World Wide Web (2021) 24:941–954
https://doi.org/10.1007/s11280-020-00838-3

Firmware code instrumentation technology for internet of things-based services

Chen Chen[1,2] · Jinxin Ma[3] · Tao Qi[1] · Baojiang Cui[1] · Weikong Qi[4] · Zhaolei Zhang[2] · Peng Sun[2]

Received: 13 May 2020 / Revised: 14 June 2020 / Accepted: 31 August 2020 /
Published online: 26 September 2020

Abstract
With the rapid development of electronic and information technology, Internet of Things (IoT) devices have become extensively utilised in various fields. Increasing attention has been paid to the performance and security analysis of IoT-based services. Dynamic instrumentation is a common process in software analysis for acquiring runtime information. However, due to the limited software and hardware resources in IoT devices, most dynamic instrumentation tools do not support IoT-based services. In this paper, we provide an analysis tool, IoTDIT, to solve the current problem of runtime detection in IoT-based services. IoTDIT employs static analysis and *ptrace* system calls to obtain dynamic firmware information, which can aid in firmware performance analysis and security detection. We perform experiments to verify the performance and effectiveness of the proposed instrumentation tool.

Keywords IoT devices · Firmware testing · Instrumentation · Dynamic instrumentation

1 Introduction

With the rapid development of information technology and the Internet of Things (IoT), IoT devices are increasingly applied in industry and social life [20, 28]. The performance

This article belongs to the Topical Collection: *Special Issue on Intelligent Fog and Internet of Things (IoT)-Based Services*
Guest Editors: Farookh Hussain, Wenny Rahayu, and Makoto Takizawa

✉ Chen Chen
00152tenten@bupt.edu.cn

✉ Baojiang Cui
cuibj@bupt.edu.cn

1 Beijing University of Posts and Telecommunications, Beijing, China

2 Air Force Engineering University, Xi'an, China

3 China Information Technology Security Evaluation Center, Beijing, China

4 China Academy of Space Technology, Beijing, China

and security of such devices must be analysed by technical means. The traditional analysis of programs is mainly based on instrumentation tools, which include dynamic instrumentation and static instrumentation tools. Instrumentation tools are used to obtain the value of the register at the specified instructions [23], catch interesting signals [2, 13] and obtain the execution information of the basic blocks [21, 24]. Furthermore, other technologies, for example, symbolic execution [1, 11] and taint analysis [6–8, 12] , based on the instrumentation are used for more complex dynamic analysis of binaries.

Traditional static instrumentation tools [9, 14, 18, 22] insert analytical code directly into the target binary code. This method is complex and depends on the specified hardware platform (x86, SPARC etc.), operating system (UNIX, ucLinux, VxWorks, Linux, uC/OS, TinyOS etc.) and file type (PE, ELF, S-record, etc.); therefore, it is not suitable for IoT devices.

Traditional dynamic instrumentation tools [3, 15, 19] use the software interfaces provided by the operating system to analyse the context environment of a program after hijacking the execution stream. However, it is difficult for these tools to be employed IoT-based services. The main reasons are as follows: First, running these tools requires the support of operating system interfaces, such as dynamic memory allocation and thread control, which are not included in many IoT device operating systems [16]. Second, some of these tools occupy a certain amount of memory when running, which may not be satisfied for IoT devices due to hardware resources limitations.

Considering the lack of dynamic analysis technology for IoT-based services, this paper applies static analysis technology and Linux *ptrace* system call to obtain dynamic firmware information, including the execution time of functions, sample execution path, etc, at runtime. This information can be applied to aid performance analysis [29] and dynamic security detection [10, 17, 25, 27] for IoT-based services.

Our contributions are as follows.

(1) A simple and effective method is proposed for the instrumentation of IoT-based services, which provides a new approach to firmware detection.
(2) We design and implement a prototype of the instrumentation tool for IoT-based services.
(3) We compare the performance of IoTDIT with that of other instrumentation tools and verify the feasibility of the proposed instrumentation tool for IoT-based services.

The remainder of this paper is structured as follows. Section 2 introduces related work. Section 3 briefly introduces the basic knowledge of the *ptrace* system call, and Section 4 gives the design details of the prototype tool. Section 5 tests and analyses the prototype tool. Section 6 discusses the limitations and shortcomings of this tool. Finally Section 7 presents the conclusions.

2 Related works

Currently, dynamic instrumentation tools are widely used in software analysis, including Pin, DynamoRIO, Valgrind, Strata, Vulcan and DTrace.

Pin [15] is a framework for the dynamic analysis of binary code; it intercepts the entry point of the program, recompiles the instrumented code with the original instructions, generates a new code sequence and executes this sequence. When Pin encounters a branch, it regains control, finds the conditional transfer instruction, and adds the instruction to the code

sequence to continue execution. The tool supports Linux, Windows and OS X operating systems, but only on the Intel platform, so use of the tool in IoT devices is difficult.

Valgrind [19] can be used for memory debugging, memory leak detection and performance analysis; however, it does not provide a programming interface for users, and some requirements of users cannot be met.

DynamoRIO [3] executes a target application by copying the application code into a code cache, one basic block at a time. The code cache is entered via a context switch from DynamoRIO's dispatch state to that of the application. The cached code can then be executed natively, which avoids emulation overhead.

Vulcan [26] provides an instrumentation technology across different instruction sets. It reads the image of the target program from memory, transforms it into an abstract representation, modifies the code in this form, generates the instrumented code, and then redirects the program to the modified code for execution.

DTrace [4] is an integrated instrumentation tool in the Solaris operating system. It uses a high-level language to describe the operation at the stake insertion point. The tool supports the operation at the user layer and the kernel layer.

Due to the difference in the technology adopted, the performance loss and dependence on the OS interfaces of different instrumentation tools also differ. The Table 1 lists the comparison of different tools in three aspects.

Pin, DynamoRIO and Valgrind are JIT-based tools. This kind of real-time compilation has a great impact on the time cost. Vulcan and DTrace are probe based tools. In this way, although multiple jump instructions contribute to performance loss, it will be smaller than that of JIT-based tools. Different tools support different platforms. DynamoRIO, Valgrind and Vulcan support different instruction sets, while Pin only supports the Intel architecture. Additionally, these dynamic instrumentation tools need interfaces provided by operating systems, such as multi-threading interfaces and dynamic memory allocation. However, these interfaces do not exist in many IoT operating systems.

3 Ptrace system call

In order to obtain the dynamic information of firmware, we can use a debugger to start the firmware program to be tested. However, we need to control the debugger to run; that is, another process needs to interact with the debugger, which will cause three processes to interact with each other during the test process and thus, slow down the test performance.

There are many different types of operating systems for IoT devices, including Linux, ucLinux, uC/OS, and TinyOS. Although the implementation of the debugging functions in these operating systems differs, the basic principles are the same. Here, we take the Linux operating system as an example. The Linux debugger is based on a *ptrace* system call, and

Table 1 Comparison of dynamic instrumentation tools

Name of tools	Performance loss	Dependence on OS interfaces
Pin	high	heavy
DynamoRIO	high	heavy
Valgrind	high	heavy
Vulcan	moderate	moderate
DTrace	moderate	moderate

thus we can build the instrumentation tool on the *ptrace* system call interface; that is, only two processes are needed to obtain dynamic information.

ptrace is a system call for process tracking provided by the Linux system; it provides the parent process with ability to observe and control the execution of a child process, and allows the parent process to check and replace the value of a kernel image (including registers) in the child process. When a process is tracked with *ptrace*, all signals sent to the tracked child process (except SIGKILL) are forwarded to the parent process, and the child process is blocked. Then, the state of the child process is marked as TASK_TRACED by the system. When the parent process receives the signal, it can check and modify the stopped child process, and control the child process. Notably, the parent process can continue to run the child process.

The prototype *ptrace* system call in this paper is described as follows:

long ptrace(enum _ptrace_request request, pid_t pid, void addr, void data).

In this case, ptrace has four parameters:

1) ***enum _ptrace_request request:*** indicates the command to be executed by ptrace;
2) ***pid_t pid:*** indicates the process id that ptrace tracks;
3) ***void addr:*** indicates the memory address to be monitored;
4) ***void data:*** indicates the data memory address to which to read or write.

The first parameter ***request*** indicates the type of commands that system calls need to execute, including tracking processes, reading data from memory, writing data to memory, killing processes, and reading and setting registers. This parameter provides the most basic control operation of the program to be tested.

We use *ptrace* to perform instrumentation as follows. We save the instruction to be analysed and set it as a breakpoint; then, *ptrace* is used to start the program and listen to the signal sent by the target program. When the target program executes to the breakpoint, it will pause running. At this time, *ptrace* is used to insert all kinds of analysis code; then, the value of the instruction pointer register is reset to the address of the breakpoint, and *ptrace* is used to continue running the program.

Some IoT devices employ Linux or tailored Linux as the operating system. Given that *ptrace* usually exists as a basic component in these systems, it can be applied to perform instrumentation for IoT-based services.

4 System design

4.1 Overall design

This paper applies *ptrace* to implement a dynamic instrumentation tool for IoT-based services, IoTDIT, which includes 4 modules: code extraction, static analysis, instrumentation and dynamic control. The overall architecture of IoTDIT is shown in Figure 1.

First, the target firmware is analysed by the code extraction module to obtain the executable code data. IoTDIT extracts the control flow graph and function call graph from the executable code and analyses and obtains the instruction positions according to the test requirements. Then, the instructions at these locations in the original file are set as breakpoints, and the location information and instruction information for each breakpoint are saved. Last, IoTDIT monitors firmware execution and analyses the collected runtime information. In this process, the firmware binary program will be suspended when it is executed at a breakpoint. The monitor collects the runtime information at this breakpoint and then

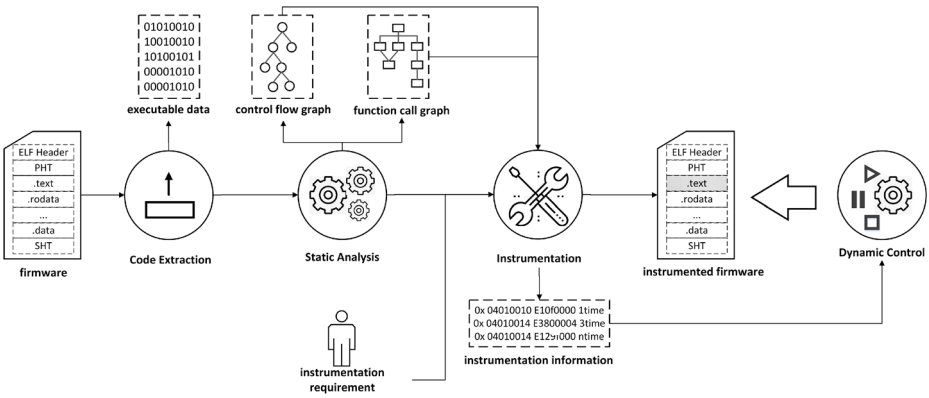

Figure 1 Overall architecture of IoTDIT

uses the original instructions saved beforehand to restore the breakpoint, so that the program can continue to run.

4.2 Code extraction

The binary code for instrumentation is saved in the firmware. Currently, many binary firmware codes are encapsulated in ELF file format. In ELF format, binary code is arranged in files according to a certain organization scheme. An ELF file consists of four parts as seen in Figure 2: ELF header, PHT (program header table), sections or segments (.text, .data, .symtab, ...), and SHT (section header table).

The ELF header describes the organization of the whole file and defines the overall attribute information of the file. The most important attributes are magic words, entry point address, start of program headers, start of segment table, length and quantity, and size of file header. The PHT describes various segments in the file and is used to tell the system how to create process images. Segments describe ELF files from the perspective of running, and a segment contains several sections. Sections describe ELF files from the perspective of linking, and the SHT contains the attribute information of each section in the file. That is, we can ignore the PHT to process this file in the linking phase and can ignore the SHT in the running phase.

To extract binary code, we need to analyse the ELF file according to its format rules. This process is shown in Figure 2. IoTDIT obtains the offset of the section header table from the ELF header, and then obtains the offset and length of sections from the section header table. The executable code is stored in the .text section, which is the target for instrumentation. In the stage of static analysis, we should construct the control flow target by analysing the executable code. Additionally, the function offset and function name information is stored in the .symtab section, which is used to analyse the function call graph in the next stage.

4.3 Static analysis

The aim of static analysis is to build a control flow graph and function call graph for firmware code. This information is used to obtain the code execution path, function execution sequence, function execution time, etc. IoTDIT disassembles the extracted code

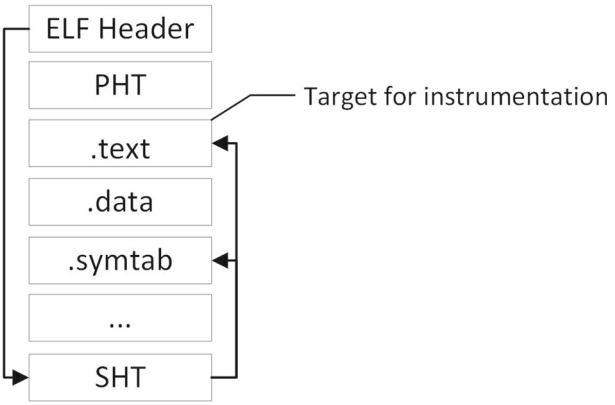

Figure 2 The process of code extraction

to reduce the difficulty of analysing and operating the machine code. The disassembly operation in IoTDIT is mainly achieved by the interfaces provided by Capstone [5].

The control flow graph is the directed graph $\boldsymbol{D} = \langle B, Trans \rangle$, which is composed of the node set $\boldsymbol{B}$ and directed edge set $\boldsymbol{Trans}$. The node set $\boldsymbol{B}$ is composed of all basic blocks $\boldsymbol{b}_i$ in the program: $\boldsymbol{B} = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots \boldsymbol{b}_n\}$. The directed edge set a is composed of transfer relations among all code basic blocks: $\boldsymbol{Trans} = \{\boldsymbol{t}_1, \boldsymbol{t}_2, \cdots \boldsymbol{t}_n | t_i = (b_k, b_j), k \neq j\}$. Therefore, the process of building a control flow graph is the process of generating a basic block set $\boldsymbol{B}$ and transfer relation set of basic code $\boldsymbol{Trans}$. The basic block refers to the sequence of statements executed in sequence. There is only one entry and one exit. The entry is the first statement, and the exit is the last statement. A basic block only enters from its entry and exits from its exit. The identification of the basic block depends on the transfer instruction; the transfer address of the instruction is the starting position of a basic block, and the address of the instruction is the end position of a basic block. These basic blocks $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots \boldsymbol{b}_i\}$ can be extracted by one-time traversal in the firmware and form the set $\boldsymbol{B}$, and then the transfer relationship $(\boldsymbol{b}_i, \boldsymbol{b}_j)$ between the basic blocks is established by the transfer instruction. All transfer relationships form the set $\boldsymbol{Trans}$ to complete the construction of the control flow graph.

The function call graph is also the directed graph $\boldsymbol{H} = \langle F, R \rangle$, which describes the functions and call relationship between two functions in firmware code. In graph $\boldsymbol{H}$, $\boldsymbol{F}$ represents the set of all functions in the firmware, and $\boldsymbol{R}$ consists of all function call relationships $(\boldsymbol{f}_i, \boldsymbol{f}_j)$. Therefore, the process of building a function call graph is the process of generating the set $\boldsymbol{F}$ and set $\boldsymbol{R}$. The function $\boldsymbol{f}_i$ in set $\boldsymbol{F}$ is obtained from the .symtab section. In this section, the function name, address, length and other information are accessed to generate set $\boldsymbol{F}$, which is used to traverse the function call instructions in each function to establish the relationship between the current function and the called function $(\boldsymbol{f}_i, \boldsymbol{f}_j)$. All of the function call relations are then collected to generate $\boldsymbol{R}$.

4.4 Instrumentation

Instrumentation is based on the actual test requirements to determine where to insert the analysis points in a program. IoTDIT provides an analytical position selection scheme at different scales, including the function level, basic block level and instruction level.

Function-level position analysis is performed with the list of functions and location information acquired in the static analysis stage. Basic block-level position analysis regards the starting position of the node in the control flow diagram as the analytical position. Instruction-level analysis regards a single instruction position as the analysis position by querying all the instructions of interest in the code.

Breakpoint insertion is performed after the analysis points are determined, and the instructions at the analysis points are changed to breakpoint instructions. In this process, IoTDIT saves the overwritten instructions and their addresses, which will be stored for subsequent breakpoint recovery.

4.5 Dynamic control

The dynamic control module collects and analyses the required information during the dynamic execution of the program being test, including steps of performing breakpoint control and runtime analysis.

4.5.1 Breakpoint control

Dynamic control in IoTDIT is based on the *ptrace* system call. The monitor starts the program to be tested in debugging mode. The program is suspended at the breakpoints that are written in the program. At this time, the monitor will take over the processing flow and analyse the collected dynamic information according to the user-defined operation. After processing the user-defined operation, the monitor restores the breakpoint instructions by using the breakpoint information saved in the static analysis stage.

According to the different requirements of instrumentation, the methods of manipulating breakpoints are also different, and mainly include two types: performing instrumentation once for one point, and performing instrumentation many times for one point. In the first case, after the breakpoint is triggered, the restored instruction is recovered to the breakpoint instruction, and the operation continues to execute. In the second case, first, the breakpoint instruction is recovered. Second, a single step is executed, and the recovered instruction is changed to a breakpoint instruction again so that it can be paused again at the next time step.

4.5.2 Runtime analysis

IoTDIT performs different analysis operations according to the test requirements each time at breakpoints; it records the information of executions for three levels: function level, basic-block level and instruction level.

In the function layer, IoTDIT can obtain function execution times, function execution sets, and function execution paths. In obtaining the executed times of a function, the function address obtained in the static analysis phase is associated with a variable; when the firmware is executed to the function breakpoint, the variable is added with 1, and after the execution is complete, the value of this variable is the number of times the function has been executed. In the process of obtaining the function execution set, all concerned functions are associated with *Boolean* variables; when the firmware is executed to the function breakpoint, the variable is set to *true*, and after the execution is completed, functions related to the variables with the value *true* constitute function execution sets. When collecting the execution path of a function, the function name will be added to a global queue every time the function breakpoint is triggered. After the execution is completed, all functions in the queue

will form a function execution path according to their order. The functions implemented in this layer are usually used for the comprehensiveness of the firmware function test.

In the basic-block layer, IoTDIT can analyse the basic-block set and the basic-block path executed in the execution. Its implementation is similar to the principle of obtaining the execution function set and the function execution path, but the unit analysed each time is the basic block. In this layer, we usually do not analyse the number of basic-block executions because there is little demand for the number of basic-block executions. The functions implemented in this layer can be used to guide fuzzing for an efficient security test.

In the instruction layer, IoTDIT can analyse the context environment information at a certain instruction location. When the program runs to this location, it uses *ptrace* to acquire all the register information and memory information, which can help the tester to analyse some key variable values in the running application. In addition, *ptrace* can be utilised to modify these values to observe the runtime changes of firmware, which can help to locate and analyse problems.

In addition, to obtain the collected analysis information from IoT-based services, it is necessary to provide a mechanism for transmitting real-time analysis data. According to the test requirements, the user manually determines the instruction addresses for startup analysis and end analysis. Every time firmware is executed involving the instruction at the startup analysis address, it resets all information, such as the collected path information, the number of execution times and the time of one execution. Each time the instruction at the end analysis address is involved in execution, the collected information is sent to the test client via the communication interface provided by the IoT device.

5 Implementation and testing

The construction of IoTDIT is based on *ptrace* system call and disassembly framework. IoTDIT uses a disassembly engine to analyse the instructions and determine the instrumented position, and then uses ptrace to perform instrumentation at these locations. IoTDIT controls the triggering of the instrumented points and the context analysis at runtime according to the actual needs. The analysis results are returned via the hardware communication interface. We employ C++ programming language to implement IoTDIT. The software framework and hardware environment that we depend on in the process of building the tool are shown in Table 2. IoTDIT is implemented in four classes, the function interfaces of which are listed in Appendix.

The function of disassembly depends on Capstone, and the compiler of the program is g++. We generate binary files of the target platform by cross-compilation. The target platform includes the desktop platform and IoT platform. The desktop platform is used to carry out a performance comparison test with other instrumentation tools, and the IoT platform is used to test the availability of IoTDIT in actual IoT equipment. The process of testing using IoTDIT is described as follows: put IoTDIT and the binary to be tested in one hardware platform, run the binary and attach IoTDIT to it. IoTDIT will perform instrumentation according to the test requirements and analyse the runtime information on the instrumented points.

To verify the efficiency and effectiveness of IoTDIT, we designed two groups of experiments. The first group tested the efficiency of IoTDIT by a specially designed program, and the second group tested the availability of IoTDIT by employing a real home wireless router.

Table 2 Experimental parameters

Environment type	Environment item	Content
	CPU	i5-7267U CPU @ 3.10GHz
Desktop platform	RAM	1GB
	Operating system	Ubuntu 16.04.5 LTS
Software	Compiler	g++ 5.4.0
	Disassembly framework	Capstone 3.0.5
IoT platform	Wireless rooter	AC1750

5.1 Instrumentation efficiency test

To verify the efficiency and availability of IoTDIT, we tested it in a desktop system. Therefore, other tools could not run on the IoT devices. We compared the runtime wastage of IoTDIT to the wastage for the three most commonly employed instrumentation tools in desktop systems: Pin [15], Valgrind [19] and DynamoRIO [3].

From the perspective of application, the instrumentation on firmware in IoT devices can be divided into three types. The first is single-point instrumentation with multiple times, which is used to determine whether the function is abnormal by collecting the execution times of a module in firmware. The second is multi-point instrumentation with one time, which is often be used to analyse test comprehensiveness through instrumentation on functions in firmware. The third is multi-point instrumentation with multiple times; it can be used to guide fuzzing to find more vulnerabilities in firmware by collecting execution path information. Additionally, more complex security detection tools can be built on the instrumentation tools.

The first group of experiments is performed to accurately analyse the performance impact by the instrumentation tools, including the impact of different numbers of instrumentation times and different numbers of instrumentation points on the running time of the instrumented program. The basic structure of the program to be tested is a loop of m times in which n if-statements are placed. The values of m and n can be set according to the needs of the test. Moreover, m represents the execution times of the instrumentation point, and n represents the number of instrumented points. To test the impact of instrumentation on the program more clearly, the execution body of if-statement contains only one statement to reduce the running time of the program itself.

We conducted three groups of experiments on the three discussed types and analysed the effects of the four tools on the program execution efficiency. First, we set the values of m and n according to the requirements of each experiment. Second, IoTDIT is applied to perform instrumentation at the target location, and the instrumented binary is run. Finally, the running time is collected for comparative analysis. The experimental parameters are listed in Table 2.

The first experiment was used to analyse the effect of single-point instrumentation with multiple times on the runtime efficiency of the program. We repeatedly performed instrumentation on a specified code block address and collected the time consumption information of program execution at different times of instrumentation. We recorded the time consumed in instrumentation at one point repeated 10 times, 20 times, 30 times, 40 times, 50 times, 60 times, 70 times, 80 times, 90 times and 100 times. The results are shown in Figure 3a.

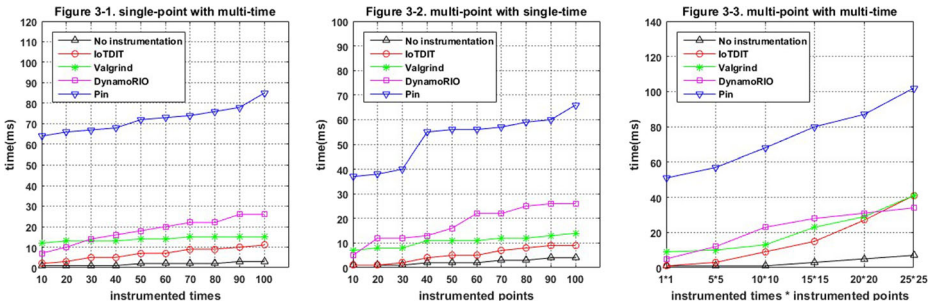

Figure 3 Time consumption of instrumentation using different tools

The second experiment analysed the effect of multi-point instrumentation with one time on the runtime efficiency of the program. We recorded the time consumed by the program after instrumenting several points in an execution path. The numbers of points in the instrumentation were 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100. The results are shown in Figure 3b.

The third experiment analysed the effect of multi-point instrumentation with multiple times on the runtime efficiency of the program. Experiments were conducted to collect the time consumption information of instrumentation at multiple points on a single path at multiple times. The results are shown in Figure 3c.

As shown in Figure 3a and b, among the four types of instrumentation tools, IoTDIT consumes the least time and Pin consumes the most time. This result is due to the simple structure design of IoTDIT; notably, there are no complex interactions between the dynamic instrumentation tools and software to be tested. The results in Figure 3a and b indicate that the consumption of time by single-time multi-point instrumentation and multiple-time single-point instrumentation is comparatively equivalent for all tools except Pin, and thus the effect of these two types of instrumentation can be treated as approximately equal. The time consumption of single-point instrumentation and single-time instrumentation is similar.

The time consumption values of IoTDIT, Valgrind and DynamoRIO in Figure 3c are similar. Based on the conclusions in Figure 3a and b, the result for the 20*20 scenario in Figure 3c is equivalent to instrumentation on 400 points. At less than 100 points, we observe that IoTDIT yields the best effect. However, when the number of instrumentation points increases gradually, the advantage of IoTDIT decreases gradually. As the firmware running in the IoT system is relatively simple and relatively few instrumentation points are needed, relatively complex software, such as *httpd*, triggers a maximum of 100 instrumentation points when collecting executed functions. Therefore, IoTDIT is well-suited for IoT systems in which the firmware is relatively small in scale.

5.2 Instrumentation effectiveness test

To verify the availability and performance of the instrumentation tool in the actual device, we have carried out three different types of instrumentation operations on *httpd* (embedded Web server) in the ac1750 wireless router and calculated the performance loss caused by the instrumentation. The calculation formula of the performance loss is shown in formula (1)

$$l = \frac{t_i - t_u}{t_u} \tag{1}$$

l: performance loss;
t_u: time from sending request to obtaining the response of un-instrumented $httpd$;
t_i: time from sending request to obtaining the response of instrumented $httpd$

In the first experiment, we tested how many times the function f_read_string was executed when visiting the login page. We instrument the point before the module, run the $httpd$ and collect the time from sending the request to obtaining the response 10 times. This experiment belongs to single-point instrumentation with multiple times. Moreover, this type of experiment is commonly performed to analyse whether the logic is normal by the number of times the function is executed.

In the second experiment, we tested which functions in $httpd$ are executed when logging in. We instrument all the functions in $httpd$, run the $httpd$ and collect the time from sending the request to obtaining the response 10 times. This experiment belongs to multiple points of instrumentation with one time. This type of experiment is commonly performed to analyse which module is not executed or to assess the comprehensiveness.

In the third experiment, we tested the executed path in the function f_read_string in $httpd$ when visiting the login page. We instrument all the basic blocks in this function, run the $httpd$ and collect the time from sending the request to obtaining the response 10 times. This experiment belongs to multiple points of instrumentation with multiple times. This type of experiment is commonly performed to guide the fuzzing tool to improve the path coverage.

The results of the experiments are shown in Figure 4. In the first experiment, the function of counting executed times was executed with a performance loss of $(-0.3)-3.4$. If the performance loss value is less than zero, the access time after instrumentation is less than that without instrumentation, which shows that the access time is unstable, and the impact caused by instrumentation is greater than that caused by the instability in access. Thus, the impact of instrumentation in this experiment is small.

In the second experiment, the function of collecting executed functions was executed with a performance loss of 3.2–8.2. It can be seen that the performance loss caused by instrumentation is relatively obvious, which is mainly due to the breakpoints inserted in each function position in the experiment. A large number of breakpoints are triggered during the execution of the instrumented program, which causes performance loss.

In the third experiment, the function of collecting executed paths was executed with a performance loss of $(-0.6)-1.4$. It can be seen that the impact of instrumentation on

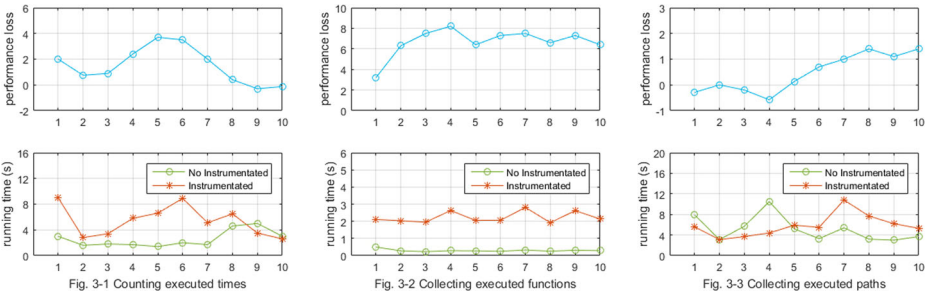

Figure 4 Performance loss of instrumentation in three scenarios

performance is not very stable, which is similar to the first experiment. For IoT-based services that do not consider latency, this kind of performance loss, below 10, can satisfy the requirements of instrumentation.

6 Limitations and shortcomings

The use of IoTDIT requires some preconditions; for example, IoTDIT needs to analyse the firmware that is being tested before it is run, but it is difficult to obtain firmware in some devices. In addition, IoTDIT needs to work in the target IoT device in the process of dynamic analysis. However, some devices do not have the operational interface to store external programs in the device. There are some techniques and methods for acquiring firmware and adding it to programs, but they are beyond the scope of this paper.

IoTDIT is built on the basis of the *ptrace* system call. To simplify the operating system in real IoT devices, some devices have a tailored system call. In these cases, IoTDIT will not be able to run. IoTDIT currently supports only 32-bit x86/ARM architecture and ELF files, but it can easily be extended to other types of file formats on embedded device and sensor platforms.

7 Conclusions

Due to the limitations of software and hardware resources in IoT devices, traditional dynamic instrumentation tools cannot be employed for analysing IoT-based services, which causes low efficiency of function and security detection for the firmware. In this paper, we design and implement a light-weight dynamic instrumentation tool, IoTDIT, which performs the instrumentation that is only based on the *ptrace* system call. It extracts the control flow chart, obtains the analytical instruction location through static analysis, and then instruments firmware and performs the dynamic analysis at the instrumented points. By two groups of experiments, we verify that IoTDIT is more suitable for IoT devices in which the firmware is relatively small in scale than other dynamic instrumentation tools, and verify the effectiveness of IoTDIT for an actual wireless router.

Funding This research was funded by Jinxin Ma of the National Natural Science Foundation of China, grant number 61872386.

Compliance with Ethical Standards

Conflict of interests The authors declare that they have no conflict of interest.

Appendix

Table 3 Core class and function interfaces of IoTDIT

Class	Functions	Description
ElfParse	IsElfFile	determine whether the target is an ELF file
	ParseElfHead	obtain the data in ELF head
	ParseElfSectionHead	obtain the data of SHT
	SetStringTable	obtain the data of string table section
	SectionHeadOfIndex	get the section header by index
	SectionHeadOfName	get the section header by name
	SectionDataOfName	get the section data by name
	SectionLengthOfName	get the section length by name
	ReplaceSection	replace the data of the section
	WriteFile	rewrite the data from memory to file
	OutPutElfHead	print out the data of ELF Header
Disassembler	SizeOfDisasm	get size of the instruction
	AddrOfDisasm	get address of the instruction
	LengthOfOpString	get the length of operation string
	LengthOfMnemonic	get the length of mnemonic
	Disasm	disassemble the instruction
	ParseBasicBlocks	get the basic blocks in the code
	CreateCFG	build control flow graph
Stub	SetINT3At	set the instrumentation point at specific location
	IsJumpInstruction	determine whether the instruction is jump instruction
	GetStubSectionSize	get the size of instrumented section
	GetStubSection	get the data of instrumented section
Controller	InitStubInfo	initialize the data for instrumentation
	ResumeStub	restore code to the status before instrumentation
	MonitorTarget	distribute tasks according to program signals

References

1. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1083–1094 (2014)
2. Babić, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 12–22 (2011)
3. Bruening, D.: DynamoRIO: dynamic instrumentation tool platform. http://www.dynamoRIO.org/ Accessed 26 Feb 2020
4. Cantrill, B., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: USENIX Annual Technical Conference, pp. 15–28 (2004)
5. Capstone. http://www.capstone-engine.org Accessed 26 Feb 2020
6. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing MAYHEM on binary code. In: 2012 IEEE Symposium on Security and Privacy, pp. 380–394 (2012)

7. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy, pp. 725–741 (2015)
8. Chen, J., Diao, W., Zhao, Q., et al.: IoTFuzzer: discovering memory corruptions in IoT through app-based fuzzing. NDSS (2018)
9. Eustace, A., Eustace, A.: Atom: a system for building customized program analysis tools. PLDI, pp. 196–205 (1994)
10. Gan, S., Zhang, C., Qin, X., et al.: Collafl: path sensitive fuzzing. In: IEEE Symposium on Security and Privacy (SP), pp. 679–696 (2018)
11. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. **40**(6), 213–223 (2005)
12. Höschele, M., Zeller, A.: Mining input grammars from dynamic taints. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 720–725 (2016)
13. Kargén, U., Shahmehri, N.: Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 782–792 (2015)
14. Laurenzano, M.A., Tikir, M.M., Carrington, L., Snavely, A.: Pebil: efficient static binary instrumentation for linux. ISPASS, pp. 175–183 (2010)
15. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not. **40**, 190–200 (2005)
16. Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D.: What you corrupt is not what you crash: challenges in fuzzing embedded devices. NDSS (2018)
17. Nagy, S., Hicks, M.: Full-speed fuzzing: reducing fuzzing overhead through coverage-guided tracing. In: IEEE Symposium on Security and Privacy (SP), pp. 787–802 (2019)
18. Nanda, S., Li, W., Lam, L.C., Chiueh, T.C.: Bird: binary interpretation using runtime disassembly. CGO, pp. 358–370 (2006)
19. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Not. **42**, 89–100 (2007)
20. Nordrum, A.: The internet of fewer things [news]. IEEE Spectr. **53**(10), 12–13 (2016)
21. Pak, B.S.: Hybrid fuzz testing: discovering software bugs via fuzzing and symbolic execution. School of Computer Science Carnegie Mellon University (2012)
22. Pani, P.: Measuring code coverage on an embedded target with highly limited resources. Master's Thesis. Graz University of Technology (2014)
23. Rawat, S., Jain, V., Kumar, A., Bos, H.: VUZzer: application-aware evolutionary fuzzing. Network and Distributed System Security Symposium (2017)
24. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. USENIX Security, pp. 861–875 (2014)
25. She, D., Pei, K., Epstein, D., et al.: NEUZZ: efficient fuzzing with neural program smoothing. In: IEEE Symposium on Security and Privacy (SP), pp. 803–817 (2019)
26. Srivastava, A., Edwards, A., Vo, H.: Vulcan: binary transformation in a distributed environment. Technical Report msr-tr-2001-50. Microsoft Research (2001)
27. Srivastava, P., Peng, H., Li, J., et al.: Firmfuzz: automated IoT firmware introspection and analysis. In: Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, pp. 15–21 (2019)
28. Xu, Y., Ren, J., Wang, G., Zhang, C., Yang, J., Zhang, Y.: A blockchain-based nonrepudiation network computing service scheme for industrial IoT. IEEE Transactions on Industrial Informatics **15**(6), 3632–3641 (2019)
29. Zhao, Q., Koh, D., Raza, S., Bruening, D., Wong, W., Amarasinghe, S.: Dynamic cache contention detection in multi-threaded applications. ACM SIGPLAN Not. **46**(7), 27–38 (2011)