IoT: A Static Instrumentation Tool for IoT Devices

CHEN CHEN1,2, ZHE JING1, JINXIN MA3, BAOJIANG CUI1, HAN XU1, AND QUANCHEN ZOU4
1School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China
2School of Information and Navigation, Air Force Engineering University, Xi’an 710077, China
3China Information Technology Security Evaluation Center, Beijing 100085, China
4360 Security Research Labs, Beijing Qihu Technology Co., Ltd., Beijing 100016, China
Corresponding author: Chen Chen (00152tenten@bupt.edu.cn)

This work was supported by the National Natural Science Foundation of China under Grant 61872386.

ABSTRACT In recent years, an increasing number of Internet of Things (IoT) devices have been employed in various fields, which has caused an increased demand for IoT device testing and analysis. However, due to the strictly limited software interfaces and hardware resources of IoT devices, traditional instrumentation-based testing and analysis technologies cannot be effectively used in IoT devices. In this paper, to address the problems encountered in IoT device detection and analysis using instrumentation methods, we present a prototype novel instrumentation tool, IoTSIT, that is suitable for the static instrumentation of firmware in IoT devices. This tool forcibly writes instrumented code into files while leaving the original program logic intact. A comparison of IoTSIT with several other instrumentation tools confirms that IoTSIT offers advantages in terms of time efficiency and code expansion rate for firmware instrumentation.

INDEX TERMS Firmware, instrumentation, IoT.

I. INTRODUCTION
The Internet of Things, or IoT, is a system of interrelated computing devices that are provided with the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. IoT devices are increasingly employed in several fields, such as industries, national defense and medical treatments. A common view is that the number of IoT devices will reach 20 billion by 2020 [1]. Many IoT devices can interact with the external physical world. Therefore, the functions of an IoT device must be tested in a real-world environment to find problems that may result when firmware is deployed on the device [2]. In addition, because IoT devices interact with the external physical world, they present considerable security risks [3]–[5].

One way to achieve function testing and security analysis for traditional software is to instrument the binary code. This approach can be utilized to analyze data and behaviors [6], [7], precisely identify performance bottlenecks [8] when running a program, or obtain run-time information [9]–[11] to support other testing techniques for discovering exceptions and vulnerabilities in a program. However, it is very difficult to apply instrumentation tools to IoT devices, mainly for the reasons given below.

Instrumentation technology can be divided into dynamic instrumentation and static instrumentation. Dynamic instrumentation tools [12]–[15] insert analytical code into the memory locations of concern while a program is running. This kind of instrumentation generates considerable interaction among the original code, instrumented code and control code, which has a substantial impact on the run-time performance of software. Such tools cannot be supported by many IoT devices because of their limited hardware resources or software interfaces.

Static instrumentation tools [16]–[20] insert analytical code into a program file before it runs. Some of these tools require the support of debugging interfaces [19], [20]. The debugging interface is a software interrupt with a context switch to user mod to execute the inserted code. It can lead to considerable delay in the system or may result in failure to detect some real-time IoT devices. In addition, current static instrumentation tools are designed for software running on desktop system [16]–[18]. They ignore the space occupied by instrumentation, which will hinder perform instrumentation in embedded devices because of the limited storage space in some cases.
Because of the above problems, firmware analysis and detection present difficult challenges for IoT devices. At present, these tasks can be performed only by means of dynamic black box detection [21]–[23] or static analysis [24]–[26] because dynamic run-time information cannot be effectively obtained and analyzed. This obstacle currently limits the efficiency of function and safety testing for IoT devices, and hence, it is difficult to meet the actual needs for the effective testing of certain equipment.

In this paper, we present a prototype instrumentation tool named IoTSTIT. It uses jump instruction and trampoline to link the inserted analysis code. The introduction of trampoline makes the inserted code completely independent, and only one copy is required in the binary file, which reduces the expansion rate of the executable file. Replacing the debugging interface with jump instruction not only improves the efficiency of testing but also avoids instrumentation failure on IoT devices that lack the debugging interface. Our work makes the following three contributions:

- We design and develop a prototype instrumentation tool, IoTSTIT, to test firmware in IoT devices with limited resources.
- We apply trampoline technology to reduce the space occupancy and run-time consumption of the instrumented code in a binary file while avoiding the use of debugging interfaces.
- We test the impact of IoTSTIT on file expansion and verify the effectiveness of IoTSTIT by comparing it with four other instrumentation tools: Pin, Valgrind, DynamoRIO and Dyninst.

The remainder of this paper is structured as follows. Section 2 discusses related works. Section 3 introduces the design of IoTSTIT. Section 4 describes the testing of the tools and presents an analysis of the test results. Section 5 lists the limitations and deficiencies of IoTSTIT. Section 6 presents the conclusions of this study.

II. RELATED WORKS

A. DYNAMIC INSTRUMENTATION

Dynamic instrumentation is a type of instrumentation technology that inserts additional code at specified memory locations while a process is running for program analysis and security detection. Many tools [12]–[14] provide dynamic instrumentation functionality.

Dyninst [12] inserts two-part snippets into original programs. The first part is referred to as a mutator, which implements a run-time compiler and utility routines to manipulate the application process. The second part is a run-time library that supports the Dyninst Application Programming Interface (API). Dyninst leverages many functions of operation systems, such as ptrace, procs, and dynamic memory allocation; however, these functions do not exist in many IoT devices, and the use of ptrace degrades the performance.

Pin [13] is a dynamic instrumentation tool provided by Intel that utilizes an approach based on just-in-time (JIT) compilation for instrumentation. Pin intercepts the execution of the first instruction in a software, compiles new code for the straight-line code sequence starting at this instruction, and then transfers control to the newly generated sequence. Pin supports programs that run on traditional desktop operating systems, such as Linux, Windows and OS X, but it supports only Intel platforms and cannot be employed for current IoT devices.

Valgrind [14] also employs JIT-compiled dynamic instrumentation technology. Compared with Pin, Valgrind is a heavyweight instrumentation tool that can be applied for memory debugging, memory leak detection and performance analysis, and provides support for multiple platforms, including x86, AMD64, PPC32, S390X, ARM, ARM64 and MIPS32. However, JIT technology relies on a certain degree of hardware and software resources, which render it inapplicable to many IoT devices.

Although these tools provide dynamic instrumentation functionality, they are designed for traditional desktop systems rather than IoT systems. For example, when code is inserted into the memory space, support from the operating system is needed, such as ptrace system calls and dynamic memory allocation in a Linux system, which is not provided in many IoT devices. Moreover, the limited storage space of these devices is not always adequate for running these tools.

B. STATIC INSTRUMENTATION

Different from dynamic instrumentation, which writes analysis code into memory at runtime, static instrumentation directly writes analysis logic into binary files before it runs. Some instrumentation tools [17]–[20] adopt this static type.

Atom [17] is an early static instrumentation tool that is based on OM [31]. This tool operates directly on the compiled binary code but relies on symbol table information from debugging files, and hence, cannot be applied to third-party commercial binaries. Moreover, this tool can be applied on Alpha platforms.

Microsoft’s Vulcan [18] converts a binary file into an abstract representation, inserts code into this abstract representation and converts the abstract representation to a binary program. Vulcan also relies on symbol table information from debugging files, which makes application to commercial binary programs difficult.

Bird [19] is an instrumentation tool that works on Windows/x86 platforms and employs dynamic and static disassembly technology to improve disassembly. Bird applies debugging techniques to solve the problem of long jump instructions; however, this approach may cause a program fail when running a program that leverages anti-debugging measures.

Pani [20] implements static instrumentation for firmware in embedded systems on MRK-III platforms. Pani rewrites each RET instruction in an ELF executable binary file with a software interrupt user call (USR) to jump to a special function and then jump back to the return address of the function that belongs to the patched RET instruction.
However, only the coverage information of this function is dynamically obtained, while extending instrumentation to other functions is difficult.

Generally, current mature instrumentation tools mainly adopt the dynamic instrumentation method. However, dynamic instrumentation requires information interaction between the instrumentation tool and the program that is being tested at runtime, which will cause numerous running costs, and rich interface support of the operating system. Thus, the application of dynamic instrumentation to the IoT devices is difficult with limited hardware resources and rare software interfaces. Static instrumentation can write analysis logic directly into binary files, which provides the possibility for the inserted code and original code to run in the same process. However, if it is to be employed in IoT devices, static instrumentation also needs to reduce the dependence on the debugging information and take into account the code expansion rate of firmware.

III. DESIGN OF IoTSIT

We apply a jump instruction and trampoline to link the instrumented code and write it directly into a new section in the ELF file. In this way, the instrumented code and original code work in the same process, which will avoid performance loss caused by information exchange between processes. The structure of IoTSIT is shown in figure 1.

The three core modules in IoTSIT are a static analyzer, an instrumentation logic generator and an instrumented code writer. IoTSIT first performs static analysis of the target binary file and obtains the information needed for instrumentation. Then, it creates instrumented code in accordance with the user’s requirements and inserts that code into the target binary file. Next, we will introduce the three core modules in IoTSIT.

A. BINARY ANALYSIS

Binary analysis provides the necessary information for instrumentation. It mainly includes code acquisition, control flow graph construction, and the creation of two data structures. The process of binary analysis is shown in figure 2.

1) CODE ACQUISITION

By analyzing the file header, IoTSIT obtains the locations and lengths of the data, text and other sections, the program entry address and other information. Using this information, data in the relevant segments can be further acquired. For example, all code instructions are stored in the text section. IoTSIT employs Capstone [27] to disassemble the obtained code.

2) CONTROL FLOW GRAPH CONSTRUCTION

A control flow graph is an abstract representation of a program, which describes the possible flow direction of all basic blocks in the program in the form of a graph. A control flow graph can be expressed as $G = (V, A)$, where each element $v_i$ ($i = 1, 2, \ldots, n$) in $V$ represents a vertex in the graph, that is, a basic code block in the program, and $A = (a_1, a_2, \ldots, a_m)$ is the arc set of the graph. The $k$th element in $A$ is denoted by $a_k$, where $a_k = (v_i, v_j)$, being an arc from $v_i$ to $v_j$, representing a jump from one basic block to another. The control flow graph is constructed by analyzing the executable code in two rounds. In the first round, IoTSIT expresses the scope of each basic block as $v_i$ when it encounters jump instructions. In the second round, the arcs($v_i, v_j$) are formed by connecting each basic block $v_i$ in which a jump instruction is located to the basic block $v_j$ to which that jump instruction jumps.

3) CREATION OF TWO DATA STRUCTURES

The basic block mapping table and the execution record unit of the basic blocks are generated after the construction of the control flow diagram. The execution record unit of the basic blocks is mainly used to collect executed basic block information at runtime. Each bit in the unit is mapped to a basic block by the basic block mapping table, which records the addresses of the basic blocks mapped to the bits in the unit. This mapping relationship is constructed by traversing each basic block in the control flow graph. These two data structures are mainly used to support basic block operations and are often used to collect path and coverage information for application analysis.

B. INSTRUMENTATION MECHANISM

We utilize trampoline [28] to organize the inserted code, write it directly into a new section in the ELF file, and then...
modify the code at the insertion point as a jump instruction to associate the original code and the inserted code. To save code space, we divide the inserted code into two parts to ensure that only one copy of the core logic exists. These two parts are trampoline and primitive. Primitive is the core logic to be inserted, and trampoline is employed to govern the operation of the primitive. The principle is shown in Figure 3.

1) TRAMPOLINES
There are two main purposes of a trampoline. The first is to prevent the run-time context of the original program from being destroyed after a primitive is executed; this is done by saving and restoring registers, relocating instructions, and modifying jump addresses. The second is to provide parameters and return addresses for the primitives.

a: SAVING AND RESTORING REGISTERS
The execution of instrumented code may use or affect register data. Therefore, the state of a register must be saved before the instrumented code is executed and then be restored after the instrumented code is executed. To minimize the performance loss caused by instrumentation without affecting the normal execution of the program, IoTSIT stores only the registers required by the primitives.

b: RELOCATING INSTRUCTIONS
To shift the flow of the original program to the instrumented code, the code at the insertion point is replaced with a jump instruction. After the insertion function has been completed, first, the overwritten code needs to be re-executed, and second, the program flow jumps back to the first position of the unaffected instructions after the insertion point to continue execution.

c: PARAMETER PREPARATION
To limit the space occupied by the stub code, only one copy of different primitives that are repeatedly executed in the program exists. IoTSIT places the parameter preparation process that is required for calling the primitive operations in the trampoline.

d: PUSHING THE RETURN ADDRESS
Using direct jump instructions to return would result in wasted space because even a primitive operation of the same type will cause a difference in the jump address, which would cause each primitive operation to require a separate copy. Instead, IoTSIT pushes the return address to redirect the primitive operation to the next instruction in the trampoline without a direct jump instruction.

2) PRIMITIVES
IoTSIT primitive operations include executed basic blocks recording, executed function recording and executed paths recording. More primitives can be added on the framework of IoTSIT by users. To support these primitives, we need to consider the following three aspects: record storage, stack balancing and implementation of primitives.

a: RECORD STORAGE
Record information for all primitive operations is stored in a specified memory area where the data are modified during primitive operations. This record information can be sent
to the client in a manner that the system allows when the record output primitive is executed. For example, this record information can be sent through serial ports or network ports or can be written to files.

b: STACK BALANCING
Stack balancing is used to mitigate the impact of primitive execution and parameter preparation on the stack. Stack balancing is not always performed for each primitive. Whether stack balancing is performed depends on whether the primitive operation and parameter preparation affect the stack.

c: IMPLEMENTATION OF PRIMITIVES
The primitives for basic block recording depend on the execution record unit of the basic blocks that is generated in the binary analysis stage. When a primitive is executed, the relevant bit mapped to the current block in the record unit will be set to 1. The data from the record unit are eventually sent to the analyzer, which can parse out the executed basic blocks using the basic block mapping table generated during the binary analysis stage. The implementation of primitives for executed function recording and executed paths recording is simpler than basic block recording, which only records the address of the instrumented points.

C. BINARY REWRITING
Dynamic instrumentation [12]–[15] can easily control an original program, but it will cause excessive performance degradation through the interaction between processes. Conversely, IoTSIT employs a completely static method to directly insert analysis code into a binary file, such that the analysis code and original code can be executed as part of the same process, which avoids the performance degradation caused by the communication between processes.

Binary rewriting is the process of writing the instrumented code into the static binary file. The main problems to be considered are how to rewrite the file without breaking the logic of the original code, thus ensuring that the program loads normally, and how to rewrite the generated code into the appropriate locations, thus ensuring the intended operation of the instrumented logic.

IoTSIT currently supports the implementation of binary rewriting for the ELF file type, for which the process includes three main aspects. The first aspect is to add an extra section at the end of the file, which stores the code for the trampolines and primitives. The second aspect is to displace the code at each insertion point to a jump instruction in the original code in accordance with the actual needs of instrumentation. The third aspect is to add a program header, which ensures that the added extra section of code can be loaded correctly, and modify the values in the program header and section header.

IV. EXPERIMENTS AND RESULTS
The experiment is divided into four groups: the analysis of software expansion caused by instrumentation, time consumption by different types of instrumentation point triggering, time consumption by instrumentation with actual software, and impact of instrumentation on the performance of IoT devices firmware.

The first three experiments were performed for comparison with existing instrumentation tools: Pin, DynamoRIO, Valgrind and Dyninst. Existing publicly available instrumentation tools only support desktop-oriented platforms and cannot support IoT devices. To verify the effectiveness of IoTSIT by comparison with various instrumentation tools, we tested it on a Linux platform that is based on x86 because most instrumentation tools can function normally in this environment. Our hardware configuration in the experiments is expressed as follows: Intel(R) Core(TM) i5-5200U CPU @ 2.20 GHz and 4 GB of RAM. In addition, the fourth experiment is carried out on an actual IoT device. We employ the home wireless router ASUS-AC1750.

A. SOFTWARE EXPANSION CAUSED BY INSTRUMENTATION
Static instrumentation involves directly writing additional code into binary files, which results in larger binary files. The extent of expansion needs to be considered in many IoT devices with limited storage resources. In our first experiment, we tested the influence of IoTSIT’s instrumentation operation on the size of binary files. Pin, DynamoRIO and Valgrind perform instrumentation at run time and do not rewrite the binary files. Therefore, they do not affect the size of the program being tested. Dyninst inserts instrumented code and related control logic into binary files. In this experiment, we tested only the impact of Dyninst and IoTSIT on the size of binary files. We instrumented eight software programs to collect the corresponding number of basic blocks, measured the size of the binaries and computed the expansion rates of the binary files after instrumentation.

As shown in Table 1, the binary files after instrumentation with IoTSIT had expanded by between 10% and 76% compared with the original binaries, while after instrumentation with Dyninst, the expansion rate was between 347% and 1336%. The reason for this large gap is that Dyninst is designed to work on desktop systems, and its design does not account for the impact on the file size. During the process of instrumentation, Dyninst inserts operation modules for instrumentation, and a library may not be needed in the original software, which causes a high expansion rate of the instrumented software. Although dynamic instrumentation tools (e.g., Pin, DynamoRIO and Valgrind) do not affect the size of the binary file, these instrumentation tools must be run on the device where the binary file is located to perform the instrumentation operation. Therefore, for instrumenting firmware on IoT devices with limited resources, not only the space occupancy of the instrumented binary but also the space occupancy of the instrumentation tool itself need
to be considered. We list the sizes of the three dynamic instrumentation tools in Table 2.

### TABLE 2. File sizes of three dynamic instrumentation tools.

| Instrumentation tool | Pin          | DynamoRIO      | Valgrind       |
|----------------------|--------------|----------------|----------------|
| Version              | 3.7          | 7.0            | 3.14           |
| Binary size (MB)     | 34           | 96             | 16             |

The smallest of these instrumentation tools is Valgrind, at 16 MB. The largest is DynamoRIO, at 96 MB. Although 96 MB is small for a universal computer, the total storage capacity in many IoT devices is only a few MB. In this case, even Valgrind may be too large.

### B. TIME CONSUMPTION BY DIFFERENT TYPES OF INSTRUMENTATION POINT TRIGGERING

Although the principles of different instrumentation tools are different, the time impacts of instrumentation can be classified into three main types: instrumentation at one point executed many times, which is often used to collect the number of times a function is executed; instrumentation at many points executed one time each, which is often used to collect which functions are executed; and instrumentation at many points executed many times each, which is often used to collect the execution paths and basic blocks of programs.

To accurately test these three situations, satisfying our test requirements using actual software is difficult; so we designed a program to be tested. This program is a loop in which the number of cycles $m$ can be established, where $m$ is applied to control the number of times the instrumentation point is triggered. Inside the loop are $n$ if statements, where $n$ is used to control the number of instrumentation points.

The first experiment was designed to analyze the impact of instrumentation on a specified code block executed many times and compare this impact among the different tools. The time consumption of instrumentation was collected, and the detailed data are listed in Table 3.

The second experiment was designed to analyze the impact of instrumentation on many code blocks each executed one time and compare this impact among the different tools. The time consumption of instrumentation was collected, and the detailed data are listed in Table 4.

The third experiment was designed to analyze the impact of instrumentation on many code blocks executed many times each and compare this impact among the different tools. It is equivalent to the superposition of Experiment 1 and Experiment 2. It increases the number of basic blocks to be instrumented as well as the number of times each basic block is instrumented. The time consumption of instrumentation are listed in Table 5.

### TABLE 3. Comparison of the different tools in terms of the time consumption for instrumentation at one point executed many times.

| Instrumentation tool | 10 t | 50 t | 100 t | 500 t | 1000 t |
|----------------------|------|------|-------|-------|--------|
| Pin (ms)             | 488  | 498  | 510   | 478   | 522    |
| Valgrind (ms)        | 304  | 313  | 351   | 358   | 414    |
| DynamoRIO (ms)       | 79   | 88   | 94    | 101   | 107    |
| Dyninst (ms)         | 3    | 3    | 5     | 6     | 8      |
| IoTSIT (ms)          | 2    | 2    | 3     | 3     |        |
| No instr. (ms)       | 2    | 2    | 2     | 2     | 3      |

*The number of times that an instrumented code position is executed*

### TABLE 4. Comparison of the different tools in terms of the time consumption for instrumentation at many points executed one time each.

| Instrumentation tool | 10p  | 50p  | 100p | 500p | 1000p |
|----------------------|------|------|------|------|-------|
| Pin (ms)             | 965  | 984  | 1036 | 1119 | 1325  |
| Valgrind (ms)        | 610  | 649  | 668  | 721  | 772   |
| DynamoRIO (ms)       | 156  | 164  | 178  | 212  | 282   |
| Dyninst (ms)         | 7    | 8    | 10   | 16   | 19    |
| IoTSIT (ms)          | 4    | 5    | 7    | 14   | 16    |
| No instr. (ms)       | 4    | 4    | 5    | 14   | 12    |

*The number of code positions instrumented*

### TABLE 5. Comparison of the different tools in terms of the time consumption for instrumentation at many points executed many times each.

| Instrumentation tool | 10p 10t | 50p 50t | 100p 100t | 500p 500t | 1000p 1000t |
|----------------------|----------|---------|-----------|-----------|-------------|
| Pin (ms)             | 1223     | 1251    | 1390      | 1764      | 1911        |
| Valgrind (ms)        | 722      | 753     | 722       | 881       | 938         |
| DynamoRIO (ms)       | 122      | 152     | 183       | 221       | 272         |
| Dyninst (ms)         | 4        | 9       | 18        | 21        | 40          |
| IoTSIT (ms)          | 4        | 5       | 17        | 20        | 36          |
| No instr. (ms)       | 4        | 4       | 10        | 14        | 20          |

*The number of times that an instrumented code position is executed*
control over the original software through interprocess communication, which also reduces the efficiency. Conversely, IoTSIT inserts analysis code into the original program in a completely static way. The inserted code and the original program function together as part of the same process, so there is no overhead caused by interprocess communication.

### C. TIME CONSUMPTION BY INSTRUMENTATION ON ACTUAL SOFTWARE

In this set of experiments, we used the five tools to instrument eight programs to collect the numbers of executed basic blocks. The operation of collecting basic blocks is a common operation in software security detection. It can be used to calculate code coverage or guide the selection of samples for vulnerability detection. Ten samples were used to test each program, and we collected the average time required to process these samples for each program. The graphical comparison results are shown in figure 4.

As shown in figure 4, compared with the other instrumentation tools, IoTSIT has the least impact on the time overhead when processing samples. This result is reasonable considering that IoTSIT is based on static instrumentation. In the case of static instrumentation, the execution of the instrumented binary does not depend on real-time control by the instrumentation tool. The time overhead mainly originates from the execution of the instrumented code itself, so there is no switching overhead between the code to be tested and the instrumented code. However, such control switching has a considerable impact on the running efficiency of programs when dynamic instrumentation tools are used, especially heavyweight instrumentation tools such as DynamoRIO and Valgrind.

### D. EXECUTION OF FIRMWARE INSTRUMENTED WITH IoTSIT ON IoT DEVICES

In the fourth set of experiments, we tested the running efficiency of binary files after instrumentation on an actual IoT device. We tested the web server `httpd` on the ASUS-AC1750 wireless router (ARM). We performed three experiments, which correspond to the three types of experiments in Section IV-B. Each experiment is performed for different goals of instrumentation. We collected the requesting time of visiting `httpd` before and after instrumentation ten times.

In the first experiment, we collected the execution times of two functions (addresses of the functions are `0x10730` and `0xD8BC`), which belongs to the instrumentation at one point executed many times. In the second experiment, we determined how many functions were executed, which belongs to instrumentation at many points executed one time each. In the third experiment, we collected the internal execution paths of two functions (addresses of the functions are `0x10730` and `0xD8BC`), which belongs to the instrumentation at many points executed many times each. `httpd` was instrumented only with IoTSIT because the other instrumentation tools cannot run on the device. The results are shown in figure 5.

Figure 5 shows that the processing time of each request is different, which is mainly caused by the instability of the router’s response to the requests. If the impact of instrumentation on `httpd` is large, the fold lines generated by the instrumented `httpd` should be significantly higher than those generated with-out instrumentation. The fold lines in the...
three figures all intersect each other, which shows that the request processing time of httpd after instrumentation is not significantly longer than that without instrumentation. In addition, the increase in the number of instrumentation points has little effect on the performance of instrumented httpd, which can be seen from the intersection of the two fold lines generated by httpd with different numbers of instrumented points in figure 5-1 and figure 5-3.

V. LIMITATIONS AND DEFICIENCIES
IoTSIT is a prototype instrumentation tool. It does not provide all the functions provided by a mature instrumentation tool. More work is still needed to perfect IoTSIT as a mature static instrumentation tool. IoTSIT cannot instrument self-modifying code, which is an open problem in static instrumentation in general [30]. IoTSIT currently supports ELF executable programs for 32-bit ARM and x86 architectures. However, the modular structure of IoTSIT will enable extension to other file formats and platforms. The use of IoTSIT requires certain conditions, which necessitates to some restrictions on its use. IoTSIT needs to be able to perform static analysis of firmware, and hence, it needs to be able to obtain the firmware from an IoT device and then place it back on the device after it is instrumented. Although some techniques and methods are available for obtaining firmware from IoT devices, in practice, it is difficult to obtain firmware from some devices. This problem is beyond the scope of the current study.

VI. CONCLUSIONS
Traditional instrumentation tools cannot be effectively employed in IoT devices due to the strictly limited software interfaces and hardware resources of IoT devices. In this paper, we present a prototype novel instrumentation tool, IoTSIT, which forcibly writes instrumented code into files while leaving the original program logic intact. IoTSIT has advantages in terms of the time efficiency and code expansion rate for binary instrumentation in comparison with other instrumentation tools, and can be effectively applied in an actual IoT device.

REFERENCES
[1] A. Nordrum, “The Internet of fewer things [News],” IEEE Spectr., vol. 53, no. 10, pp. 12–13, Oct. 2016.
[2] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, “Sympathy for the sensor network debugger,” in Proc. SENSYS, 2005, pp. 255–267.
[3] M. M. Kermani, M. Zhang, A. Raghunathan, and N. K. Jha, “Emerging frontiers in embedded security,” in proc. VLSID, Jan. 2013, pp. 203–208.
[4] P. Koopman, Embedded System Security. Los Alamitos, CA, USA: IEEE Computer Society Press, 2004.
[5] S. Parameswaran and T. Wolf, “Embedded systems security—An overview,” Design Autom. Embedded Syst., vol. 12, no. 3, pp. 173–183, Sep. 2008.
[6] D. Bruening and Q. Zhao, “Practical memory checking with Dr. Memory,” in proc. CGO, 2011, pp. 213–223.
[7] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W. F. Wong, “How to do a million watchpoints: Efficient debugging using dynamic instrumentation,” in Proc. Joint Eur. Conf. Theory Practice Softw. Int. Conf. Compiler Construct., vol. 4959, Berlin, Germany: Springer, 2008, pp. 147–162.
[8] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, “Dynamic cache contention detection in multi-threaded applications,” ACM SIGPLAN Notices, vol. 46, no. 7, pp. 27–38, Jul. 2011.
[9] M. Zalewski. American Fuzzy Lop. Accessed: Apr. 22, 2020. [Online]. Available: https://github.com/mincore/ad.
[10] D. Bruening. Accessed: Apr. 22, 2020. [Online]. Available: https://github.com/google/honggfuzz
[11] Syzkaller. Accessed: Apr. 22, 2020. [Online]. Available: https://github.com/syzkaller
[12] Dyninst. Accessed: Apr. 22, 2020. [Online]. Available: http://www.dyninst.org
[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klausen, G. Lowney, S. Wallace, R. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” ACM SIGPLAN Notices, vol. 40, no. 6, pp. 190–200, 2005.
[14] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” ACM SIGPLAN Notices, vol. 42, no. 6, pp. 89–100, Jun. 2007.
[15] D. Bruening. Dynamorio: Dynamic Instrumentation Tool Platform. Accessed: Apr. 22, 2020. [Online]. Available: http://www.dynamorio.org/
[16] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, “PEBL: Efficient static binary instrumentation for linux,” in Proc. ISPASS, Mar. 2010, pp. 175–183.
[17] A. Eustace and A. Eustace, “ATOM: A system for building customized program analysis tools,” in Proc. PLDI, 1994, pp. 196–205.
[18] A. Srivastava, A. Edwards, and H. Vo, “Vulcan: Binary transformation in a distributed environment,” Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2001-50, 2001.
[19] S. Nanda, W. Li, L.-C. Lam, and T.-C. Chien, “BIRD: Binary interpretation using runtime disassembly,” in Proc. CGO, 2006, pp. 358–370.
[20] P. Pani, “Measuring code coverage on an embedded target with highly limited resources,” M.S. thesis, Inst. Appl. Inf. Process. Commun., Graz Univ. Technol., Graz, Austria, 2014.
[21] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, “What you corrupt is not what you crash: Challenges in fuzzing embedded devices,” in Proc. NDSS, 2018.
[22] Peach. Accessed: Apr. 22, 2020. [Online]. Available: http://www.peachfuzzer.com/products/peach-platform
[23] J. Pereyda. Boofuzz: Network Protocol Fuzzing for Humans. Accessed: Apr. 22, 2020. [Online]. Available: https://github.com/jpereyda/boofuzz
[24] S. L. Thomas, T. Chothia, and F. D. Garcia, “Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality,” in Proc. ESORICS, 2017, pp. 513–531.
[25] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting Web application vulnerabilities,” in Proc. S&P, 2006, pp. 258–263.
[26] J. Dahse and T. Holz, “Simulation of built-in php features for precise static code analysis,” in Proc. NDSS, 2014.
[27] Capstone. Accessed: Apr. 22, 2020. [Online]. Available: http://www.capstone-engine.org/
[28] J. K. Hollingsworth, O. Niam, B. P. Miller, Z. Xu, M. J. R. Goncalves, and L. Zheng, “MDL: A language and compiler for dynamic program instrumentation,” in Proc. PACT, 1997, pp. 201–212.
[29] J. K. Hollingsworth, B. P. Miller, and J. Cargille, “Dynamic program instrumentation for scalable performance tools,” in Proc. Scalable High Perform. Comput. Conf., 2002, pp. 841–850.
[30] J. R. Larus and E. Schnarr, “EEL: Machine-independent executable editing,” in Proc. PLDI, 1995, pp. 291–300.
[31] A. Srivastava and D. W. Wall, “A practical system for intermodule code optimization at link-time,” J. Program. Lang., vol. 1, no. 1, pp. 1–18, 1992.

CHEN CHEN is currently pursuing the Ph.D. degree with the Beijing University of Posts and Telecommunications, with a major in network information security. His research interests include program analysis, software security, embedded device security, and protocol security.
ZHE JING is currently pursuing the M.S. degree with the Beijing University of Posts and Telecommunications, with a major in network information security. His research interests include software security and embedded device security.

JINXIN MA received the Ph.D. degree in computer science and technology from Central South University, China, in 2019. From November 2015 to February 2017, he worked with the China Information Technology Security Evaluation Center. His research interests include program analysis, software security, embedded device security, and protocol security.

BAOJIANG CUI is currently a Professor of network information security with the Beijing University of Posts and Telecommunications. His research interests include network and host security behavior analysis, software security detection, analysis of web/software and operating system security defects, smart terminals and mobile internet security, and the Internet of Things security.

QUANCHEN ZOU received the Ph.D. degree from the Communication University of China. He is currently a Senior Security Research Engineer with 360 Security Research Labs. His primary research interests include the intersection of computer security and machine learning.

ZHE JING is currently pursuing the M.S. degree with the Beijing University of Posts and Telecommunications, with a major in network information security. His research interests include software security and embedded device security.

HAN XU is currently pursuing the M.S. degree with the Beijing University of Posts and Telecommunications, with a major in network information security. Her research interests include software security and embedded device security.

VOLUME 8, 2020