that performance fixes are more likely to introduce new functional bugs or increase code complexity and maintenance costs. So it is necessary to take more time for developers to review and fix performance bugs. However, aiming at saving storage resources and power energy of IOT OS, the fixing suggestions to these performance bugs will bring at least a few bits of space or millisecond of time saved. Moreover, most of bugs we detected can be repaired by lines of code modification as shown in section III.C.

TABLE III. IOT OSES ANALYZED BY RULEDE

Target OS	Version	SLOC
Contiki	V2.7	418382
	V3.0	460472
RIOT OS	V.2014.12	269396
	V.2015.09	449189
TinyOS	V2.1.2	926719

Comparison with other tools. The four checkers of Rulede can detect not only performance bugs caused by code redundancy or code inefficiency, but also security bugs caused by buffer overflow or buffer over-read. To demonstrate the detection ability and time performance of Rulede, we compare it with cppcheck [25] and ccfinder [26], both of which are static bug detection tools. Ccfinder is dedicated to detect code clone to increase maintainability of software, we use it to detect code redundancy for comparison. Cppcheck can detect many kinds of errors and bugs in source codes, such as compatibility, security and performance problems. We conduct our experiment on a 4 core CPU and 16GB memory computer which runs Ubuntu 14.04 operating system.

We use cppcheck and ccfinder to analyze three target Oses (Contiki, TinyOS and RIOT OS) and find that neither of them can detect any performance bugs found by rulede. cppcheck can detect a security bug found by Rulede.

Furthermore, we run three tools to analyze contiki2.7 which contains 1255 C files for five times and record the detection duration respectively. The average detection durations of Rulede, cppcheck and ccfinder are 10.36s, 18.656s and 35.58s.

Rulede is implemented as a LLVM pass which works on LLVM IR. Taking advantage of LLVM, it achieves good analysis ability and speed. Moreover, we hope the detection rules and fixing suggestions will be guidance to future IOT OS development and be reused in other detection tools.

V. RELATED WORKS

There are several studies on the performance analysis and improvement of IOT OS protocol stack [27-30] and on security in IOT OS protocol stack [31,32]. The performance and security bugs in our work can't be avoided through well-designed IOT OS protocols. These research are complementary with our work. In this paper we focus on those bugs resulted from bad programming practice, such as API misuse, memory recycling improperly, code redundancy, and so on.

Researchers have paid more attention to performance bugs detection in recent years. A study [33] on performance bugs of Apache, MySQL and Mozilla, shows that performance bugs exist widely in released software and are difficult to detect. The study also shows that bug patches include reusable efficiency rules that can help detect and fix performance bugs. Suncat [34] can help developers understand and predict performance bugs in Windows Phone applications. It uses a small input to predict performance problems that would occur on a large-scale input. CRAMEL [35] has a novel perspective on performance bugs detecting and fixing, and comes up with a new family of performance bugs that have non-intrusive fixes likely to be adopted by developers. In addition, CRAMEL provides a potential source-level fix for bugs detected.

Detection of security bugs is more mature than that of performance bugs, two of the most valuable static analysis tools are Fortify SCA [36] and Coverity Prevent [37]. Fortify analyzes all the paths in source codes based on collections of security bug rules, a bug will be caught once a path matches a rule. And the bug report contains bug information, related security knowledge and fixing advice, and so on. Satisfiability analysis technology is used in Coverity Prevent to improve power of bug detection, and acquires great achievements. Now it can detect not only C programs, but also C++, Java and PHP programs.

VI. CONCLUSION

The key of IOT services quality is to ensure efficiency, security and privacy of the data and applications. Considering of the requirements of limited resources and low power in IOT OS, we have a comprehensive study on 23 performance bugs and security bugs of three representative IOT OSes (Contiki, TinyOS and RIOT OS). With our study on these bugs characteristics, we hope that developers would understand more about performance bugs and security bugs in IOT OS, and our study results will be guidance of bugs detection and fixing. Moreover, a rule-based bug detection tool—Rulede is presented and is evaluated which can find some performance bugs and security bugs in three IOT OSes. Moreover, we hope the detection rules and fixing suggestions will be guidance to future IOT OS development and be reused in other detection tools.

REFERENCES

[1] *TinyOS*. Available from: http://www.tinyos.net/

[2] *Contiki*. Available from: http://www.contiki-os.org/.

[3] *RIOT*. Available from: http://www.riot-os.org/

[4] *Editorial: Android@Home*; Available from: http://www.engadget.com/2011/05/11/editorial-android-home-is-the-best-worst-thing-that-could-happe/.

[5] *Nest Labs*. Available from: https://nest.com/.

[6] *HomeKit*. Available from:https://developer.apple.com/homekit/

[7] *WindowsIOT*.Available from:https://dev.windows.com/en-us/iot

[8] Huawei's Lite OS; Available from: http://betanews.com/2015/05/20/huaweis-liteos-internet-of-things/

[9] *Mozilla Firefox*. Available from: https://www.mozilla.org/en-US/firefox/new/.

[10] Zaman S, Adams B, Hassan A E. Security versus performance bugs: a case study on firefox[C]//Proceedings of the 8th working conference on mining software repositories. ACM, 2011: 93-102.

[11] National Vulnerability Database(NVD)[Z/OL]. http://nvd.nist.gov/

[12] China National Vulnerability Database of Information Security(CNNVD)[Z/OL]. http://www.cnnvd.org.cn/

[13] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004: 75-86.

[14] Dunkels A, Grönvall B, Voigt T. Contiki-a lightweight and flexible operating system for tiny networked sensors[C]//Local Computer Networks, 2004. 29th Annual IEEE International Conference on. IEEE, 2004: 455-462.

[15] Levis P, Madden S, Polastre J, et al. Tinyos: An operating system for sensor networks[M]//Ambient intelligence. Springer Berlin Heidelberg, 2005: 115-148.

[16] Baccelli E, Hahm O, Günes M, et al. RIOT OS: Towards an OS for the Internet of Things[C]//Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on. IEEE, 2013: 79-80.

[17] *buffer over-read*. Available from: https://en.wikipedia.org/wiki/Buffer_over-read.

[18] *buffer overflow*. Available from: https://en.wikipedia.org/wiki/Buffer_overflow.

[19] *Clang compiler*. Available from: http://clang.llvm.org/.

[20] Brewer E, C.D., Gay D, et al. *nesC: A programming language for deeply networked systems*. 2004; Available from: http://nescc. sourceforge. net.

[21] *Limit content of web browser version of http-strings to web browser*. 2013; Available from: https://github.com/contiki-os/contiki/pull/150.

[22] *add ,full() to Queue*. 2014; Available from: https://github.com/tinyos/tinyos-main/pull/266.

[23] *Coffee: Copy the correct number of bytes in filenames #710*. 2014; Available from: https://github.com/contiki-os/contiki/pull/710.

[24] *core/net/ip: Prevent (tiny) buffer overflow in resolv_found()*. 2014; Available from: https://github.com/contiki-os/contiki/pull/796.

[25] Rothstein M. Cppcheck design[J]. Secure Programming: cource for the Spring 2012.

[26] Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code[J]. Software Engineering, IEEE Transactions on, 2002, 28(7): 654-670.

[27] Zhang T, Li X. Evaluating and analyzing the performance of RPL in contiki[C]//Proceedings of the first international workshop on Mobile sensing, computing and communication. ACM, 2014: 19-24.

[28] Ko J G, Dawson-Haggerty S, Gnawali O, et al. Evaluating the Performance of RPL and 6LoWPAN in TinyOS[C]//Workshop on Extending the Internet to Low Power and Lossy Networks (IP+ SN). 2011.

[29] Reusing T. Comparison of operating systems TinyOS and Contiki[J]. Sens. Nodes-Oper. Netw. Appl.(SN), 2012, 7.

[30] Roussel K, Song Y Q, Zendra O. RIOT OS Paves the Way for Implementation of High-Performance MAC Protocols[J]. arXiv preprint arXiv:1504.03875, 2015.

[31] Karlof C, Sastry N, Wagner D. TinySec: a link layer security architecture for wireless sensor networks[C]//Proceedings of the 2nd international conference on Embedded networked sensor systems. ACM, 2004: 162-175.

[32] Casado L, Tsigas P. Contikisec: A secure network layer for wireless sensor networks under the contiki operating system[M]//Identity and Privacy in the Internet Age. Springer Berlin Heidelberg, 2009: 133-147.

[33] Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs[J]. ACM SIGPLAN Notices, 2012, 47(6): 77-88.

[34] Nistor A, Ravindranath L. SunCat: Helping developers understand and predict performance problems in smartphone applications[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 2014: 282-292.

[35] Nistor A, Chang P C, Radoi C, et al. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes[C]. ICSE, 2015.

[36] *HP Fortify Static Code Analysis(SCA)*. Available from: https://www.fortify.com/products/hpfssc/source-code-analyzer.html

[37] *Coverity: Software Testing and Static Analysis Tools*. Available from: http://www.coverity.com/