A consistency-guaranteed approach for Internet of Things software refactoring

Yang Zhang¹, Shixin Sun¹, Dongwen Zhang¹, Jing Qiu² and Zhihong Tian²

Abstract
The software architecture of Internet of Things defines the component model and interconnection topology of Internet of Things systems. Refactoring is a systematic practice of improving a software structure without altering its external behaviors. When the Internet of Things software is refactored, it is necessary to detect the correctness of Internet of Things software to ensure its security. To this end, this article proposes a novel refactoring correction detection approach to ensure software security. Control flow analysis and data flow analysis are used to detect code changes before and after refactoring, and synchronization dependency analysis is used to detect changes in synchronization dependency. Three detection algorithms are designed to detect refactoring correctness. Four real-world benchmark applications are used to evaluate our approach. The experimental results show that our proposed approach can ensure correctness of Internet of Things software refactoring.

Keywords
IoT software, consistency detection, control flow analysis, data flow analysis, synchronization dependency analysis

Date received: 19 November 2019; accepted: 20 December 2019
Handling Editor: Xiaojiang Du

Introduction
In recent years, the wide adoption of the Internet of Things (IoT) systems and immature IoT technologies pose multiple challenges to the development of IoT software.¹,² Despite the multitude of IoT software architectures proposed in previous studies, the optimal IoT software architecture has not been found on a global scale, which means that the IoT technology still needs to be optimized.³ IoT products have provided much convenience to people’s lives. Juniper Research predicts that nearly 38 billion devices will be connected to the Internet by 2020.⁴

With the increase in IoT applications, the type and quantity of IoT terminal devices increase as well. Therefore, the intelligence and correctness of IoT terminals draw wider attention than before.⁵,⁶ However, because the functions and structures of the IoT terminals are different, some terminal devices will not be able to meet the needs of users.⁷,⁸

Some developers refactor the architecture of IoT software to improve reusability and maintainability. However, the existing refactoring methods may incur a variety of concurrency bugs and lead to changes in behaviors. These problems can also cause the security of IoT software to be compromised.⁹,¹⁰ In order to avoid the problem of post-refactoring behavior inconsistency, it is necessary to study the consistency detection approaches.
We propose a novel detection approach to detect software security. This approach uses the control flow analysis, synchronization dependency analysis, and data flow analysis to detect the security of the refactoring under the WALA software analysis framework, and detection algorithms are designed for three kinds of problems that are common in software development to ensure the security of IoT software. In the experiment, we refactor the benchmark programs using the Eclipse refactoring tool and use the proposed detection approach to assess the refactored program. The experimental results show that the proposed approach can effectively resolve the security problems.

**Related work**

This section reviews previous studies on IoT software-based methods and refactoring consistency-based methods.

**IoT software-based method**

IoT security involves several abstraction layers and a number of dimensions. Most security attacks happen at the software level because these attacks are currently the most popular and can affect a large number of devices and processes simultaneously. Most attacks are semantic attacks in data processing. Rebuilding the IoT software is very likely to trigger security threats to the IoT. Therefore, IoT security detection is of great importance.

Xu et al. proposed a trajectory privacy-protection scheme based on a trusted anonymous server. Zhang et al. introduced some background knowledge of information security and ongoing challenges to IoT security. Conti et al. introduced existing major security and forensics challenges in the IoT domain and briefly analyzed some papers targeting identified challenges. Xu proposed a method to address the security issues and key technologies in IoT. He elaborated the basic concepts and the principle of the IoT and combined the relevant characteristics of the IoT as well as the international main research results to analyze the security issues and key technologies of the IoT.

IoT has become a popular term around the globe. Although IoT systems have brought convenience to users, they also cause huge security risks. The risks of IoT software are immeasurable. Problems that occur in IoT software refactoring may lead to changes in user requirements or security vulnerabilities in the software. Therefore, it is important to detect the refactoring of IoT software. Security problems related to IoT systems are drawing more and more attention from security experts and government departments. Both the business community and relevant governmental departments have put forward necessary security assessment requirements for information systems and IoT systems.

**Refactoring consistency-based method**

Many previous studies focused on the consistency of software refactoring. Changes in the behavior may cause security problems in the software. Therefore, some researchers proposed refactoring tools and methods. If the time spent using the refactoring tools and fixing the bugs is less than the time doing it manually, the tool is useful.

Schafer et al. illustrated several types of behavior changes that may cause inconsistency by current refactoring engines and proposed techniques to make the concurrent programs behavior-preserving. They introduced synchronization dependencies that modeled the ordering constraints imposed by the Java memory model and proved that their techniques yielded a strong behavior-preservation guarantee.

Maruyama et al. presented an approach that tames behavior preservation by introducing the concept of a frame. In order to accommodate individual problems in refactoring, a frame was used to represent the boundary of a stakeholder’s concern about the refactored codes. This frame-based refactoring approach preserved the observable behavior within a particular frame and helped programmers distinguish the behavioral changes.

Zhang et al. presented an automated refactoring method among locks at the bytecode level. With the promising features of StampedLock, Zhang et al. presented an automated refactoring framework to convert a synchronized lock to a StampedLock. Although many methods are proposed to address software refactoring issues, there is still no static analysis method to validate the synchronization dependency of synchronized methods and blocks and to detect the consistency of the refactoring behavior. To this end, we use static analysis methods to create an automated detection tool that can detect the security problems of IoT software.

**Motivation**

Refactoring is an effective way to improve software efficiency. In this section, we use an example to illustrate the problem of software security, as shown in Figure 1. In Figure 1(a), method v1() first acquires the monitor object B.class and then calls A.m(), which in turn acquires A.class lock. Similarly, method v2() first acquires the monitor object A.class and then calls A.n(), which acquires the monitor object lock A.class.

In Figure 1(b), we apply the Move method refactoring to move method n() from class A to class B. Moving the synchronized method A.n() to class B leads the method to acquire the monitor object B.class.
Method v2() first attempts to acquire A.class and then B.class. Method v1() acquires the monitor object B.class and then A.class. Hence, refactoring may end in a deadlock.

To address concurrency problems in IoT software, we designed three detection algorithms based on static analysis: deadlock detection algorithm, object reusable detection algorithm, and shared static field detection algorithm.

**Approach overview**

In this section, we introduce our approach to detect code changes before and after refactoring. The framework of our approach is shown in Figure 2:

1. *The input of the refactoring program.* We refactor the open-source programs using the Eclipse refactoring tools. The refactored code is...
obtained by refactoring a particular method, function, or variable.

2. Static analysis. First, we analyze and compare the control flows before and after refactoring to find the structure that leads to inconsistent behavior. Then, we conduct synchronization dependency analysis to detect synchronized methods or blocks and detect the structures before and after refactoring in which the synchronization dependency changes. Finally, we analyze and compare the data flows before and after refactoring to find the structure that leads to inconsistent behaviors.

3. Detection algorithm. We design three algorithms to detect inconsistent behaviors and software security, including a deadlock detection algorithm, an object reusable detection algorithm, and a static shared field detection algorithm.

4. Generating detection results.

Static analysis

Control flow analysis

Control flow analysis generates a directed control flow graph. Node D represents the basic code block, and D = \{d₁, d₂, ..., dₙ\}, where d₁, d₂, ..., dₙ represents the node. Each node has a set of successor nodes which can be empty, and (dᵣ, dₛ) represents the directed edge between nodes.

By comparing the changes of nodes before and after refactoring, we find that the software structure changes because of the refactoring. We define that cfref(dₙ): <exClass, dₙ, dₙ₊₁> is the control flow before refactoring; cfref'(dₙ'): <exClass', dₙ', dₙ₊₁'> is the post-refactoring control flow, where exClass is the class name, dₙ is the nth node, and dₙ₊₁ is a successor node of the nth node.

We assume a refactoring node changes when the node meets the following conditions:

1. exClass = exClass'
2. dₙ = dₙ'
3. dₙ₊₁ ≠ dₙ₊₁'
4. cfref(dₙ) = \{dₙ | cfref(dₙ) ∩ cfref'(dₙ') = φ\}
5. cfref(dₙ) ≠ φ

Comparing cfref(dₙ) and cfref'(dₙ) results in an inconsistent structure. When we detect the program in the same class (condition 1), we then execute condition 2. When dₙ and dₙ’ are the same (condition 2), the node information of dₙ₊₁ before and after refactoring is compared. If the node information of dₙ₊₁ has changed, that is, dₙ₊₁ ≠ dₙ₊₁' (condition 3), it is considered that the node dₙ₊₁ in the control flow has changed.

cfref(dₙ) is the intersection of cfref(dₙ) and cfref'(dₙ') before and after refactoring. If the intersection is empty, it indicates that the control flow information has changed in dₙ₊₁. We store the node dₙ₊₁ in cfref(dₙ) (condition 4). If cfref(dₙ) is not empty (condition 5), it indicates that the nodes have changed before and after refactoring.

For example, we conduct control flow analysis for Figure 1. The code in line 14 before and after refactoring is the same, but in line 15, A.n() ≠ B.n(), that is, the node corresponding to the 15th row is changed.

Synchronization dependency analysis

Synchronization dependency analysis is to analyze the methods that contain synchronized blocks or methods. Synchronization dependencies occur in the following situations:

1. There is a nested relationship between synchronized blocks;
2. There is a calling relationship between the synchronized methods;
3. Synchronized methods contain synchronized blocks;
4. Synchronized methods are called in the synchronized blocks.

A monitor-enter is an instruction in the synchronized block that acquires a lock, and a monitor-exit is an instruction in the synchronized block that releases a lock. If the lock of the monitor is the current class object, it is a static synchronized method. If the lock of the monitor is an instance object of the current class, it is a synchronized method.

The synchronization dependence edge is defined as follows: Synchronization dependence edge analysis is based on the control flow graph analysis. All nodes include an entry node and an exit node of the monitor on the control flow graph:

1. A control flow graph node, Node b, has an acquire dependence on Node a if Node a corresponds to an acquire action and there is a path from a to b in the control flow graph. In this case, we consider there is an acquire edge between a and b, denoted as a.
2. A control flow graph node, Node a, has a release dependence on Node b if Node b corresponds to a release action and there is a path from a to b in the control flow graph. In this case, we consider there is a release edge between a and b, denoted as b.

Synchronization dependency is also defined as follows: A situation is considered to have synchronization
dependency if the following four conditions are met between the synchronized methods and synchronized blocks. Method g() represents that this method contains synchronized blocks, and method f() represents that this method contains synchronized methods:

1. If g(m1) happens before g(m2), g(m2) synchronization depends on g(m1);
2. If f(m1) happens before f(m2), f(m2) synchronization depends on f(m1);
3. If g(m2) happens before f(m1), f(m1) synchronization depends on g(m2);
4. If g(m1) happens before f(m2), f(m2) synchronization depends on g(m1).

Table 1 describes the synchronization dependency relationships of Figure 1. In Figure 1, we first access the synchronized block in the method v1() and then access the synchronized method m() in the static class A. Hence, synchronized method m() has a synchronization dependency relationship with the synchronized block, that is, synchronization of the method m() is dependent on the synchronized block in method v1(). Similarly, synchronization of the synchronized method n() is dependent on the synchronized block in method v2(). However, after refactoring, synchronization of the synchronized method n() is dependent on the synchronized block in method v2(). Since the synchronization dependency relationship has changed, the behavior has changed.

### Data flow analysis

Data flow analysis is based on control flow analysis. It analyzes the flow direction of data on the execution path of a program. The purpose of data flow analysis is to detect changes in the data flow. The set of nodes \( D = \{d_1, ..., d_k \} \), where \( d_i \) represents the \( i \)th node. The entry node is the start of a data flow graph and the exit node is the end. The input and output of Node \( d \) are recorded as \( \text{In}[d] \) and \( \text{Out}[d] \), respectively.

We define \( \text{dfref}(d_i) = <\text{In}[d], \text{Out}[d]> \) as the data flow before refactoring and \( \text{dfref}(d_i) = <\text{In}[d], \text{Out}[d]> \) as the data flow after refactoring. We consider that the refactoring nodes will change when the following conditions are satisfied:

1. \( \text{In}[d_i] = \text{In}'[d] \)
2. \( \text{Out}[d_i] \neq \text{Out}'[d] \)
3. \( \text{dfref}(d_i) = \{ d_i | \text{dfref}(d_i) \cap \text{dfref}^*(d_i) = \emptyset \} \)
4. \( \text{dfref}(d_i) \neq \emptyset \)

When a node \( d_i \) remains the same before and after refactoring (condition 1), the \( i \)th node is performed. If the output data flow of \( d_i \) is different (condition 2), the node \( d_i \) of data flow is identified as having changed. \( \text{dfref}(d_i) \) represents the intersection of changes of each node of the data flow before and after refactoring (condition 3). If the node intersection is empty, this node has changed, and the node information \( d_i \) is stored to \( \text{dfref}(d_i) \). If the final \( \text{dfref}(d_i) \) is not empty (condition 4), it means that there are nodes that have changed before and after refactoring.

### The algorithm

In this section, using three examples, we design three detection algorithms to accurately detect security problems.

#### Deadlock detection

We describe the situation of deadlock threads. Thread A requests acquiring lock L2 while holding lock L1, and thread B requests acquiring lock L1 while holding lock L2. The example program is shown in Figure 3.

In Algorithm 1, the main idea is, first, to acquire the monitor object of the synchronized block and then acquire the pointed address of the monitor object. Finally, if the pointed address of the monitor object in the two different synchronized blocks is the same, we detect a deadlock.

Method doPerformAnalysis is the step to perform the algorithm. javaProject is a Java project that needs to be detected, and basicAnalysisData contains multiple variables for analysis.

Method getSynchronizedClassTypeNames accesses the monitor instruction instructionInfo and acquires instances that meet the conditions. Method populateSynchronizedBlocksForNode calls the method ComparedVariable. If we access the monitor instructions, we will call the method getAccessedField.
Algorithm 1. Deadlock detection algorithms

Input: javaProject, basicAnalysisData
Output: bugInstances
BugInstances doPerformAnalysis(javaProject javaProject, BasicAnalysisData basicAnalysisData)
if instructionInfo belong to isMonitorEnter then
    synchronizedClassTypeNames ← getSynchronizedClassTypeNames ((SSAMonitorInstruction) instructionInfo, cgNode)
    bugInstances ← synchronizedClassTypeNames
end
return bugInstances
Set < String > getSynchronizedClassTypeNames(SSAMonitorInstruction monitorInstruction, CGNode cgNode)
if instructionInfo ≠ null then
    populateSynchronizedBlocksForNode(bugInstances, instructionInfo)
end
return instructionInfo
BugInstances populateSynchronizedBlocksForNode(BugInstances bugInstances, InstructionInfo instructionInfo)
if instructionInfo ≠ null then
    ComparedVariable(instructionInfo)
end
return instructionInfo
Collection < InstructionInfo > ComparedVariable(InstructionInfo instructionInfo)
if instructionInfo is a monitor instruction then
    getAccessedField(instructionInfo)
end
return instructionInfo
Collection < InstructionInfo > getAccessedField(InstructionInfo instructionInfo)
get the instance pointed to by instructionInfo
if the next element pointed to by pointedInstances exists then
    i ← System.identityHashCode(pointedInstances)
    j ← System.identityHashCode(pointedInstances.hansNext())
    if j != MonitorExit then
        i ← System.identityHashCode(pointedInstances)
        j ← System.identityHashCode(pointedInstances.hansNext())
        if i == j and j == i then
            get the instructionInfo instruction of the instance pointededInstances and store
        end
    end
end
return instructionInfo

public class Deadlock{
    public static Object obj1 = new Object();
    public static Object obj2 = new Object();
    public void one(){
        synchronized (obj1) {
            synchronized (obj2) {
                ......
            }
        }
    }
    public void another(){
        synchronized (obj2) {
            synchronized (obj1) {
                ......
            }
        }
    }

    Figure 3. Deadlock example.

Method getAccessedField is the core part of the algorithm. We acquire the instruction pointed to instances pointedInstances. This step of the algorithm assigns the pointed address of the instance to i and assigns the pointed address of the next instance to j which is used as the final object of judgment.

Object reusability detection

The object reuse problem is very likely to occur in synchronized methods or blocks when the lock objects are Boolean, Integer, or String objects. For example, a Boolean object has only two values: true and false. If we use a Boolean object as the monitor object, the object may point to the same address and cause problems. In Figure 4, the lock monitor object is a Boolean object in a synchronized block. Because the two constants, Boolean.FALSE and false, represent the same memory location, they are the same synchronized
object, which makes the resources access mutually exclusive.

Algorithm 2 is the object reuse detection algorithm we designed. By detecting the type of a monitor object, we can determine whether the monitor object is a Boolean, an Integer, a String, or other types. If the type is a reusable type, we output the detection result.

Method doPerformAnalysis is the step to execute the algorithm. If the program method is detected to be not empty, we will call the method populateBugInstances to detect the monitor object and assign it to the instance bugInstances to acquire the final reused object.

Method populateBugInstances determines whether the “acquire” instruction is a type of reused object. The instruction acquire instruction must be a monitor instruction. We assign a value to reusableLockObjectTypes by calling the method getReusableLockObjectTypes. If the reusableLockObjectTypes is consistent with the object reuse type, we return bugInstances.

Method getReusableLockObjectTypes analyzes the instruction to acquire the lock object type. We acquire the pointed address of the monitor instruction monitorInstruction and assign it to the instanceKey. We use the method createReusableChecker to determine whether the instanceKey is a reusable object.

```java
public void anotherMethod() {
    synchronized(anotherBoolean) {
        //TODO
    }
    ...
}
```

Figure 4. Object reusable example.
Method createReusableChecker is the core part of the algorithm to detect the type of instanceKey. If its type is a reusable type such as Boolean type, Integer type, or String type, it is detected that the program has object reuse problems.

**Static shared field detection**

For software programs, shared resources are subjected to conflicts due to simultaneous access by multiple threads. As shown in Figure 5, they create two instances of the monitor object when two runnable tasks start. In this situation, it locks two instances, separately.

Algorithm 3 is the static shared field detection algorithm. The algorithm acquires all static shared fields and checks whether the field has been modified in the program. If it is modified, it acquires the pointed instance of the field and outputs the detection result.

![Figure 5. Shared static field example.](image)

Method doPerformAnalysis is the step to perform the algorithm. We call the method getAllStaticFields to acquire the static field and store the detected field in staticFields.
Method `populateAllInstancesPointedByStaticFields` acquires the static field pointed to the instance `pointededInstances` and stores all the static fields pointed to the instance `pointededInstances`.

Method `populateModifyingStaticInstancesMap` acquires the modify static instance. If the modified instruction `modificationInstruction` instance belongs to `pointededInstances`, we acquire the instance `bugInstances` of the instruction `modificationInstruction`.

Method `getModifyingStaticFieldsInstructions` is the core part of the algorithm. It determines the static field that needs to be modified by calling the method `canModifyStaticField`. If the field instruction `instruction` is static, the detection is successful.

**Evaluation**

**Benchmarks**

We select four benchmarks to evaluate our refactoring tool. Quark is an open-source tool for developing applications for networked devices based on IoT sensing data. JGroups is an open-source group communication tool. Apache Mina is a network communication application framework, but it mainly provides a programming model for event-driven and asynchronous operations based on the IoT TCP/IP and UDP/IP protocol stacks. In addition, the Apache Mina-core is a core network application framework and HSQLDB is a small database.

Table 2 shows the benchmarks and their respective attributes. The second column represents the total number of classes in the program; the “Method” column represents the number of methods in the benchmark; “Sync” represents the number of methods that may involve synchronization; and “No sync” represents the number of methods not to involve synchronization.

In summary, the result shows that our analysis can search synchronization methods in real-world programs and analyze their synchronization dependencies. All experiments were conducted on a 16-core 2.60 GHz Intel Xeon E5-2650 workstation with 128GB RAM. The workstation ran on Windows 7 operating system with Eclipse 4.5.1 and JDK 1.8.0 installed.

**Case study.** The importance of IoT software is highlighted in the “Introduction” section. IoT software is subjected to security problems. In many cases, refactoring does not preserve program behaviors in the presence of concurrency. The new behavior will cause problems that did not exist before refactoring, such as security problems and deadlock.

Figure 6 is the benchmark Mina-core, which classifies the synchronized blocks to parent classes `AbstractAcceptor`. The original program is identified to have no problems. After refactoring, thread A acquires the bindLock lock, which acquires the boundAddresses lock. Thread B acquires the boundAddresses lock, which acquires the bindLock lock. By using Algorithm 1 to acquire the pointed address of a monitor object of a synchronized block, we found that the pointed addresses `i (boundAddresses)` and `j (bindLock)` were the same. We determined that a deadlock occurred after the refactoring and caused security problems of IoT software.

**Experimental results and analysis**

**Experimental results.** The refactoring tool Eclipse was used to convert the benchmarks. We evaluated all the benchmarks, except for Mina which only detected the core package Mina-core.

We refactored the software in each benchmark. By executing three detection algorithms, we detected the existing problem in each of the benchmarks. For example, we found deadlock problem in Quark, Mina-core, and HSQLDB. We detected the object reuse problem and static shared field problem in JGroups.

By using the three algorithms, we detected the three problems (such as deadlock, object reusable, and static shared field). The experimental results are given in Table 3. We assessed the number of inconsistencies and detection time in all benchmarks. Detection of inconsistency indicates that the problems can occur in the refactored program. The detection time shows that our tools are efficient in a short time.

**Table 2.** Benchmarks and their attributes.

| Benchmark | Class  | Method  | Sync  | No sync |
|-----------|--------|---------|-------|---------|
| Quark     | 716    | 16,647  | 317   | 16,330  |
| JGroups   | 715    | 10,635  | 494   | 10,141  |
| Mina-core | 1461   | 23,187  | 235   | 22,952  |
| HSQLDB    | 1257   | 11,337  | 2517  | 8820    |
| Total     | 4149   | 61,806  | 3563  | 58,243  |

**Table 3.** Experimental results.

| Benchmark | Inconsistent | Consistent | Time (s) |
|-----------|--------------|------------|----------|
| Quark     | 2            | 317        | 4.63     |
| JGroups   | 2            | 492        | 8.71     |
| Mina-core | 4            | 231        | 3.08     |
| HSQLDB    | 2            | 2515       | 10.08    |
| Total     | 10           | 3555       | 26.50    |

**Table 2.** Benchmarks and their attributes.

**Table 3.** Experimental results.
Conclusion

This article presents a detection approach which uses control flow analysis, synchronization dependency analysis, data flow analysis, and three detection algorithms to ensure consistency and security of IoT software. Static analysis analyzes the structure of changes, and the three detection algorithms are used to detect software security problems. The three detection algorithms solve three problems: deadlock, object reuse, and static shared field. In the experiment, we evaluated our approach by four benchmarks, that is, Quark, JGroups, Mina-core, and HSQLDB. Experimental results show that our approach is efficient in detecting existing problems.

One possible area of future work would be to explore more complex refactoring detection beyond the field of IoT software. For instance, some advanced refactorings incur new problems and lead to more challenges in software development. The approach proposed herein is not enough to solve all of the problems, but the concepts and techniques developed in this study are expected to serve as a basis for addressing new challenges.

Acknowledgements

The authors gratefully acknowledge the helpful comments and suggestions of the reviewers.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the National Key Research and Development Plan (grant no. 2018Y
1. Tellez M, El- Tawab S and Heydari MH. IoT security attacks using reverse engineering methods on WSN applications. In: Proceedings of the 2016 IEEE 3rd world forum on internet of things (WF-IoT), Reston, VA, 12–14 December 2016, pp. 182–187. New York: IEEE.

2. Qui J, Du L, Zhang D, et al. Nei-TTE: intelligent traffic time estimation based on fine-grained time derivation of road segments for smart city. *IEEE T Ind Inform*. Epub ahead of print 26 September 2019. DOI: 10.1109/TII.2019.2943906.

3. Tian Z, Gao X, Su S, et al. Vcash: a novel reputation framework for identifying denial of traffic service in internet of connected vehicles. *IEEE Internet Things*. Epub ahead of print 5 November 2019. DOI: 10.1109/JIOT.2019.2951620.

4. Smith S. *IoT connected devices to triple to over 38Bn units*. Basingstoke: Juniper Research, 2015.

5. Xiuo Y, Rayi V, Sun B, et al. A survey of key management schemes in wireless sensor networks. *J Comput Commun* 2017; 30(11–12): 2314–2341.

6. Xiao Y, Li X, Huang X, et al. Cloud-based malware detection game for mobile devices with offloading. *IEEE T Mobile Comput* 2017; 16(10): 2742–2750.

7. Akgun M and Caglayan MU. Providing destructive privacy and scalability in RFID systems using PUFs. *Adhoc Netw* 2015; 32: 32–42.

8. Tian Z, Cui Y, An L, et al. A real-time correlation of host-level events in cyber range service for smart campus. *IEEE Access* 2018; 6: 35355–35364.

9. Tian Z, Luo C, Qiu J, et al. A distributed deep learning system for web attack detection on edge devices. *IEEE T Ind Inform*. Epub ahead of print 30 August 2019. DOI: 10.1109/TII.2019.2938778.

10. Yin L, Luo X, Zhu C, et al. ConnSpoiler: disrupting C&C communication of IoT-based Botnet through fast detection of anomalous domain queries. *IEEE T Ind Inform*. Epub ahead of print 11 September 2019. DOI: 10.1109/TII.2019.2940742.

11. Tian Z, Gao X, Su S, et al. Evaluating reputation management schemes of internet of vehicles based on evolutionary game theory. *IEEE T Veh Technol* 2019; 68(6): 5971–5980.

12. Shen M, Ma B, Zhu L, et al. Cloud-based approximate constrained shortest distance queries over encrypted graphs with privacy protection. *IEEE T Inform Forens Secur* 2018; 13(4): 940–953.

13. Tian Z, Li M, Qiu M, et al. Block-DEF: a secure digital evidence framework using blockchain. *Inform Sci* 2019; 419: 151–165.

14. Xu T, Wendt JB and Potkonjak M. Security of IoT systems: design challenges and opportunities. In: *Proceedings of the IEEE/ACM international conference on computer-aided design*, San Jose, CA, 2–6 November 2014. New York: IEEE.

15. Zhang ZK, Cho MCY, Wang CW, et al. IoT security: ongoing challenges and research opportunities. In: *Proceedings of the 2014 IEEE 7th international conference on service-oriented computing and applications (SOCA)*, Matsue, Japan, 17–19 November 2014. New York: IEEE.

16. Conti M, Delfhantanz A, Franke K, et al. Internet of things security and forensics: challenges and opportunities. *Future Gener Comput Syst* 2018; 78: 544–546.

17. Xu X. Study on security problems and key technologies of the internet of things. In: *Proceedings of the 2013 international conference on computational and information sciences*, Shijiang, China, 21–23 June 2013, pp. 407–410. New York: IEEE.

18. Fuji N and Koike N. IoT remote group experiments in the cyber laboratory: a FPGA-based remote laboratory in the hybrid cloud. In: *Proceedings of the 2017 International Conference on Cyberworlds (CW)*, Chester, 20–22 September 2017, pp. 162–165. New York: IEEE.

19. Liu Y, Briones J, Zhou R, et al. Study of secure boot with a FPGA-based IoT device. In: *Proceedings of the 2017 IEEE 60th international Midwest symposium on circuits and systems (MWSCAS)*, Boston, MA, 6–9 August 2017, pp. 1053–1056. New York: IEEE.

20. Tan Q, Gao Y, Shi J, et al. Toward a comprehensive insight into the eclipse attacks of Tor hidden services. *IEEE Internet Things* 2019; 6(2): 1584–1593.

21. Dong P, Du X, Zhang H, et al. A detection method for a novel DDoS attack against SDN controllers by vast new low-traffic flows. In: *Proceedings of the IEEE international conference on communications (ICC)*, Kuala Lumpur, Malaysia, 22–27 May 2016. New York: IEEE.

22. Xiao L, Wan X, Dai C, et al. Security in mobile edge caching with reinforcement learning. *IEEE Wirel Commun* 2018; 25(3): 116–122.

23. Wu L, Du X, Wang W, et al. An out-of-band authentication scheme for internet of things using blockchain technology. In: *Proceedings of the international conference on computing, networking and communications (ICNC)*, Maui, HI, 5–8 March 2018. New York: IEEE.

24. Tian Z, Shi W, Wang Y, et al. Real time lateral movement detection based on evidence reasoning network for...
Zhu L, Tang X, Shen M, et al. Privacy-preserving DDoS attack detection using cross-domain traffic in software defined networks. *IEEE Internet Things* 2018; 6(2): 1584–1593.

Steimann F. Refactoring tools are trustworthy enough and trust must be earned. *IEEE Software* 2015; 32(6): 80–83.

Schafer M, Julian D, Manu S, et al. Correct refactoring of concurrent java code. In: *Proceedings of European conference on object-oriented programming*, Maribor, 21–25 June 2010, pp. 225–249. New York: Springer.

Maruyama K, Hayashi S, Yoshida N, et al. Frame-based behavior preservation in refactoring. In: *Proceedings of IEEE international conference on software analysis, evolution and reengineering*, Klagenfurt, 20–24 February 2017, pp. 573–574. New York: IEEE.

Zhang Y, Shao S, Liu H, et al. Refactoring Java programs for customizable locks based on bytecode transformation. *IEEE Access* 2019; 7(1): 66292–66303.

Zhang Y, Dong S, Zhang X, et al. Automated refactoring for stampedlock. *IEEE Access* 2019; 7(1): 104900–104911.

Carlos P and Michael D. Randoo: feedback-directed random testing for Java. In: *Proceedings of the OOPSLA’07: Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion*, Montreal, QC, Canada, 21–25 October 2007, pp. 815–816. New York: ACM.

Gordon F and Andrea A. Evosuite: automatic test suite generation for object-oriented software. In: *Proceedings of the ACM symposium on the foundations of software engineering (FSE)*, Szeged, 5–9 September 2011, pp. 416–419. New York: ACM.