Received November 19, 2019, accepted December 11, 2019, date of publication December 20, 2019, date of current version December 31, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2961144*

# A Dynamic Taint Analysis Framework Based on Entity Equipment

**YUZHU REN, WEIYU DONG, JIAN LIN, AND XINLIANG MIAO**
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
Corresponding author: Weiyu Dong (dongxinbaoer@163.com)

**ABSTRACT** With the development of the Internet of Things, the security of embedded device has received extensive attention. Taint analysis technology can improve the understanding of the firmware program operating mechanism and improve the effectiveness of security analysis. It is an important method in security analysis. Traditional taint analysis of embedded device firmware requires complex pre-preparation work, setting up a virtual operating environment. Those security analysts have to invest a lot of time and effort in this work, and the results are usually unsatisfactory. In this paper, we propose a dynamic taint analysis method based on entity equipment. The core idea of our approach is to divide the taint analysis into two parts: the simulation analysis on the host and the real execution on the entity equipment. Since one of the features of our method is based on entity equipment, there is no need to build a dedicated virtual environment. Another feature is that the tested firmware program runs on entity equipment and can ensure the accuracy of the analysis by comparing the results of the taint analysis with the device firmware run-time information. We implement a prototype system and verified the effectiveness of the method, which can perform taint analysis on multiple architecture embedded firmware programs and detect vulnerabilities such as stack overflow, heap overflow and so on. Finally, we verify our prototype with a test case to effectively detect vulnerabilities in the firmware program. And we evaluate the performance of the prototype, compared with PANDA, the time overhead of our prototype is reduced by 5.9%.

**INDEX TERMS** Embedded firmware, dynamic taint analysis, cross-debugging.

## I. INTRODUCE

With the pervasive application in economic development and social life, embedded devices have become more and more important. The need for security testing of embedded device firmware has also emerged. Taint analysis technology has been widely used in software vulnerability analysis [1]. The core principle is to add a ***taint*** label to the data, track the propagation process of ***taint*** data during program running, and detect ***taint*** labels in the sensitive data area or when programs call APIs to send data out [2]. Through the taint analysis of programs, we can have a deeper understanding of the mechanism of the vulnerability.

Due to its advantages, taint analysis has been applied to many research fields such as vulnerability mining [3]–[6] and sensitive data leakage detection [7]–[10]. However, there are some difficulties in applying taint analysis to embedded device firmware analysis. One obvious reason is that for commercial confidentiality. Embedded device manufacturers generally do not provide firmware source code, so security researchers can only use binary code to analyze the security performance of embedded device firmware. In the binary code analysis of embedded device firmware, it is usually necessary to set up a virtual environment. Since the platform architecture and dependent libraries of the embedded device firmware are different, various problems often occur when setting up the virtual environment [11]. This will occupy a large number of researchers' time and effort. Therefore, security analysts urgently need an analysis tool with a simple environment and a wide range of applications, which can perform taint analysis on the firmware program if the system performance overhead can be tolerated.

In this paper, we design a taint analysis framework based on entity equipment, which can perform taint analysis to embedded device in real environments. The core idea is to divide taint analysis into simulation analysis on the host and real execution on embedded devices. The simulation analysis on the host, which can be seen as a monitor and

The associate editor coordinating the review of this manuscript and approving it for publication was Md. Arafatur Rahman.

replicator, obtains run-time information of the program firmware through cross-debugging, performs simulation execution and taint analysis on the host. The results of the simulation execution are compared with the results of the device firmware operation to ensure the correctness of the simulation execution. Our method does not need to build a completed firmware virtual environment on the host. Therefore, we have reduced the manual work required to build a virtual environment.

We use cross-debugging provided by embedded system vendors to achieve coordination between simulation analysis and real execution. For the simple analysis environment and strong adaptability, the dynamic taint analysis technology based on entity equipment proposed in this paper has a very broad application prospect in firmware program security analysis.

We implement our method by gdb. Then, the prototype system is evaluated by test machine to verify the validity and effectiveness of the prototype. The results show that our prototype achieves a certain degree of analytical accuracy with acceptable performance overhead.

Contributions of this work are:

1) We design a dynamic taint analysis framework based on entity equipment. The method performs the taint analysis and simulation analysis on the host. At the same time, firmware program performs real execution on the target device. The host obtains the run-time information of the target device through cross-debugging. The advantage of our design is that the bottleneck of the hardware and software capabilities of the embedded device can be avoided by performing taint analysis on hosts. It can also save the preparation for setting up the virtual environment.

2) We implement a prototype system and test it. The system can effectively detect firmware vulnerabilities such as stack overflow and heap overflow, and also can mark the output formatted strings with ***taint*** label.

3) We study the impact of the prototype on the efficiency of the target program and evaluate it with the bin64 toolset.

The paper is organized as follows. Section II reviews related works briefly. Motivations of this work are described in Section III. Section IV presents the overall design of the system and the functions of each module. Section V introduces the algorithm of our method. Section VI details the implementation on gdb, and Section VII examines the actual case. Section VIII describes the experiment and analysis of the test set, Section IX discusses the limitations of our approach, as well as possible solutions, and Section X concludes the paper.

## II. RELATED WORK

Taint analysis is generally divided into hardware-based, virtual-environment-based, software-based, and code-based four classes [12], each of which has its own characteristics.

Hardware-based taint analysis tools such as Raksha, FlexiTaint and PIFT [13], need to expand the system at the hardware level, redesign the hardware structure of registers, caches and memory, and add corresponding taint information labels to achieve the storage, dissemination and detection function of taint information. Although such methods have low system overhead and high efficiency in taint analysis, they are too laborious and not portable for embedded device firmware analysis.

Virtual-environment-based taint analysis tools [14], [15] usually add a taint analysis module in the virtual environment, so that the taint analysis module and the target program run in different layers. The taint analysis module of Phosoher [16] runs on the virtual machine monitoring layer, and target programs run on the target operating system layer. Since the analysis module runs on the bottom layer of the target program, the taint information recorded by such methods is more accurate. However, this method requires virtual environments to run firmware programs. This preparation is also very complicated due to the different architectures and dependencies of the firmware program. For example, FIRMADYNE [11] can only successfully simulate 8617 firmware images while 23035 firmware images is downloaded.

Software-based taint analysis mainly performs taint analysis on operating systems and applications. The core idea is to mark related resources in operating systems or applications, and perform taint analysis on this basis, such as TaintDroid [17]. This method requires frequent instrumentation or code rewriting, and the analysis efficiency is greatly reduced. Subject to the hardware and software environment of embedded device, this approach is rarely used in embedded device environments. There are similar jobs in reference [18]–[21].

Code-based taint analysis tools [19], [22]–[24] often fail to analyze source code for commercial reasons, and only binary code analysis is possible. The main practice is to insert taint tracking codes in the binary code to get taint propagation information when the program is running. Dynamic taint analysis often uses this approach. Taint analysis of binary code can perform taint tracking detection at the instruction level with high accuracy, but frequent instrumentation will occupy system resources and reduce system operation efficiency. In addition, the type and size of the computer's instructions make it extremely difficult to accurately model binary code, and binary code lacks high-level semantic support, and generally relies on instrumentation platforms or binary disassembly tools to reduce the workload.

The analysis objects of such hybrid execution tools are generally intermediate representation languages translated from machine code. The advantages include two aspects: First, the hybrid execution tool based on intermediate representation languages can be independent of the architecture, that is, if the translation module of the specific architecture is available, the analysis work can be used to test the firmware program based on different platforms; Second, a lot of development work can be saved because the workload of analyzing intermediate representation languages is much lower than the analysis of machine language. The above hybrid execution tools were originally designed to alleviate the work required for code migration, and to solve the problem that the computational performance of the embedded system is difficult to meet the symbol execution, but they are good references for our work.

In existing works, researchers have applied symbol execution and hybrid execution to the testing of embedded device firmware. For example, Mayhem [25] uses the BAP framework [26] to convert instructions into IR statements, so Mayhem is independent of the architecture. Overall, Mayhem separates concrete executions of the hybrid execution from symbolic executions, where concrete executions are performed by target devices and symbolic executions are performed on hosts. Later, Chen *et al.* [27] improved this structure, and realized a hybrid execution tool with lower development cost and better consistency, and implemented on the embedded system VxWorks. Specifically, the concrete implementation only includes the CE core responsible for concrete implementations and the debugging agent for cross-debugging, this design were more suitable for the limited hardware resource environment of the embedded system.

But ours is different from the existing work [28]–[31]. Take Avatar as an example, there are three main differences between our work and Avatar. Firstly, In Avatar the firmware runs in the emulator while in our work it runs in the real device. Secondly, the target of Avatar is devices without operating system, such as Hard Disk Drive and Common Feature Phone. While our target is embedded devices with Unix-like operating system, such as Network printer. Thirdly, Avatar is a framework which requires a JTAG-based hardware debugger, so it needs some hardware's help like jlink. And our prototype can perform safety analysis only by using cross-debugging tools such as gdb.

Our work implements a general approach to performance taint analysis of embedded device firmware, as well as details of the prototype system and experiments implemented on gdb.

## III. MOTIVATION

**The defect in embedded device firmware** is the main reason we conducted this research. Figure 1 shows the trend of the numbers of vulnerabilities found in several popular embedded systems (Android, Apple iOS, and Cisco iOS). From the statistics, we find that so many vulnerabilities were discovered every year.

From our perspective, one major reason is **inadequate testing technique**. A common method of performing security testing on embedded device firmware is black box testing. Test cases are generated randomly, or some heuristics. Then test cases are imported into the program, observing and analyzing the output of the program. If the output is incorrect, it means that a potential vulnerability has been discovered. One disadvantage of the black box testing is that the internal state of the tested program cannot be obtained, and the researchers cannot obtain the mechanism of the vulnerability. Another disadvantage is about code coverage of black box testing. Even if all the test inputs generated by the black box

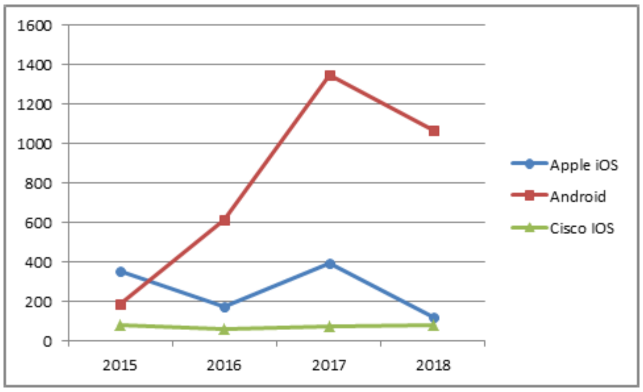


**FIGURE 1. Trend of the numbers of discovered vulnerabilities from Jan. 1, 2015 through Dec. 31, 2018 (data collected from National Vulnerability Database. http://nvd.nist.gov/).**

testing have been executed, there may still be a large number of vulnerabilities in the tested program.

In recent years, binary symbol analysis of device firmware has become a hotspot and has been widely used in many cyber security challenges. However, in practical applications, problems such as path space heuristic search, constraint solving, parallel processing, memory modeling and performing environmental simulations require further research and improvement.

**The effectiveness of taint analysis** has been verified on general-purpose computer platforms, which is our second motivation. Researchers have developed a number of sophisticated PC-based taint analysis tools, including TEMU, FlowDroid [32] and TaintDroid, for reference.

**The instrumentation can obtain run-time information on the tested firmware**, which can be used to perform the taint analysis of the embedded device firmware. Instrumentation is a commonly used method for dynamic taint analysis of the tested program on PC platforms. But the use of instrumentation on embedded platforms is also a more complicated task, because instrumentation is closely related to the operating systems and hardware structures.

Fortunately, cross-debugging can be obtained from vendors that allow third parties to develop applications for their embedded systems. For example, embedded Linux can be debugged through gdb; WTX protocol is the debugging technology provided by Wind River for VxWorks; Windows CE debugging function is integrated in Platform Builder IDE; CodeWarrior Development Studio provides debugging function for Palm; we can separately get run-time information for Android and iOS apps through Android Debug Bridge and debugserver. Our approach can be applied to different embedded systems through cross-debugging techniques provided by different platforms.

## IV. SYSTEM ARCHITECTURE

In this section, we first introduce the overall framework and briefly introduce each module, and then illustrate the collaboration between modules through a typical taint analysis. Figure 2 demonstrates the system framework.

In our framework, the firmware program is actually executed on the device. The host gets the run-time information of the firmware program by using cross-debugging tools such as gdb. Then the host performs taint analysis by using the run-time information. The results performed by the host are checked against the values in the real device to ensure the accuracy of the analysis. Of course, we design some modules to implement the functions.

### A. DESCRIPTION OF MODULES

The function of the ***gdb Server*** module is to debug the target firmware program and get the run-time information. Based on commands received from the ***gdb + instrumentation engine*** module, the ***gdb Server*** can start and stop debugged programs, set breakpoints, perform individual steps. Run-time information obtained from programs includes executed instructions, registers, and values in memory.

The ***gdb + instrumentation engine*** is the only interface between the host and the embedded device. Its function is to send the obtained program's run-time information to the ***IR lifter*** module. Another function of the ***gdb + instrumentation engine*** is to accept commands from the ***taint analysis engine*** and sends them to the ***gdb Server*** module. In addition, it has a special function named ELF-file-parsing to parsing the ELF-file and deal with the preprocess work.

The ***IR lifter*** module is to translate machine instructions into intermediate representation statements. The inputs are machine instructions from the ***gdb + instrumentation engine***, and the outputs are intermediate representations that can be interpreted by the ***taint analysis engine***. The ***IR lifter*** module relies on the embedded device firmware architecture, we implemented the prototype system on gdb, and we used the VEX module under the Valgrind framework for IR lifter.

The ***taint analysis engine*** is the core part of the framework. Its main functions are to set taint labels, track the spread of taint data by invoking the taint-propagation-strategy function, and detect taint labels at the taint convergence points. The ***taint analysis engine*** has a function named check to compare and verify the results executed by the ***taint analysis engine*** with the information obtained by the ***gdb Server*** and the ***gdb + instrumentation engine***. Because the taint analysis engine is simulation, the results executed may be inconsistent with the values performed by the embedded device firmware, this will affect the accuracy of the taint analysis.

The functions of the ***storage*** module are to record the IR file generated by the ***IR lifter*** module and the values of the registers and memory calculated by the ***taint analysis engine***, as well as the taint markers, and save them in the taint analysis file database for replay analysis.

### B. COOPERATION OF MODULES

In this section, a typical taint analysis process is used to describe the collaborative process between all modules.

At first, the ELF-file-parsing function of the ***gdb + instrumentation engine*** module parse related information about device's platform architecture and instruction set.

Instrumentation: 1) The ***gdb + instrumentation engine*** module sends gdb debug commands to the ***gdb Server*** of the entity equipment, and the firmware program is aborted at the specified breakpoint; 2) The ***gdb Server*** returns command response and device firmware run-time information (in the form of a basic block file) to the ***gdb + instrumentation engine*** module; 3) The ***gdb + instrumentation engine*** calls the subsequent processing function to perform subsequent processing steps on the obtained basic block file.

The IR lifter: 4) The ***IR lifter*** module converts the resulting basic block file into an IRSB. At the same time, the IRSB information is stored into the VEX information storage area ***VCache***.

Taint analysis: 5) The ***taint analysis engine*** invokes the taint-propagation-strategy function to perform the taint analysis process and stores the information after execution about memory, registers and temporary variables in the ***MCache RCache*** and ***TCache*** of the ***storage*** module; 6) After the taint analysis to the IRSB, the ***taint analysis engine*** passes the next-PC-address to the ***gdb + instrumentation engine*** module. And the ***gdb + instrumentation engine*** module repeats the previous process based on the next-PC-address.

After the entire taint analysis is completed, the information stored in the ***storage*** is output to the ***taint analysis file database***.

### C. ADVANTAGES OF THE ARCHITECTURE

The advantages of the architecture can be seen in Figure 2.

1) Our method overcomes the hardware limitations of embedded system devices, making it possible to perform taint analysis on the embedded firmware of the entity equipment. And with this architecture, we can analyze and test any embedded device that supports remote debugging, this is more portable.

2) Save a lot of work on environment. Figure 2 shows that most work of the taint analysis (ELF file parsing, IR conversion, taint analysis, data saving, result verification, etc.) occurs on hosts. Therefore, our method is not limited by the hardware resources of the device. In addition, the firmware program runs on the physical device the architecture, our prototype obtains related information through cross-debugging. So researchers do not need to set up a virtual environment, which is much less expensive than virtual-environment-based taint analysis.

3) Get accurate run-time information. Due to the use of the instrumentation module, the taint analysis framework can obtain accurate run-time information, and cooperate with the check function to compare and verify the results of the ***taint analysis engine***. This overcomes the defects of simulation execution.

4) No need to obtain firmware program source code. With the increase in aggressive behavior against embedded devices, embedded device vendors are also upgrading the level of security for firmware, and security researchers have difficulty in accessing firmware source code. This framework can overcome this shortcoming.

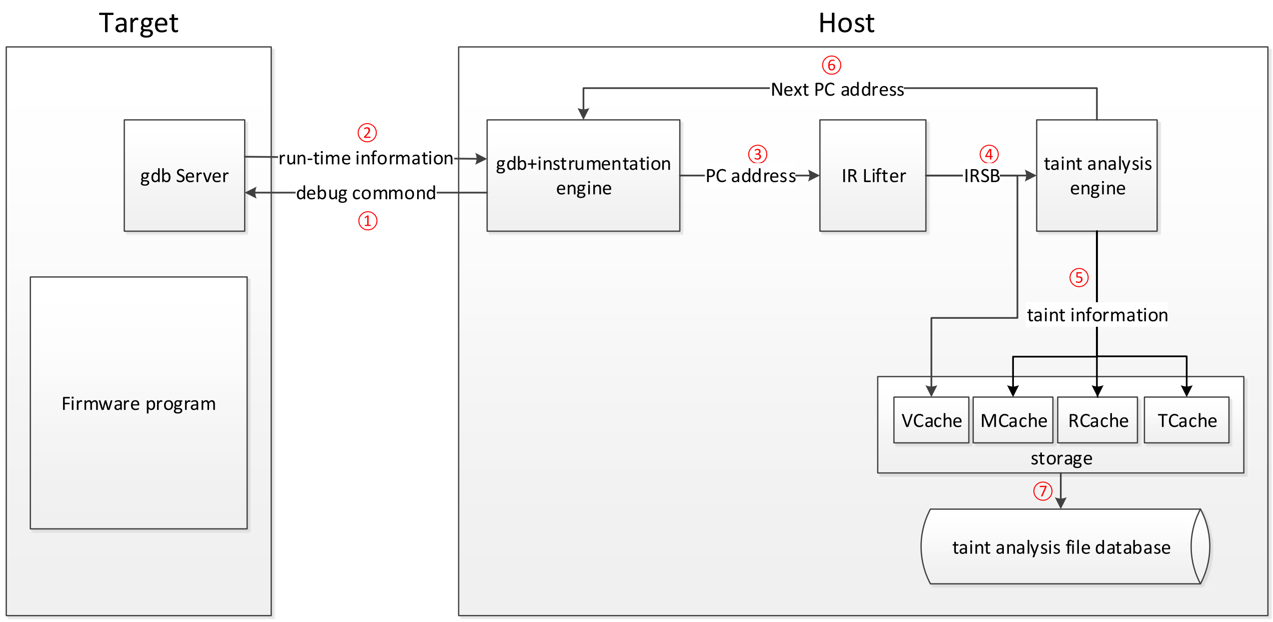


**FIGURE 2.** Dynamic taint analysis framework.

## V. ALGORITHM

Algorithm 1 shows the algorithm of our method. The tested program ***P*** and the initial taint source ***S*** are the inputs of the algorithm. The way to introduce taint sources will be described in the next section. The output of the algorithm is a set of formatted results generated according to our method.

**Algorithm 1** Algorithm of the Proposed Method

```
Parameteres: Program $P$, Taint sources set $S$
Results: $R, M, T$
1:  $I = ELF\_parsing(P)$
2:  $sim\_state(R, M, T) = initialize(P, I)$
3:  while $not\_end\_of(P)$ and $not\_breakpoint\_of(P)$ do
4:      $B = current\_basicblock(P)$
5:      $IR = ir\_translate(B)$
6:      for $I$ in $IR.inst$ do
7:          $set(value, label) = simulation\_execution(I, S)$
8:          $sim\_state(R, M, T) = update(set(value, label))$
9:      end for
10:     $real\_value(R, M, T) = instrumentation(bp')$
11:     if $sim\_state(R, M, T) == real\_value(R, M, T)$ then
12:         $save\_state(R, M, T)$
13:     else
14:         return -1
15:     end if
16: end while
```

The analysis framework parses the ELF file to obtain related information in Line 1. Then an initial simulation state that includes registers ***R***, memory ***M***, and temporary variables ***T*** is created in Line 2. Starting from Line 3, the taint analysis framework runs the program until the program finishes running, or a termination command is met. Typical termination command includes running out of preset test times, reaching preset program breakpoints, or throwing uncaught exceptions. The first step of the loop is to read the current basic block to be executed from the running program, and then translate it into a set of intermediate representation statements (Lines 4, 5).

The algorithm then performs corresponding operations depending on the type of operations. The result is a set of values and labels about simulation memory, registers and so on. The set will be used to update the initial simulation state. After a basic block is executed, the values in the initial simulation state are compared with the values obtained by instrumentation.

At a higher level, all operations are divided into three categories such as instrumentation, intermediate representation conversion, taint analysis. If the operation type is to obtain data input, perform the instrumentation operation to obtain related information when the program is running; if the operation type is a conversion instruction form, convert the obtained binary instruction into an intermediate representation language; if the operation type is to perform taint analysis, The taint analysis engine is called for taint analysis. When the result of comparison is consistent, the program continues, or an error is raised if a mistaken occurs.

## VI. IMPLEMENTATION

We implement our prototype based on our designed framework. In this section, we will focus on the implementation of the system. The prototype system is designed for embedded systems on general-purpose computers. The total code amount of the system is about 12,000 lines of Python code without comments.

Note that the implementation details described in this section are specific to gdb, but the architecture and algorithm

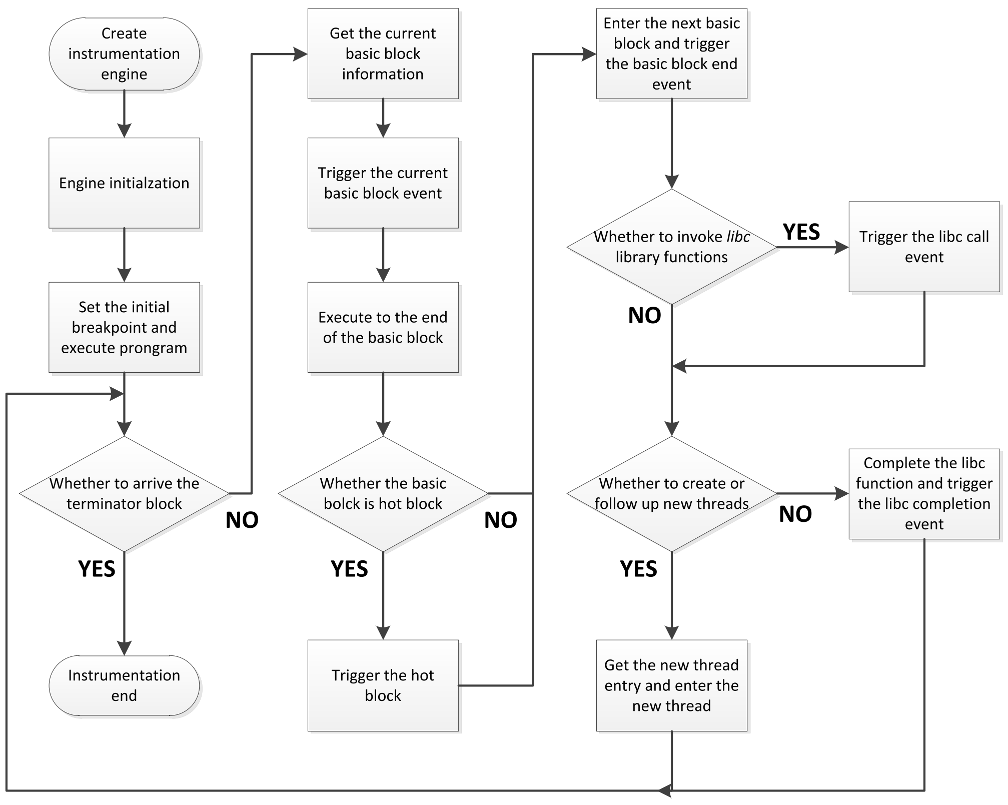

Create instrumentation engine
Engine initialzation
Set the initial breakpoint and execute prongram
Whether to arrive the terminator block
YES
NO
Instrumentation end
Get the current basic block information
Trigger the current basic block event
Execute to the end of the basic block
Whether the basic bolck is hot block
YES
NO
Trigger the hot block
Enter the next basic block and trigger the basic block end event
Whether to invoke libc library functions
YES
NO
Trigger the libc call event
Whether to create or follow up new threads
YES
NO
Get the new thread entry and enter the new thread
Complete the libc function and trigger the libc completion event

**FIGURE 3. Dynamic instrumentation flow chart.**

of our method are generic and applicable to other embedded systems.

### A. INITIALIZATION AND DYNAMIC INSTRUMENTATION

The system first creates and initializes the ***gdb + instrumentation engine*** module. The initialization is mainly divided into two parts. One is to initialize the information and some parameters of the system platform and firmware program obtained in the ***ELF file parsing*** module. This is due to the difference between the instruction set, register type and byte order of different platform architectures. The other part is to register the taint analysis event and the signal, address, etc. of interest. This part mainly provides an interface for the ***simulation taint analysis engine*** module, which facilitates the analysis of the basic block.

The initial breakpoint is set after initialization to start executing the program. The prototype system analyzes in units of basic blocks, triggers taint analysis events at the beginning and end of basic blocks for simulation execution, taint analysis, and verification. When the target program calls library functions, the taint propagation analysis can be performed directly according to the function of library functions to improve the efficiency of the taint analysis.

### B. INTERMEDIATE REPRESENTATION CONVERSION

In order to implement the taint analysis tool based on intermediate representation language, we need to add a module to convert the machine code into intermediate representation statements. Existing binary translation frameworks such as LLVM (static) and Valgrind (dynamic) already have this function. Valgrind does not require program source code and can operate on executable files. Currently, it supports multiple mainstream platforms (x86, ARM, MIPS, PPC). Our prototype system draws on Valgrind's method of calling the pyVEX module to convert machine instructions, including intermediate expressions, intermediate expression operations, temporary variables, states, blocks, and more.

1) Expressions: The value evaluated from the intermediate expression or constant, includes memory load, register read, and arithmetic results.

2) Operations: The processing of intermediate expressions includes integer operations, floating point operations, bit operations, and so on.

3) Temporary variables: The temporary variable used by the VEX expression processing, starting with t0.

4) Statements: Intermediate statements that change the target machine's state. For example, the writing of memory and registers.

5) Blocks: A set of intermediate statements and intermediate expressions, represents a basic block of the target machine.

### C. SIMULATION EXECUTION AND RESULT VERIFICATION

Our prototype system obtains run-time information from the real execution of the target device, and performs simulation execution on the host, and performs taint mark propagation

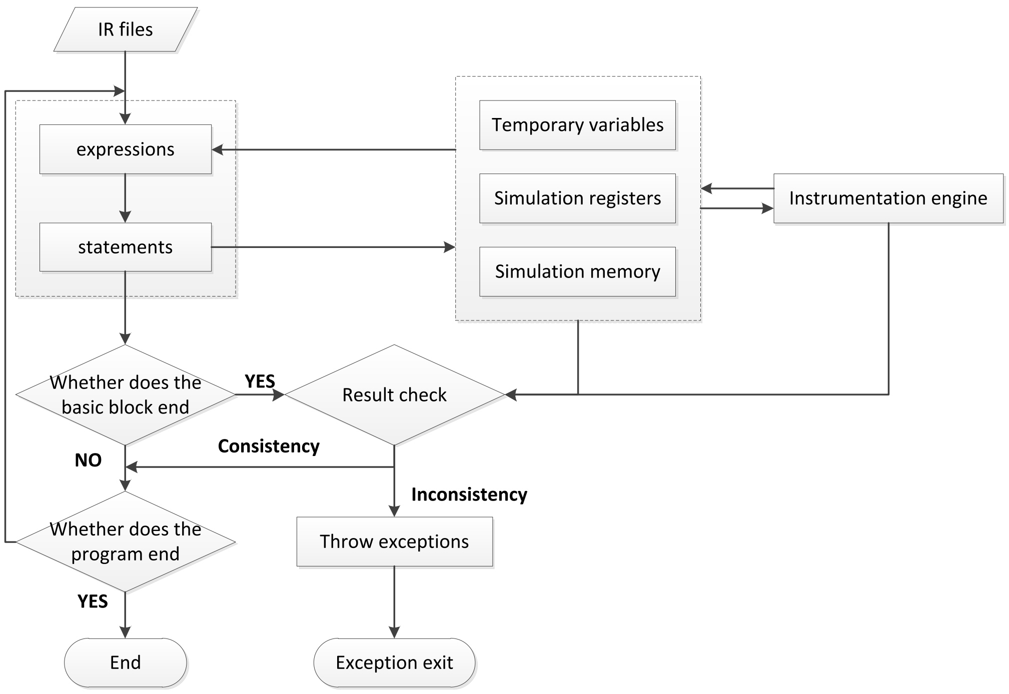


**FIGURE 4.** Simulation execution flow chart.

according to the taint propagation strategy. The specific process is shown in Figure 4.

In order to ensure the correctness of simulation executions, the calculated simulation memory's and registers' values are compared with the results obtained by the entity equipment to ensure the accuracy of the analysis, after the end of each basic block analysis.

### D. TAINT ANALYSIS

The taint source introduction is the beginning of taint analysis. In this prototype, the granularity is marked by byte, and the taint information is stored by the way of bitmap. In order to make the prototype meet the actual needs, we provide three ways to introduce taints:

1) Program execution parameters. Some programs input data through command line parameters, so we can choose to use command line parameters as taint sources.

2) Library functions. Functions such as read, scanf, getchar, etc., can enter data into the program by reading files, so it is possible to mark the input information as taint sources by recognizing the input of relevant library functions.

3) Manually marking. In some cases,, the above two methods may not be used for taint marking, thus providing a manual marking method for taint marking data is necessary.

According to the characteristics of intermediate representation statements, it can be divided into four categories:

1) single-data dependencies, such as constants, temporary variables, registers, memory values, etc., whose taint labels can be directly propagated to the target data.

2) Multi-data dependencies, mainly including multi-parameter expressions such as bit operations and arithmetic operations.

3) data-extension dependencies, mainly some data-length-extension expressions, including 0-extension and sign-extension.

4) special dependencies, refers to the taint propagation of ITE expressions. In addition, since the libc library function is called in the simulation taint analysis, the taint propagation strategy should also include the processing of the libc library function.

The taint detection section mainly checks whether the taint propagation process violates security policies, and warns and records relevant information if the security policies are violated. The taint detection of our prototype mainly checks two aspects. One is to check whether the privacy is leaked. For some programs that have sensitive information (such as password, personal data), mark private data as taint data to check whether there is a leak of privacy during the running process. The second is to check whether the control flow is hijacked, that is, when the taint data modifies the return addresses, function pointers, etc. to control the transfer related data, the program is considered to have a vulnerability.

Algorithm 2 shows the algorithm details of the function simulation_execution(). The instruction $\boldsymbol{I}$, taint sources set $\boldsymbol{S}$ and taint convergence points set $\boldsymbol{C}$ are the inputs of the function. The set(value, label) is initialized to ø in Line 1. The value and label of results are calculated by the function of execution and propagation, which are used to update set(value, label) in Line 5. Then the function will detected

**Algorithm 2** Algorithm of the Function Simulation_Execution

```
Parameteres: Instruction $I$, Taint sources set $S$,
    Taint convergence points set $C$
Results: $set(value, label)$ or $taint\_information(C)$
 1: $set(value, label) = \emptyset$
 2: while do
 3:     $value = execution(I, S)$
 4:     $label = propagation(I, S)$
 5:     $set(value, label) = update(value, label)$
 6:     if $taint\_detection(C)$ ==true then
 7:         return $taint\_information(C)$
 8:     else
 9:         return $set(value, label)$
10:     end if
11: end while
```

whether taint convergence points are tainted. If the result is true, the information about the taint detection will be returned in Line 7.

## VII. CASE STUDY

### A. TEST ENVIRONMENT

This subsection illustrates the effectiveness of the system design and the operation as expected through an example operation. Before displaying the results, we first introduce the experimental environment shown in Table 1. The host is a personal computer and the target is an embedded device. The target is connected to the host through a serial port.

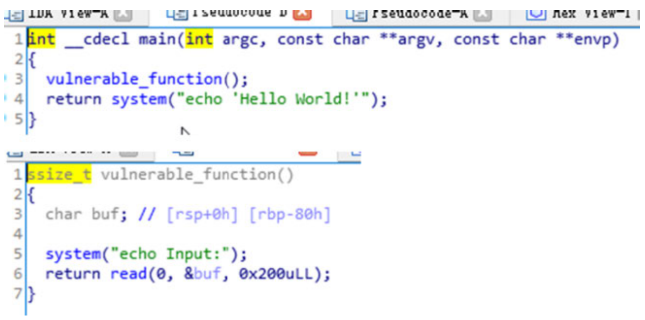
**TABLE 1.** Experimental environment.

| | Host | Target |
|---|---|---|
| CPU | CORE i7 | Pentium IV 1.6GHz |
| Memory | 2GB | 256MB |
| OS | Ubuntu16.04 | X86 |

### B. TEST CASE

This subsection examines the feasibility of a prototype system by performing taint analysis to a program with stack overflow vulnerability. The target program of this experiment is from a PWN competition named [XMAN] level2 in the Jarvis OJ CTF training platform (website is https://www.jarvisoj.com/challenges). The program's decompiled result in IDA pro is shown in Figure 5.

It can be seen from Figure 5 that the function vulnerable_function() reads 0 × 200 bytes of data into the buf, which exceeds the size of the buf space, causing the stack to overflow. In the process of taint analysis using our system, the prototype prompts that the eip register is marked, and there may be a control flow hijacking vulnerability, as shown in Figure 6.



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  return system("echo 'Hello World!'");
}
```

```
ssize_t vulnerable_function()
{
  char buf; // [rsp+0h] [rbp-80h]

  system("echo Input:");
  return read(0, &buf, 0x200uLL);
}
```

**FIGURE 5.** Decompile result.

When manually analyzing the log, we can find that the system marked the function read() as taint source, and record the taint propagation process. Finally, when the eip register is marked with taint labels, a warning is issued. After manual analysis, we can find a vulnerability is triggered at 0 × 40061f of the program, as shown in Figure 7.

Similarly, our system can detect heap overflow and format string vulnerability by the target program with ***taint*** label.

The above case confirms the feasibility of our method in practical applications. But we are also concerned about another question that cannot be answered in this section: Does our approach have a negative impact on the effectiveness and efficiency of embedded systems? This question will be answered in the next section.

## VIII. EVALUATION

### A. EXPERIMENTS SETUP

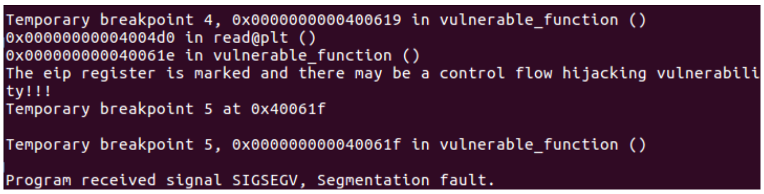
In this section, we use the prototype to perform taint analysis on several programs in the test standard set and use time consumption to evaluate the impact of the prototype system on the efficiency of the target program. Because we need code checking and reverse engineering to analyze the reasons for the comparison of results, the test program is relatively small. Please note that all programs tested are from the bin64 toolset.

### B. SYSTEM PERFORMANCE ANALYSIS

In this experiment, we analyzed the average time of a taint analysis, and the results are shown in Table 2. In this section, we will answer the question of the negative impact our analysis methods have on the efficiency of embedded systems.

Dynamic instrumentation of the prototype system is in units of basic blocks. It can be seen from Table 2 that the more breakpoints of the analyzed program, the greater the time overhead of dynamic taint analysis. The reason for this phenomenon is that frequent instrumentation reduces the efficiency of taint analysis. As can be seen from the last line of Table 2, an average of 98511 breakpoints are insert into a program, and the average analysis time is 23m1.550s. It takes about 0.014s to analyze a basic block.

In addition, the prototype system performs taint analysis at the Intermediate representation statements level,

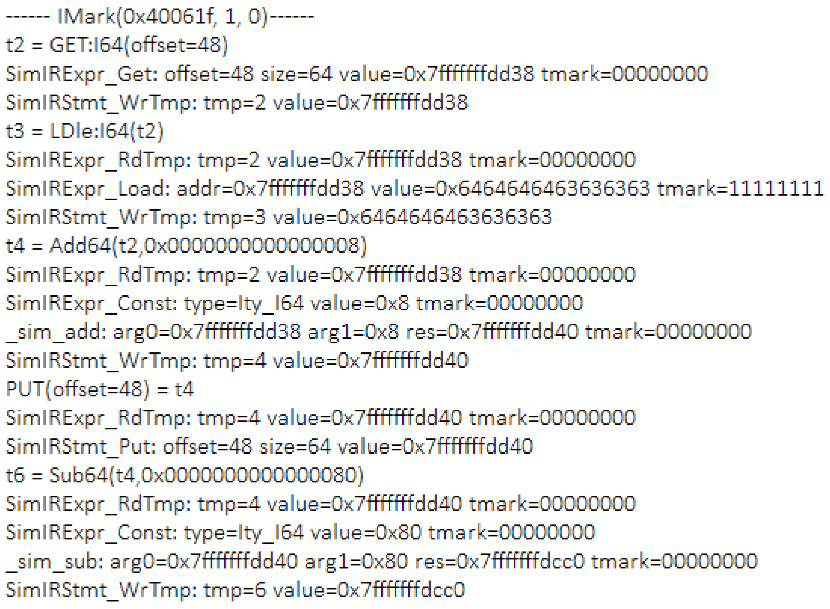


```
Temporary breakpoint 4, 0x0000000000400619 in vulnerable_function ()
0x00000000004004d0 in read@plt ()
0x000000000040061e in vulnerable_function ()
The eip register is marked and there may be a control flow hijacking vulnerabili
ty!!!
Temporary breakpoint 5 at 0x40061f

Temporary breakpoint 5, 0x000000000040061f in vulnerable_function ()

Program received signal SIGSEGV, Segmentation fault.
```

**FIGURE 6.** Taint analysis tips.



```
------ IMark(0x40061f, 1, 0)------
t2 = GET:I64(offset=48)
SimIRExpr_Get: offset=48 size=64 value=0x7fffffffdd38 tmark=00000000
SimIRStmt_WrTmp: tmp=2 value=0x7fffffffdd38
t3 = LDle:I64(t2)
SimIRExpr_RdTmp: tmp=2 value=0x7fffffffdd38 tmark=00000000
SimIRExpr_Load: addr=0x7fffffffdd38 value=0x6464646463636363 tmark=11111111
SimIRStmt_WrTmp: tmp=3 value=0x6464646463636363
t4 = Add64(t2,0x0000000000000008)
SimIRExpr_RdTmp: tmp=2 value=0x7fffffffdd38 tmark=00000000
SimIRExpr_Const: type=Ity_I64 value=0x8 tmark=00000000
_sim_add: arg0=0x7fffffffdd38 arg1=0x8 res=0x7fffffffdd40 tmark=00000000
SimIRStmt_WrTmp: tmp=4 value=0x7fffffffdd40
PUT(offset=48) = t4
SimIRExpr_RdTmp: tmp=4 value=0x7fffffffdd40 tmark=00000000
SimIRStmt_Put: offset=48 size=64 value=0x7fffffffdd40
t6 = Sub64(t4,0x0000000000000080)
SimIRExpr_RdTmp: tmp=4 value=0x7fffffffdd40 tmark=00000000
SimIRExpr_Const: type=Ity_I64 value=0x80 tmark=00000000
_sim_sub: arg0=0x7fffffffdd40 arg1=0x80 res=0x7fffffffdcc0 tmark=00000000
SimIRStmt_WrTmp: tmp=6 value=0x7fffffffdcc0
```

**FIGURE 7.** Taint analysis log.

**TABLE 2.** Time consumption for one run of taint analysis.

| Programs | Number of breakpoints | Original running time | Current running time | PANDA's running time |
|---|---|---|---|---|
| ./size ar | 12289 | 0.533s | 2m42.789s | 3m34.115s |
| ./readelf -v | 43 | 0.001s | 7.098s | 8.817s |
| ./objdump -V | 168 | 0.003s | 4.107s | 5.152s |
| ./nm ar | 420236 | 0.019s | 74m3.760s | 78m37.285s |
| ./strings test.txt | 358 | 0.005s | 10.100s | 12.808s |
| ./ar -V | 186 | 0.001s | 3.545s | 4.600s |
| ./strip a.out | 267 | 0.003s | 3.900s | 4.705s |
| ./c++filt _Z5printRKSs | 643 | 0.003s | 15.551s | 19.882s |
| ./addr2line -e main 080483fd | 202193 | 0.006s | 53m53.344s | 57m52.228s |
| ./ld -o output /lib/crt0.o main.o -lc | 348727 | 0.004s | 98m51.302s | 103m42.693s |
| Average | 98511 | 0.058s | 23m1.550s | 24m28.229s |

and translates each instruction into multiple Intermediate Representation Statements during the execution process. It can be seen in Figure 7. Therefore, the analysis workload is larger than the taint analysis at the instruction level, it's another reason which also causes the system analysis efficiency to decrease.

In order to more realistically evaluate the performance of our prototype, we added a comparative evaluation test of the system performance overhead of PANDA for taint analysis of test sets (the bin64 toolset).

PANDA is a dynamic analysis platform based on QEMU virtual machine with functions such as record, replay and analysis. In this experiment, the total time for PANDA to perform taint analysis on the test set is the sum of record, replay, and analysis. According to our tests, the time recorded in each test is different due to various aspects such as system operation. Therefore, we calculated the average time cost of 5 tests on the test set by PANDA and compared them. From the comparison results, the total time of our prototype is reduced by about 5.9%.

## IX. DISCUSSION

Although this paper proposes a new embedded device taint analysis method, which is an alternative to the existing method. But we do not think it can solve all the problems related to the subject of this research. In this section we will discuss the limitations of this approach, as well as possible solutions for future research.

The first limitation is ***the decline in the efficiency of programs***, which stems from the design of our approach. Specifically, we take the instrumentation method to get programs' run-time information, and this will greatly reduce the running speed of the embedded device firmware. Therefore, our system may have problems when testing time sensitive program. In order to solve this problem, we need to design a more efficient instrumentation method. Another solution that we are doing is combining taint analysis with fuzzing. Firstly, we get a collection of test cases that crash the firmware program through the fuzzing test, and then research the reason of the crash caused by the specific test case. So, we do not have to do a lot of analysis work except for a potential vulnerability. This can reduce the impact of low efficiency.

The second limitation is ***the accuracy of taint analysis***. The problem is caused by the implementation of the prototype. Since the firmware program is simulation execution on hosts, the result may be different from the real execution result of the firmware program on target devices, so the ***check*** module is required to ensure the consistency of the two results. At present, we adopt a conservative strategy to deal with the inconsistency of the result verification. That is, if the results are different, the simulation execution is performed again on the host. The implementation of the ***check*** module and the use of conservative strategies can affect the efficiency of the embedded device firmware.

The last limitation is ***the portability of our framework***. Due to the difference between the architectures and platforms, there are still a lot of problems to migrate the prototype, such as pointer, local call, string operation, register alias, and floating point operation. Detailed information about these challenges and related solutions can be found in other's works [33].

## X. CONCLUSION

In this work, we propose a taint analysis method based on entity equipment. The core idea of this method is to perform the taint analysis by simulation execution on the host and real execution on the target device. The host uses the running information of firmware program obtained by GDB debugging to construct the simulation execution environment, and simulates the execution of intermediate statements at the level of intermediate representation language to get the simulation state of program operation and the state of taint propagation. The target device actually runs the firmware program and waits for the mainframe to verify the status at the set breakpoint. After the verification is passed, the firmware program continues to run downward. Our approach overcomes the strict limitations of embedded device hardware and software. Through the embedded system cross- debugging feature, the host can communicate with the target device. This method overcomes the shortcomings of performing a large number of manual tasks while constructing a virtual environment for taint analysis.

We implemented a prototype system on gdb and practiced the feasibility of the method. Experiments show that the prototype can perform taint analysis on embedded device firmware, and can detect stack overflow, heap overflow and other types of vulnerabilities. As far as we know, this is the first system to do this work.

## REFERENCES

[1] L. Wang, L. I. Feng, L. I. Lian, and X. B. Feng, "Principle and practice of taint analysis," *J. Softw.*, 2017.

[2] J. Kim, T. Kim, and E. G. Im, "Survey of dynamic taint analysis," in *Proc. 4th IEEE Int. Conf. Netw. Infrastruct. Digit. Content*, Sep. 2014, pp. 269–272.

[3] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Dexterjs: Robust testing platform for DOM-based XSS vulnerabilities," in *Proc. Joint Meeting Found. Softw. Eng.*, 2015.

[4] P. Shuanghe, L. Peiyao, and H. Jing, "A Python security analysis framework in integrity verification and vulnerability detection," *J. Natural Sci. Wuhan Univ. (English Ed.)*, vol. 24, no. 2, pp. 141–148, 2019.

[5] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: Detecting the taint-style vulnerability in embedded device firmware," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 430–441.

[6] C. Feng and X. Zhang, "A static taint detection method for stack overflow vulnerabilities in binaries," in *Proc. 4th Int. Conf. Inf. Sci. Control Eng. (ICISCE)*, Jul. 2017, pp. 110–114.

[7] S. Cong, P. Feng, L. Teng, J. Ma, S. Cong, P. Feng, L. Teng, J. Ma, S. Cong, and P. Feng, "Data-oriented instrumentation against information leakages of Android applications," in *Proc. Comput. Softw. Appl. Conf.*, 2017.

[8] D. Zou, J. Zhao, W. Li, Y. Wu, W. Qiang, H. Jin, Y. Wu, and Y. Yang, "A multigranularity forensics and analysis method on privacy leakage in cloud environment," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 1484–1494, Apr. 2019.

[9] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices," *IEEE Trans. Depend. Sec. Comput.*, to be published.

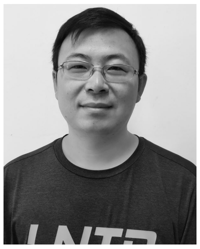
[10] D. Rathi and R. Jindal, "Droidmark: A tool for android malware detection using taint analysis and bayesian network," *Int. J. Recent Innov. Trends Comput. Commun.*, vol. 6, no. 5, pp. 71–76, 2018.

[11] D. D. Chen, M. Egele, M. K. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proc. NDSS*, 2016, pp. 1–16.

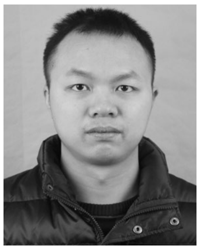
[12] Y. Ren, Y. Zhang, and C. Ai, "Survey on taint analysis technology," *J. Comput. Appl.*, vol. 39, no. 8, pp. 2302–2309, 2019.

[13] M. K. Yoon, N. Salajegheh, C. Yin, and M. Christodorescu, "Pift: Predictive information-flow tracking," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 4, pp. 713–725, 2016.
[14] M. Backes, S. Bugiel, O. Schranz, P. Von StypRekowsky, and S. Weisgerber, "Artist: The Android runtime instrumentation and security toolkit," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroSP)*, Paris, France, 2017, pp. 481–495.
[15] I. Roy, D. E. Porter, M. D. Bond, K. S. Mckinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 63–74, 2009.
[16] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity JVMS," *ACM Sigplan Notices*, vol. 49, no. 10, pp. 83–101, 2014.
[17] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. Mcdaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, 2014.
[18] Y. Zhang, T. Tian, L. Yue, and J. Xue, "Ripple: Reflection analysis for Android apps in incomplete information environments," *Softw. Pract. Exper.*, vol. 48, no. 1, pp. 1419–1437, 2018.
[19] H. Cai and J. Jenkins, "Leveraging historical versions of Android apps for efficient and precise taint analysis," in *Proc. ACM 15th Int. Conf.*, May 2018, pp. 265–269.
[20] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information flow control in Webkit's javascript bytecode," in *Principles of Security and Trust* (Lecture Notes in Computer Science), vol. 8414, M. Abadi and S. Kremer, Eds. Berlin, Germany: Springer, 2014, pp. 159–178.
[21] G. Chinis, P. Pratikakis, S. Ioannidis, and E. Athanasopoulos, "Practical information flow for legacy Web applications," in *Proc. Workshop Implement.*, 2013.
[22] R. Karim, F. Tip, A. Sochurkova, and K. Sen, "Platform-independent dynamic taint analysis for javascript," *IEEE Trans. Softw. Eng.*, to be published.
[23] F. Y. Tang, C. Feng, and C. J. Tang, "Binary vulnerability exploitability analysis," in *Proc. Int. Conf. Inf. Syst. Artif. Intell.*, 2017.
[24] S. Yovine and G. Winniczuk, "Checkdroid: A tool for automated detection of bad practices in Android applications using taint analysis," in *Proc. IEEE/ACM 4th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, May 2017, pp. 175–176.
[25] K. C. Sang, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, 2012, pp. 380–394.
[26] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Proc. Int. Conf. Comput. Aided Verification*, 2011.
[27] T. Chen, X.-S. Zhang, X.-L. Ji, C. Zhu, Y. Bai, and Y. Wu, "Test generation for embedded executables via concolic execution in a real environment," *IEEE Trans. Rel.*, vol. 64, no. 1, pp. 284–296, Mar. 2015.
[28] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. San Diego, CA, USA: ÉTATS-UNIS, Feb. 2014.
[29] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.)*, vol. 18, Feb. 2018.
[30] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. Checker: A soundy analysis for linux kernel drivers," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 1007–1024.
[31] J. Lin, L. Jiang, Y. Wang, and W. Dong, "A value set analysis refinement approach based on conditional merging and lazy constraint solving," *IEEE Access*, vol. 7, pp. 114593–114606, 2019.
[32] 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," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[33] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2011.

**YUZHU REN** was born in 1985. He received the B.S. degree in network engineering from the National University of Defense Technology, in 2008. He is currently pursuing the M.D. degree in computer technology with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include vulnerability exploit, the Internet of Things security, and deep learning.



**WEIYU DONG** was born in 1976. He is currently an Associate Professor and a Master Supervisor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. His main research interests include computer architecture, system virtualization, and computer security.



**JIAN LIN** was born in 1989. He received the M.S. degree in computer science and technology from the Information Engineering University, in 2016. He is currently pursuing the Ph.D. degree in cyberspace security with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His main research interests include binary program analysis, and vulnerability detection and exploit.

**XINLIANG MIAO** was born in 1994. He is currently pursuing the Ph.D. degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include computer architecture and microarchitectural side-channel attacks.