Hindawi
Security and Communication Networks
Volume 2018, Article ID 7693861, 13 pages
https://doi.org/10.1155/2018/7693861

Research Article

OFFDTAN: A New Approach of Offline Dynamic Taint Analysis for Binaries

Xiajing Wang, Rui Ma, Bowen Dou, Zefeng Jian, and Hongzhou Chen

Beijing Key Laboratory of Software Security Engineering Technology, School of Software, Beijing Institute of Technology, Beijing 100081, China

Correspondence should be addressed to Rui Ma; mary@bit.edu.cn

Received 11 January 2018; Revised 10 March 2018; Accepted 8 April 2018; Published 30 May 2018

Academic Editor: Raymond Choo

Dynamic taint analysis is a powerful technique for tracking the flow of sensitive information. Different approaches have been proposed to accelerate this process in an online or offline manner. Unfortunately, most of these approaches still have performance bottlenecks and thus reduce analytical efficiency. To address this limitation, we present OFFDTAN, a new approach of offline dynamic taint analysis for binaries. OFFDTAN can be described in terms of four stages: dynamic information acquisition, vulnerability modeling, offline analysis, and backtrace analysis. It first records program runtime information and models the stack buffer overflow vulnerabilities and controlled jump vulnerabilities. Then it performs offline analysis and backtrace analysis to locate vulnerabilities. We implement OFFDTAN on the basis of QEMU virtual machine and apply it to off-the-shelf applications. In order to illustrate how our approach works, we first employ a case study. Furthermore, six applications have been verified so as to evaluate our approach. Experimental results demonstrate that our approach is correct and effective. Compared with other offline analysis tools, OFFDTAN has much lower application runtime overhead.

1. Introduction

Software vulnerabilities are the root of various cyber-attacks. Common attacks, such as XSS and buffer overflows, all exploit software vulnerabilities; thus vulnerability analysis attracts extensive research during the past decade. One of the research hotspots is vulnerability analysis for binary program, which can be performed dynamically or statically. In the dynamic analysis, the taint analysis techniques have been studied recently, and numerous taint analysis tools have been implemented and applied extensively to the field of vulnerability analysis for binary program [1].

Taint analysis was first proposed by funnywei at the summit of XCon2003 [2]. The basic idea is to label data originating from or arithmetically derived from untrusted sources, such as network input or user input, as tainted, then keep track of the propagation of tainted data, and further detect whether tainted data is used in dangerous ways. At present, taint analysis could be divided into static taint analysis and dynamic taint analysis on the basis of running state of analysis object. The former keeps track of tainted data by performing semantic and grammatical analysis of the source code. That has some advantages, like higher path coverage and better analysis efficiency, but there still exist false positives. However, the latter marks and tracks certain data during program runtime. That has better accuracy, but there also exist some problems such as lower path coverage, larger space overhead, and lower analysis efficiency.

To address these limitations, in this paper, we present OFFDTAN, an approach of offline dynamic taint analysis for binary program, implement it on top of QEMU [3] to support fine-grained real-time monitoring for target program, and evaluate it in six realistic applications. In order to illustrate how our approach works, we employ an extended case study. Specifically, in the proposed method, OFFDTAN uses Hook technology to mark taint source and acquires the program runtime information such as register and memory information. Additionally, we summarize the applicable model of stack buffer overflow vulnerabilities and controlled jump vulnerabilities, which can be further used for offline analysis. Moreover, to obtain accurate tainting results, we improve existing taint propagation policy, mainly considering the effect of the tainted instruction operation on flag registers and associated registers, which makes taint propagation policy more accurate than previous approach. In particular, taint propagation flow graph has been established to backtrack taint data and locate its specific offset within taint source file.

The contributions of this paper can be summarized as follows:

(i) *Presentation of framework*: we present OFFDTAN, a generic offline dynamic taint analysis framework that uses KVM acceleration on QEMU virtual machine to detect vulnerabilities.

(ii) *Enhancement of propagation policy*: we enhance taint propagation policy of flag register and related register and summarize two types of specific vulnerability models applicable to this method.

(iii) *Construction of propagation flow graph*: we propose the construction method of taint propagation flow graph to backtrack taint source, which can precisely locate the specific offset of taint data.

(iv) *Evaluation of framework*: we evaluate OFFDTAN by applying it to large off-the-shelf applications such as FeiQ and Microsoft Word 2010. The experimental results show that our approach is effective and has better performance.

The remainder of this paper is organized as follows. Related work was discussed in Section 2. Section 3 presents the details of proposed approach. Section 4 discusses the experimental results and validation, and our conclusions were finally proposed in Section 5.

2. Related Work

Dynamic taint analysis can be performed online or offline. Online analysis handles taint propagation during the program execution, while offline analysis first records the trace of program execution and then performs the taint analysis by replaying program.

Online analysis usually leverages instrumentation technology to monitor the taint propagation. Common instrumentation tools, such as Valgrind [4], Pin [5], and Dynamo [6], have been used extensively to implement most of online analysis tools. James Newsome et al. released TaintCheck [7] on the basis of Valgrind, which provides taint analysis for data flow and enables the detection of buffer overflow vulnerabilities. However, this tool needs larger space overhead and ignores the analysis for control flow. Considering the effect of control flow, Clause et al. propose Dytan [8], which achieves the analysis for control flow, but this tool still presents the limitation of time overhead. LIFT [9] is developed by Qin et al. based on StarDBT, which sharply shortens the duration of taint analysis by screening for unnecessary data flow information whereas the problem of memory consumption has not been properly addressed.

Due to the rise of symbolic execution, some researchers attempt to provide a combination method of dynamic taint analysis and symbolic execution, such as DTA++ [10], BitBlaze [11], and DECAF [12], which can improve the path coverage of dynamic taint analysis. Lai et al. [13] mark each byte of external input data to perform fine-grained taint analysis, which improves the granularity of dynamic taint analysis. Wang et al. [14] propose a method to bypass the checksum mechanism, which combines with symbolic execution and fine-grained dynamic taint analysis, to develop TaintScope. Zhuge et al. [15] present a method of type-based dynamic taint analysis, according to the type information of instructions and functions, which provides better semantic support. In addition, this method presents the combination of taint analysis and symbolic execution at variable level. Although above methods have improved the performance of online analysis, many problems are still not fundamentally solved, especially the limitation of high runtime overhead.

To address these limitations, several researchers propose the method of offline analysis, which is attractive at present. Jee et al. propose a dynamic taint analysis method based on shadow memory, which separates taint analysis from program execution and develops ShadowReplica [16]. Dan Caselden et al. introduce a hybrid information and control-flow graph (HI-CFG) and give algorithm to infer it from an instruction-level trace. Then they use the Tracecap tool of BitBlaze to record instruction traces [17]. Manolis Stamatogiannakis et al. [18] leverage full-system execution trace logging and replaying to decouple analysis from the original execution. Shi et al. [19] propose a combination of coarse-grained and fine-grained dynamic taint analysis (DTA) method. It executes online coarse-grained DTA to screen out effective instruction and then uses offline fine-grained DTA to calculate taint information. Wang et al. [20] introduce the propagation policy of multi-tainted label and implement the prototype system FlowWalker. Their taint propagation strategy makes a further support for the extended instruction set of MMX/SSE family. Ma et al. [21] present a taint analysis method based on trace offline indices which are byte-grained and utilize taint tags. Their approach fixes the problem of taint loss which resulted from just-in-time translation first time. Ming et al. [22] propose a full-featured offline taint analysis tool StraightTaint, which completely decouples the program execution and taint analysis, resulting in much lower execution slowdown. Dolan-Gavitt et al. [23] present a full-system analysis tool PANDA that is based on QEMU emulator and has the ability to record and replay executions.

The above-mentioned offline analysis methods alleviate the problem of lower analysis efficiency to a certain extent. However, most analysis tools or methods are running on the same operating system as the target program and thus cannot eliminate the impact of vulnerability analysis tool on the target program. PANDA eliminates above impact, but its performance is a bit slow. In particular, the propagation policies of these methods are still not complete.

To address the first problem, we managed to use QEMU virtual machine that supports KVM acceleration to create a simulated computer environment isolated from the host and precede fine-grained observation for relevant target program in the client. In addition, this paper develops appropriate propagation policy and vulnerability checking strategy to improve the accuracy of analysis. In the aspect of propagation policy, we focus on improving the taint update policy of flag

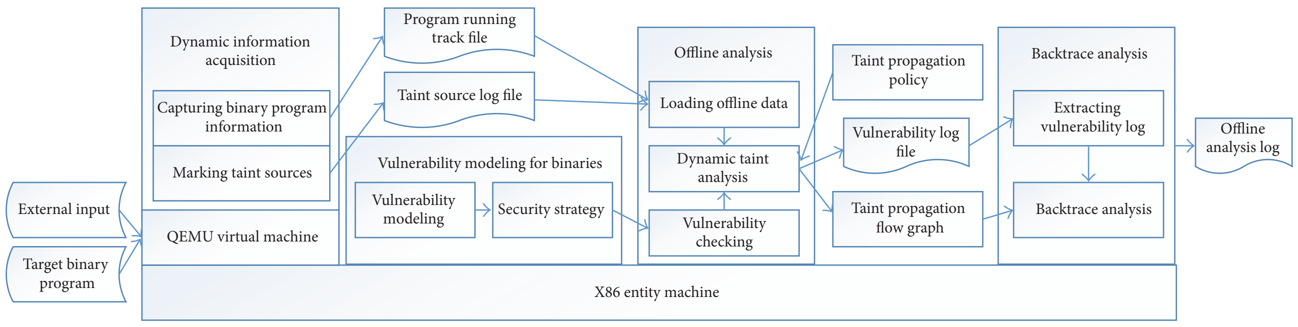


FIGURE 1: The OFFDTAN framework.

register and related register. To check vulnerabilities, this paper summarizes two types of specific vulnerability models applicable to this method and establishes the vulnerability checking strategy by studying the cause of released vulnerabilities. Moreover, the approach of backtrace analysis has been presented to locate taint data's specific offset within taint source file. This method further reduces the proportion of manual analysis and improves the efficiency of vulnerability analysis.

3. OFFDTAN

In this section, we first outline the framework of OFFDTAN and then detail each stage.

3.1. General Framework. This paper proposes OFFDTAN, an approach of offline dynamic taint analysis for binary program. We employ KVM acceleration on QEMU to implement OFFDTAN, because it is isolated from the operating system of the host. In order to better describe this method, it can be divided into the following four stages: dynamic information acquisition, vulnerability modeling, offline analysis, and backtrace analysis, as shown in Figure 1.

More precisely, dynamic information acquisition stage is used for dynamically recording the trace of program execution and taint source log, which can be used for offline analysis stage. The basic condition of this stage is that QEMU simulates the CPU and memory state. Vulnerability modeling stage establishes the vulnerability model by summarizing existing vulnerability characteristics and then generates the vulnerability checking strategy. Offline analysis stage loads the trace file of program execution and taint source log file to perform offline analysis on the basis of taint propagation policy and vulnerability checking strategy. Additionally, offline analysis stage builds taint propagation flow graph and generates vulnerability log file. Finally, backtrace analysis stage takes vulnerability log file and taint propagation flow graph as input to backtrack taint data to locate vulnerability.

3.2. Dynamic Information Acquisition. This stage is mainly to extract the runtime information of target program and mark the taint source. Moreover, the generated trace file of program execution and taint source file are used for offline analysis. The key processes of this stage are the capture of program runtime information and the marking of taint sources.

3.2.1. Capturing Program Runtime Information. The capture of program runtime information is mainly to obtain four aspects of information, namely, the information of CPU running state, memory information, process information, and the trace file of program execution.

(1) Acquiring the Information of CPU Running State. The indispensable CPU state information mainly includes general register, instruction register, flag register, segment register, control register, and the internal variable of QEMU. This information can be obtained by capturing the current CPU pointer.

(2) Acquiring Memory Information. Clearly, there are four types of memory addresses in the QEMU, namely, client virtual address, client physical address, host virtual address, and host physical address.

Reading and writing the QEMU memory should actually be working in the physical address of host. Therefore, the key point is performing the address translation from client address to host address to acquire current memory information.

(3) Acquiring Process Information. The processes store much program information, and offline analysis is accompanied by the virtual replaying of program which requires a large number of data types and semantic information. Therefore, the acquisition of process information is the core during the capture of program runtime information.

In the Windows operating system, data structures associated with the process include five structures: KPROCESS, EPROCESS, KTHREAD, KPRCB, and KAPC_STATE. That almost describes all the information about processes and threads. To obtain current process, our method first needs the memory address of KPRCB, which can be acquired by the reverse analysis of the Windows kernel. On the basis of KPRCB, our approach can further calculate KTHREAD, KAPC_STATE, KPROCESS, and EPROCESS, respectively, to acquire process information.

(4) Generating the Trace File of Program Execution. Actually, the generating process of trace file is to integrate captured program runtime information in accordance with the organization and management way of kernel data structure. The trace file mainly includes obtained CPU running state

information, memory information, and process information. The generated trace file of program execution records instruction running number, current process number, thread number, instruction, operand, register, and other information. Specifically, it can be divided into the following five steps:

(a) Through CPU running state information to obtain instructions, registers, and other information.
(b) Taking instruction information and relevant register information as input, the instruction opcode and operand can be obtained through memory information.
(c) Obtaining process-related information by acquired register information, memory information, and process information.
(d) The acquired information is integrated into the trace file of program execution, including register information, instruction opcode, instruction operand, and process information.
(e) Generate the trace file of program execution.

3.2.2. Marking Taint Sources. Taint markings are prerequisite for taint propagation, and the way of taint markings and taint operation recordings has a great influence on the efficiency of dynamic taint analysis. In the dynamic taint analysis, the external data, such as file input or network input, are generally marked as taint data. In order to mark taint source, the corresponding system services that are responsible for reading external data need to be monitored. In the Windows operating system, some system services, such as *NtReadFile*, *NtCreateFile*, *NtWriteFile*, *NtClose*, *WSARecv*, and *recv*, are responsible for file input and network input. Thus, in this paper, OFFDTAN marks taint sources of network input and file input by coding the Hook function for above system services to monitor the parameters and return value of those system services. As a result, our method obtains and records the offset of taint data in the taint source file, the length of taint data, and the head address of memory where taint data is stored.

3.3. Vulnerability Modeling. This paper focuses on the research of stack buffer overflow vulnerabilities and controlled jump vulnerabilities, so corresponding models have been established according to the characteristic of offline analysis.

3.3.1. Stack Buffer Overflow Vulnerabilities Model. Since the *strcpy*, *memcpy*, and other string manipulation functions are optimized in the process of compiling, this paper summarizes the assembly code of string library functions to analyze the program path that could trigger buffer overflows. Moreover, this paper establishes the dependency among the execution paths and combines offline analysis with modeling buffer overflow vulnerabilities.

The stack buffer overflow vulnerabilities model of this paper mainly concerns whether the operand of instruction *rep movsd* can override function return address or *EBP* to further affect the control flow of program.

In order to better represent this model, firstly, we explain various terms used in below definition. The symbol *EDI* represents destination address operated by crucial instruction. The symbols *EBP* and *ESP* denote stack base address and stack top pointer, respectively, and symbol *RA* represents function return address. The program counter register is denoted as *ECX*. Then the following definitions are given.

Definition 1. The symbol I denotes whether there is a crucial instruction, $I \in \{0, 1\}$. The value of I is 1 if and only if instruction *rep movsd* exists in the trace of program execution.

Definition 2. The symbol F denotes whether the instruction I writes on stack memory, $F \in \{0, 1\}$. The value of F is 1 if and only if $ESP < EDI < EBP$ (or $ESP < EDI < RA$), which can determine that instruction I is operating the stack memory.

Definition 3. The symbol P denotes the safe distance of stack buffer, $P \in [0, 2097152]$. The safe distance is defined as $P = EBP - EDI$ or $P = RA - EDI$.

Definition 4. The symbol D denotes whether EBP (or RA) is overwritten, $D \in \{0, 1\}$. In this case, the length of source data operated by instruction I is expressed as L, which is $4 * ECX$. The value of D is 1 if and only if $L > P$.

Definition 5. The symbol E represents whether ECX is tainted, $E \in \{0, 1\}$. The value of E is 1 if and only if ECX is tainted.

Definition 6. The symbol C denotes whether ECX has a comparison instruction before instruction I, $C \in \{0, 1\}$. The value of C is 1 if and only if there is a comparison instruction to ECX before instruction I.

Definition 7. The symbol T denotes whether the parameters compared with ECX are tainted, $T \in \{0, 1\}$. The value of T is 1 if and only if the parameters are tainted

Definition 8. The symbol V expresses whether the program has stack buffer overflows and V is byte type. The values of I, F, D, E, C, T are the lower six bits of V in turn. For example, the value of V is 00111010 if the values of I, F, D, E, C, T are 1, 1, 1, 0, 1, 0, respectively.

In summary, this model considers two modes of stack buffer overflows primarily.

(1) *The triggered mode of stack buffer vulnerabilities*: in this mode, $L > P$, i.e., $D = 1$; thus the data written to stack could overwrite EBP or RA. The value of V is 00111XXX.
(2) *The nontriggered mode of stack buffer vulnerabilities*: in this mode, ECX is tainted, thus triggering potential stack buffer vulnerabilities. This model is divided into two cases on the basis of the value of C. (i) When $C = 0$, the value of V is 0011010X. (ii)When $C = 1$ and $T = 1$, the value of V is 00110111.

3.3.2. Controlled Jump Vulnerabilities Model. Controlled jump usually refers to the fact that taint data is used for return address, function pointer, and destination address, etc. and makes the program hijacked to the shellcode code, which causes the program to be controlled by the attacker. First, this model locates this type of jump instruction by the sequence of instruction during the program execution. Second, according to offline analysis, the coincident conditions that triggered vulnerabilities can be concluded. For any of jump instructions, it can trigger vulnerabilities if and only if its destination address is tainted.

This model mainly considers three types of jump modes.

Mode 9 (call/jump register mode). In this mode, the operand of the instruction is only one register. If the register is tainted, this instruction operation is tainted.

Mode 10 (call/jump [register + offset] mode). In this mode, the operand adds the offset on the basis of register. If the operand or register of instruction is tainted, this instruction operation is tainted.

Mode 11 (Ret/Retf mode). This instruction is used for function return, and usually ESP will be assigned to EIP (the instruction address of CPU execution) before return. If the data pointed by ESP is tainted, this instruction operation is tainted.

3.4. Offline Dynamic Taint Analysis. Due to the logic complexity of internal system call, the traditional dynamic taint analysis has the disadvantages of lower analysis efficiency and higher runtime overhead while analyzing large-scale programs. OFFDTAN presents an approach of offline dynamic taint propagation, which combines instruction and the system call, and separates dynamic analysis from program execution. More precisely, our approach leverages the trace file of program execution and taint log file to implement the offline analysis through virtually replaying program. That ultimately improves the accuracy and efficiency of dynamic taint analysis.

3.4.1. Overview of Offline Dynamic Taint Analysis. The offline dynamic taint analysis can be mainly divided into three key points.

(1) *Program virtual replaying*: by loading instructions and contexts recorded by the trace file of program execution, it is possible to simulate program execution according to the order of instruction.

(2) *Dynamic taint analysis*: by loading taint source log file, it performs taint analysis during the virtual replaying. This analysis is guided by propagation policy.

(3) *Program vulnerability analysis*: it checks the program vulnerabilities in the process of dynamic taint analysis, which is guided by security policy.

The (1) is mainly to simulate CPU's running process, and in this context, the semantics of different instructions are parsed to implement program virtual execution. Specifically, OFFDTAN first loads the trace file of program execution, reads line by line, and stores the necessary information such as instruction and register information. With the help of *udis86*, a third-party disassembly engine, read instruction information then needs to be disassembled. At the same time, some relevant program runtime information will be copied to the simulated CPU. Finally, we perform semantic and syntactic analysis for each instruction within the context of simulated CPU.

Since (3) is described in Section 3.3, this section will focus on dynamic taint analysis, which mainly includes taint data recording and taint propagation. The former records the taint state of memory and register during the taint propagation. The latter primarily provides proper propagation policy for taint analysis. According to the taint data recording and corresponding taint propagation, this stage ultimately generates the taint propagation flow graph, which provides data input for backtrack analysis and forward analysis.

3.4.2. Taint Data Recording. Taint data recording primarily includes the introduction of taint source and the state update of taint data. The introduction of taint source is the first step in taint analysis. Unlike traditional dynamic taint analysis, offline analysis introduces taint source by analyzing taint source log file. Specifically, according to the instruction number of virtual replaying, the taint source information with the same number would be loaded into the taint state space. Next, this method uses shadow memory to store and maintain taint state of memory address and register in taint state space. It updates taint state in real time based on taint propagation policy during the dynamic taint analysis.

Taint state space records the taint state of memory address and register, as shown in Figure 2. However, most of previous approaches have not recorded taint operation; thus taint propagation path cannot be stored. In order to support the backtrace analysis of taint data, taint operation needs to be recorded according to different storage carrier.

(1) *Taint operation recording for memory*: the tainted memory of each byte is stored in the form of data structure *TM*, which is arrayed in a form of linked list based on the order of taint operation when taint is propagated. Data structure *TM* mainly contains memory address, taint state, memory address of taint source, and pointer to the node of taint propagation flow graph, as shown in the upper part of Figure 2.

(2) *Taint operation recording for register*: by learning from the value of register enumerated variable *ud_type*, which is the third-party disassembler *udis86*, serialized storage management would be taken for the taint state information of register. In the *udis86*, the value of register enumerated has 141 types; thus this paper defines the structure array *TRArray*[141] of *TR* to store the taint state of register, where array index corresponds to the specific register. The data structure *TR* mainly contains taint state, pointer to the node of taint propagation flow graph, and taint source operation, as shown in the lower part of Figure 2.

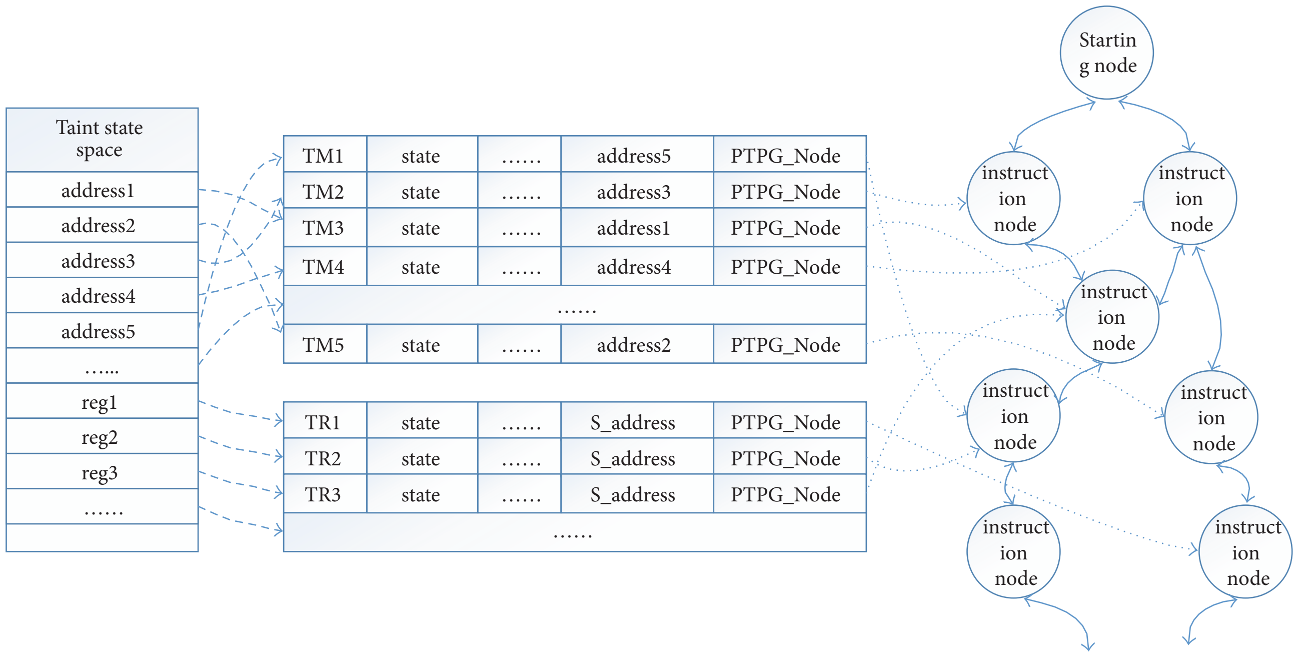

Taint state space
address1
address2
address3
address4
address5
......
reg1
reg2
reg3
......
TM1 state ...... address5 PTPG_Node
TM2 state ...... address3 PTPG_Node
TM3 state ...... address1 PTPG_Node
TM4 state ...... address4 PTPG_Node
......
TM5 state ...... address2 PTPG_Node
TR1 state ...... S_address PTPG_Node
TR2 state ...... S_address PTPG_Node
TR3 state ...... S_address PTPG_Node
......
Starting node
instruction node

FIGURE 2: Taint state space and corresponding taint propagation flow graph.

3.4.3. Taint Propagation Policy. Taint propagation is the key to taint analysis. Furthermore, a complete and proper propagation policy, which describes how taint data should be propagated during program replaying, is crucial for taint propagation. In order to develop taint propagation policy, we first extract and analyze common instruction set in the complex instruction space and instruction semantics through many program tests. Then we study six categories of instruction, such as data transfer class, operation class, shifting instruction class, string operation class, and control transfer class. According to its instruction semantics, source operands, destination operands, and addressing mode, taint propagation policy is developed. Finally, we set query interface, tainting interface, and sanitization interface, which is used for updating the taint state in real time.

A corresponding taint propagation policy is made for above six instruction categories.

(1) Data transfer class: *MOV*, *PUSH*, *XCHG*, *etc*.

(i) If one of the source operands is tainted, the destination operands will be tainted.

(ii) If source operands are untainted, the destination operands need to be sanitized.

(2) Operation class: *ADD*, *ADC*, *AND*, *XOR*, *etc*.

(i) If any of the operands of an instruction is tainted, this result of operation is tainted and destination operands are also marked as tainted.

(ii) If the conditional flag of processor is affected by tainted operand, the affected conditional flag is also marked as taint data.

(iii) For instruction *XOR*, if the source operand and destination operand have identical register or memory address, the operand of instruction needs to be sanitized.

(3) Shifting instructions: *SAL*, *SHL*, *SHLD*, *RCL*, *etc*.

(i) If the arbitrary operands of this instruction are tainted, this result of operation is tainted and destination operands are also marked as taint data.

(ii) If the conditional flag of the processor is affected by tainted operand during the shifting, the affected conditional flag is also tainted.

(4) String operation class: *MOVS*, *LOAD*, *STOS*, *etc*.

(i) If the source operands of this instruction are tainted, the destination operands are tainted.

(ii) If an instruction with *REP* exists, the *ECX* needs to be sanitized after this instruction has been executed.

(5) Control transfer class: *JMP*, *CALL*, *RET*, *etc*.

(i) If these instruction operands are tainted, the *EIP* is tainted.

(ii) If instruction operands are indirect addressing, the *EIP* is marked as taint data as long as the internal register of operand is tainted.

(6) Others: *CLD*, *CLC*, *STC*, *CBW*, *etc*.

(i) For these instructions of clear direction flag register, such as *CLD* and *CLC*, the corresponding flag registers need to be sanitized.

(ii) The instruction *STC* does not need to be processed.

(iii) For extended instructions such as *CBW*, the extended high register is marked as taint data if the low register is tainted.

In summary, our propagation policy focuses on improving the update ways for the taint state of flag register and associated register. For example, when the taint state of accumulator *AX* changes, it is necessary to modify the taint state of associated registers, such as the *AL* register, *AH* register, and *EAX* register.

3.4.4. Constructing Taint Propagation Flow Graph. Dynamic taint analysis is the fundamental of the constructing of taint propagation flow graph. During the offline dynamic taint analysis, taint operation would be recorded to provide the data dependency between instruction operands and function parameters for building taint propagation flow graph. More precisely, OFFDTAN first loads the trace file of program execution and analyzes those instructions during the program virtual replaying. According to the propagation policy and the taint state space of operand, OFFDTAN then determines the effect of each instruction on the taint state space. If there is a change in the taint state space, OFFDTAN will record this trace of taint operation and point to the corresponding nodes in the taint propagation flow graph.

Dynamic taint analysis can achieve the forward analysis and backtrace analysis to taint data by using taint propagation flow graph. The traditional taint propagation flow graph is composed of instruction nodes and directed edges. In this method, the taint state recording structure can reflect the distribution of taint data in the taint state space. In order to locate taint source, our method employs a bidirectional edge as connection edge and adds a serial number for each node in the taint propagation flow graph to distinguish the upper and lower relations between the nodes.

In a word, a taint propagation flow graph is composed of instruction nodes and bidirectional edges, as shown in Figure 2, which also demonstrates the relationship between taint state space and the node of taint propagation flow graph.

The node, which is composed a triple, has two types: the starting node and the intermediate node. Taint source is taken as starting node, and intermediate node is established by determining whether the direct or indirect operand of relevant operation instruction is tainted. If it is, a node is generated for this instruction. In particular, a pointer of the taint state recording structure corresponding to tainted operand points to the node of taint propagation flow graph. Then intermediate nodes would be created sequentially until the end of program execution.

The edges are used for connecting these nodes according to the dependencies between the operands of instruction nodes. Based on the real-time mapping relationship between the recording structure of taint state and the corresponding node of taint propagation flow graph, our approach can find the corresponding node from the recording structure of taint state and then connect the current node and the queried node through bidirectional edge.

3.5. Backtrace Analysis for Taint Data. The backtrace analysis of taint data can be used for locating taint source. In addition, the backtrace analysis has a significant effect on improving vulnerability analysis and assisting manual analysis.

This paper first takes the taint data in the recorded program vulnerabilities as the source of backtrace analysis. According to its address, the corresponding node of taint propagation flow graph can be found from taint state space. Then, according to the taint source information of this node, the prior node of taint propagation flow graph can be further inquired from taint state space. Finally, our approach successively backtracks taint propagation flow graph until it located to the source of taint information. In a word, the process of backtrace analysis is to continuously find a feasible path from program vulnerabilities to taint source.

Backtrace analysis could be formally described as follows. Firstly, taint source information can be defined as S, which is the n-tuple of all the set of taint source information and is denoted as $S = (s_1, s_2, \ldots, s_n)$. We assume that any of the propagation paths can be represented by the elements in the n-tuple $V = (v_1, v_2, \ldots, v_n)$, where v_i ($i \in [0, n]$) is the node of taint propagation flow graph and n is the number of nodes. The data structure of v_i is $\langle src_i, ins_i, dst_i \rangle$, where src_i represents its source operand, ins_i is the current operation instruction, and dst_i refers to the destination operand. The goal of backtrace analysis is to obtain the src_i in the taint operation v_i of program vulnerabilities. Then propagation path i-tuple $V = (v_1, v_2, \ldots, v_i)$ will be traversed reversely, so that the src_i in v_i belongs to any elements of S; i.e., $v_i\text{-} > src_i \in S$.

4. Evaluation

This section describes the verification process of proposed approach. We first analyze the implementation step by step through a small case and illustrate the correctness and feasibility of our method. Then, we apply it to six large applications, such as FeiQ 2.5 and Word 2010, to further verify the correctness and effectiveness of proposed approach. Finally, we set a comparative experiment to evaluate OFFDTAN's performance.

4.1. Experimental Setup. Our approach is implemented on QEMU 1.2.0 that supports KVM acceleration. Host hardware is Dell 8900 with Intel Core i7-4770 processor and 32 GB memory, and host operating system is 64-bit CentOS 7. Client hardware is the x86 architecture simulated by QEMU with a virtual CPU and 512 M memory, and client operating system is 32-bit Windows 7.

4.2. Case Study

4.2.1. Case Design. In this case analysis, we manually construct *C* program with vulnerability, which is *test.cpp*, and then generate *test.exe* by compiling it. Algorithm 1 is the

```
void myMemcpy(char si[], int count){
     char dest[10];
     memcpy(dest, si, count);                                   //building program
}                                                               //vulnerability point
int main(int argc, char *argv[]){
     HANDLE hOpenFile = (HANDLE)CreateFile(argv[1],             //reading taint
                                GENERIC_READ,                   //source file test.txt
                                FILE_SHARE_READ, NULL,
                                OPEN_EXISTING, NULL, NULL);
......
     count = readCount(buf);                                    //reading the count
     newBuf = readNewBuf(buf);                                  //reading the string
     myMemcpy(newBuf, count);
     return 0
}
```

Algorithm 1: Source code of *test.cpp*.

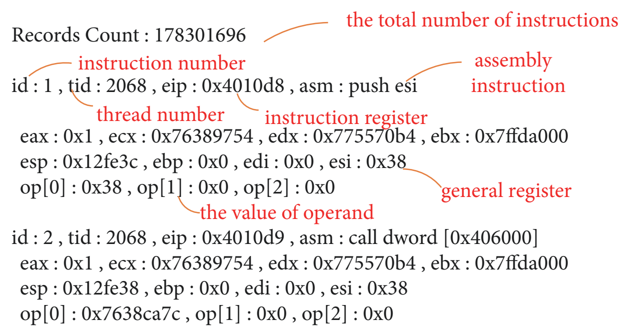
source file of target program *test.exe*. In this program, the *test.txt* is taken as input file, which is taint source file, and its contents are "16aaaaaaaa. . .aaaa". From the *test.cpp* and *test.txt*, we can draw a conclusion that the test program reads "16" in taint source file as the value of *count* in the $memcpy(dest, si, count)$, and reads "aaaaaaaa. . .aaaa" as the value of string *newBuf*. Moreover, *count* is the counter that tracks how many times function *memcpy* writes data to destination address *dest*, where *dest* is an array whose size is 10. Therefore, this case must cause an overflows.

We first use debugging tool *Ollydbg* to analyze *test.exe* in the local Windows 7 operating system, and then the address of stack buffer can be obtained, which is *0x401046*. At this time, becuase the value of *ECX* is too large, the function return address *0x12FE30* is covered by taint data when the *rep movsd* is executed. The taint data that covers the function return address is "aaaa".

4.2.2. Case Analysis. This part will analyze this case in detail in accordance with the implementation process of OFFDTAN.

(1) Capturing Program Runtime Information. This case executes target program *test.exe* to acquire the program dynamic information. Specifically, the data structure TraceNode is defined to integrate the CPU, memory, and process information. The trace file of program execution *record.log* ultimately will be generated. Figure 3 demonstrates the information contained in the trace file of program execution, mainly including instruction running number, thread number, instruction register, assembly instruction, and general register. These will be used for program virtual replaying.

(2) Recording Taint Sources. This method writes Hook function for some services that is related to file input and network input (such as *NtCreateFile*, *NtReadFile*, and *recv*) to mark taint source. At last, the taint source log file *taintsource.log* will be generated. We take a taint source information as an example to illustrate the information contained in the taint source log file, as shown in Table 1. This information will



```
Records Count : 178301696
id : 1 , tid : 2068 , eip : 0x4010d8 , asm : push esi
  eax : 0x1 , ecx : 0x76389754 , edx : 0x775570b4 , ebx : 0x7ffda000
  esp : 0x12fe3c , ebp : 0x0 , edi : 0x0 , esi : 0x38
  op[0] : 0x38 , op[1] : 0x0 , op[2] : 0x0
id : 2 , tid : 2068 , eip : 0x4010d9 , asm : call dword [0x406000]
  eax : 0x1 , ecx : 0x76389754 , edx : 0x775570b4 , ebx : 0x7ffda000
  esp : 0x12fe38 , ebp : 0x0 , edi : 0x0 , esi : 0x38
  op[0] : 0x7638ca7c , op[1] : 0x0 , op[2] : 0x0
```

Figure 3: The trace file of program execution.

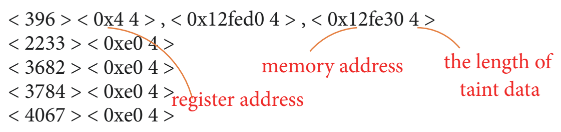
Table 1: The content of taint source log file.

Name	Content
No.	1
The head address of taint data in memory	0X12FE54
The length of taint data	200
The offset of taint data in taint source file	0

be used for marking taint source in the process of offline dynamic taint analysis.

(3) Offline Dynamic Taint Analysis. This step takes the taint source log file *taintsource.log* and the trace file of program execution *record.log* as input to perform offline dynamic taint analysis in accordance with the taint propagation policy and program vulnerability checking strategy. It will ultimately generate program vulnerability log file *sink.log* (Figure 4(a)) and corresponding taint propagation flow graph (Figure 4(b)).

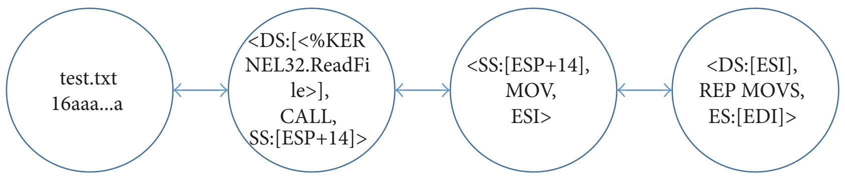
As can be seen from Figure 4(a), our method detects five program vulnerabilities. The information of each vulnerability includes instruction number, the memory or register address of taint data, and the length of taint data. All these will be used for the backtrace analysis of taint data.



```
< 396 > < 0x4 4 > , < 0x12fed0 4 > , < 0x12fe30 4 >
< 2233 > < 0xe0 4 >
< 3682 > < 0xe0 4 >
< 3784 > < 0xe0 4 >
< 4067 > < 0xe0 4 >
```

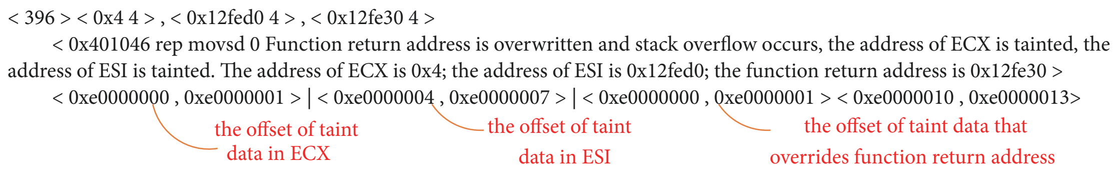
memory address
the length of taint data
register address

(a) Program vulnerability log file


test.txt 16aaa...a
<DS:[<%KERNEL32.ReadFile>], CALL, SS:[ESP+14]>
<SS:[ESP+14], MOV, ESI>
<DS:[ESI], REP MOVS, ES:[EDI]>

(b) An example of taint propagation path

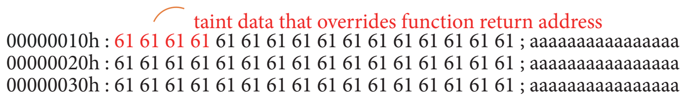
Figure 4: The results of offline dynamic taint analysis.



```
< 396 > < 0x4 4 > , < 0x12fed0 4 > , < 0x12fe30 4 >
    < 0x401046 rep movsd 0 Function return address is overwritten and stack overflow occurs, the address of ECX is tainted, the
address of ESI is tainted. The address of ECX is 0x4; the address of ESI is 0x12fed0; the function return address is 0x12fe30 >
    < 0xe0000000 , 0xe0000001 > | < 0xe0000004 , 0xe0000007 > | < 0xe0000000 , 0xe0000001 > < 0xe0000010 , 0xe0000013>
```

the offset of taint data in ECX
the offset of taint data in ESI
the offset of taint data that overrides function return address

Figure 5: Offline analysis log file.



```
00000010h : 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 ; aaaaaaaaaaaaaaaa
00000020h : 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 ; aaaaaaaaaaaaaaaa
00000030h : 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 ; aaaaaaaaaaaaaaaa
```

taint data that overrides function return address

Figure 6: Taint source file.

(4) Backtrace Analysis. This step uses program vulnerability log file as input and then backtracks taint data according to the corresponding relationship between taint propagation flow graph and taint state space. In Figure 4(b), we take the first program vulnerability as an example to show the corresponding taint propagation path V and illustrate how to backtrack taint source. Each intermediate node has three elements, namely, src_i, ins_i, and dst_i. We start with the src_i of last node, which is $DS : [ESI]$, and inquire it from taint state space to get the prior node in which the $dst_{i.}$ is ESI. Then, the src_i of obtained node is inquired, which is $SS : [ESP + 14]$, to acquire the prior node related to $SS : [ESP + 14]$. Next, we successively traverse acquired node to inquire its src_i until it belongs to taint source S, and $DS : [< \%KERNEL32.ReadFile >]$ is the entry address of function *ReadFile*. The memory block $SS : [ESP + 14]$ is the buffer address read by function *ReadFile*, and this memory block stores the string loaded from taint source file *test.txt*. This step will ultimately generate the offline analysis log file *result.log*, as shown in Figure 5.

As can be seen from Figure 5, each offline analysis log is corresponding to the program vulnerability log file, indicating that our method backtracks each of program vulnerabilities. Taking the first record in Figure 5 as an example, the address *0x12FE30*, which is detected by proposed method, is covered by taint data when the instruction *rep movs* is executed. This is consistent with the result of *Ollydbg* analysis. In addition, the result of backtrace analysis is that the offset of taint data within taint source file, which covers return address, is $0x10 \sim 0x13$. The taint source file (see Figure 6) shows that the "aaaa" covers return address, which also coincides with the analysis in Figure 5.

According to above case analysis, we elaborate the analysis procedure of our approach and verify the correctness and feasibility of this method as well.

4.3. Off-The-Shelf Applications. To evaluate our method's ability to detect vulnerabilities for real applications, we take FeiQ and Microsoft Office Word as an example and analyze the vulnerability information of these two projects in detail.

4.3.1. Experimental Objective. This experiment is mainly to verify the correctness and effectiveness of taint source marking, the vulnerability model, and the final experimental results of this method.

4.3.2. Experimental Design. The design of this experiment is to use realistic application as test object. Through analyzing application program, the correctness and effectiveness of proposed method can be verified.

Two programs have been selected first. FeiQ 2.5 is a network communication program and widely used in the enterprise. Microsoft Office Word 2010 is a word processing program, which is the mainstream of current word processing software.

```
< 50487 > < 0x4 4 >
 < 0x49d04e rep movsd 0 There may be a stack overflow, the address of ECX is
tainted, the address of ECX is 0x4 >
 < 0xe0000026 , 0xe000002f >   the offset of taint data in
                                   taint source file
```

(a) Offline analysis log file

```
00000000h : 31 5F 6C 62 74 34 5F 31 23 36 35 36 36 34 23 36 ; 1_1bt4_1#65664#6
00000010h : 43 46 30 34 39 38 37 43 43 31 41 23 35 37 30 23 ; CF04987CC1A#570#
00000020h : 33 31 37 34 31 23 34 32 39 34 39 36 37 32 39 35 ; 31741#4294967295
```

(b) UDP message

Figure 7: The analysis results of FeiQ.

```
<1 0X2C1E000 4096 0>
<213263 0X2C1D000 4096 1000>  read the taint data
<283973 0X2C1C000 4096 2000>    in CVE-2014-
<63203752 0X2C1A000 4096 3000>  1761 POC file
```

(a) Taint source log file

```
< 620923169 > < 0xe0 4 >        program crash address
 < 0x66e9195d call dword [ eax+0x4] 0 the address is tainted [0x275a48e8] >
 < 0xe0001d3f , 0xe0001d3f > < 0xe0001d4a , 0xe0001d4a >
 < 0xe0001d54 , 0xe0001d54 > < 0xe0001e18 , 0xe0001e1a >   the offset of taint data
                                                              in taint source file
```

(b) Offline analysis log file

Figure 8: The analysis results of Word 2010.

Next, in order to perform experiment, taint sources need to be introduced. In this paper, two different ways are adopted, respectively. For FeiQ, this paper writes the Python script and sends UDP packets to the client on QEMU to introduce network taint source. More precisely, the 2425 port of the client on QEMU is employed to send UDP packets "1_lbt4_1#65664#6CF04987CC1A#570#31741#4294967295#2.5a: 1317316152: admin:XXCCLI-A10D5C26:288:AAAAAAAAA-AAAAAAAAAAAAAAAAAAAAAAAAAAAAA". Whereas for Word 2010, this paper uses the POC file of *CVE-2014-1761*, which is provided by CVE, as input to introduce file taint source.

Table 2: The content of taint source log file for FeiQ.

Name	Content
No.	1
The head address of taint data in memory	0X11F1C4
The length of taint data	129
The offset of taint data in taint source file	0

4.3.3. Results and Discussion for FeiQ. In order to verify this method, we first use debugging tool *Ollydbg* to analyze the results of FeiQ. Then, the experimental results of proposed method are given. By comparing two results, it is possible to verify that our method is correct and effective.

By the analysis of *Ollydbg*, we obtain the crash address, which is *0x49D04E*. The cause of program crash is that the value of *ECX* is too large, resulting in stack overflows when the instruction *rep movsd* is executed. In addition, *ECX* is assigned to *EAX* at address *0x49D047*, and the tainted value is "4294967295" in the UDP message, which can be converted to hexadecimal *0xFFFFFFFF*.

In our approach, OFFDTAN can generate the trace file of program execution, taint source log file, program vulnerability log file, and offline analysis log file to verify experiment objective. In this section, we only focus on taint source log file (Table 2) and offline analysis log file (Figure 7(a)). It can be seen from Table 2 that this method successfully marks the UDP message, and the length of taint data is 129, which is consistent with the length of UDP message. It can be seen from Figure 7(a) that the address of program vulnerability is *0x49D04E*, which corresponds to the assembly instruction *rep movsd*. In addition, OFFDTAN analyzes the cause of stack overflows, which is that *ECX* is tainted. Furthermore, this method uses backtrace analysis to analyze taint data in the *ECX* derived from the UDP message whose offset is from *0x26* to *0x2F*. The specific characters of UDP message is listed in Figure 7(b). The characters, from *0x26* to *0x2F*, are "4294967295", and their hexadecimal form is *0xFFFFFFFF*, which is consistent with the results of *Ollydbg*. Therefore, we verify the correctness and effectiveness of marking taint source and stack buffer overflows model proposed by this method.

4.3.4. Results and Discussion for Microsoft Office Word 2010. To verify this approach, we first analyze the vulnerability analysis processes given by CVE. Then, the experimental results of proposed method are given.

By the analysis of CVE, we find that the crash address of program is *0x66E9195D*. The reason for vulnerability is that the data in the memory corresponding to the operand [$EAX +$ 4] of *call* instruction is tainted, leading to a controlled jump when the instruction *call dword*[$EAX + 4$] is executed. The jump address is [*0x275A48E8*].

This experiment uses the POC file of *CVE-2014-1761* to analyze Microsoft Office Word 2010 and generates the taint source log file (as shown in Figure 8(a)) and offline analysis log file (as shown in Figure 8(b)) to verify experiment objective.

Table 3: Program description and evaluation results.

Program	Version	Vulnerability	Crash Address	Crash Instruction	Offset of Taint
Adobe Reader	9.3.4	CVE-2010-2883	0x0803DDAB	call strcat	0x12C
Microsoft Office Excel	2003	CVE-2011-0104	0x300DE834	rep movs	0x300
Firefox	3.6.16	CVE-2011-0073	0x1046659B	call dword ptr [ECX + 70h]	0x4A
Microsoft Office Word	2003	CVE-2012-0158	0x275C8A0A	rep movs	0xA0F

Table 4: The overhead of OFFDTAN and PANDA when running FeiQ and Word.

Applications	PANDA				OFFDTAN			
	Record Time (sec.)	Replay Time (sec.)	CPU%	Mem%	Record Time (sec.)	Replay Time (sec.)	CPU%	Mem%
FeiQ	53.67	86.09	17.4%	35.3%	47.04	69.58	12.4%	34.5%
Word 2010	74.02	127.95	18.3%	36.2%	65.92	94.81	14.6%	34.9%

As is depicted in Figure 8(a), OFFDTAN successfully marks the POC file read by Microsoft Office Word 2010. In Figure 8(b), OFFDTAN analyzes the vulnerability address of Microsoft Office Word 2010, which is *0x66E9195D*, and the corresponding assembly instructions is *call dword*$[EAX + 4]$. It also found that the reason of program controlled jump is that the memory address of [*0x275A48E8*] is tainted, which is consistent with the analysis result provided by CVE previously. In addition, this method uses backtrace analysis to analyze the taint data in [*0x275A48E8*] derived from the POC file of *CVE-2014-1761* whose offset is *0x1D3F*, *0x1D4A*, *0x1D54*, and from *0x1E18* to *0x1E1A*. Therefore, we verify the correctness and effectiveness of marking taint source and controlled jump model proposed by OFFDTAN.

4.4. Further Verification.

We proceed to evaluate our approach on four other applications to further verify the correctness and effectiveness of OFFDTAN. In addition, we compared the performance overhead of this method with other tools to verify the analysis efficiency of proposed method.

4.4.1. Effectiveness.

We analyze the security vulnerabilities of four programs: Adobe Reader, Excel 2003, Firefox, and Word 2003. For each of these projects, our method generates the valid description of vulnerability analysis. Table 3 provides an overview of these results, showing the program and its version, crash address, crash instruction, and the offset of taint source. Moreover, the vulnerabilities are shown and denoted by their CVE-identifiers.

In Table 3, the crash address and crash instruction are the memory address and corresponding instruction that crashed during program execution, respectively. The offset of taint refers to the offset address of taint data in taint source file.

We take the first vulnerability as an example to briefly illustrate analysis result. We use the POC file of *CVE-2010-2883* to analyze Adobe Reader and generate taint source log file. The result shows that the program crashed when the instruction *call strcat* at *0x0803DDAB* is called, and the reason for crash is that the string length of field *uniqueName* is not judged, causing stack overflows, which is consistent with the result of CVE. Moreover, our method uses backtrace analysis to obtain the offset of taint data in taint source file, which is *0x12C*.

4.4.2. Performance.

In this experiment, we describe the performance overheads introduced in two real applications by our approach to taint analysis and compare them with PANDA, which is another state-of-the-art whole-system offline taint analysis tool built on QEMU 2.1.0. Similar to OFFDTAN, PANDA also has the ability to record and replay executions. We evaluate both tools on four items that represent time and space overheads.

The comparison results of performance overheads are shown in Table 4. The record time refers to the time of information recording during program execution, and replay time includes program simulation replay time and taint analysis time. The CPU% and Mem% mean the percentage of CPU and memory used during program execution, respectively.

As shown in Table 4, in addition to memory usage, our tool has improved significantly in terms of other performance. OFFDTAN is about 1.13x faster than PANDA during the recording, and its offline analysis is faster than PANDA with a factor of 1.29. Overall, OFFDTAN operates about 1.23x faster than PANDA. Besides, OFFDTAN's CPU usage is about 24.4% smaller than PANDA, and the memory usage is 2.9% lower than PANDA. We attribute this to the employ of KVM acceleration on QEMU.

4.5. Experimental Summary.

OFFDTAN can detect two kinds of vulnerabilities, i.e., stack buffer overflows and controlled jump, which fully validate that the two-vulnerability model combined with offline dynamic taint analysis can better achieve analysis effects.

In addition, whether with small program or large-scale program, such as FeiQ and Microsoft Office Word 2010, OFFDTAN can correctly analyze the address of program vulnerability, the cause of program crash, and the specific offset of taint data within taint source file, thus ensuring that our approach is strictly correct and effective. Compared with PANDA, our method runs about 1.23x faster than PANDA. Moreover, OFFDTAN enhances the automation of vulnerability analysis as well.

5. Conclusion

The analysis efficiency of online dynamic taint analysis is a challenging problem, which usually occupies a considerable number of resources. In this paper, we generally eliminate this limitation by using the research approach of offline dynamic taint analysis, and on this point we propose OFFDTAN, an approach of offline taint analysis for binary program, which includes four stages: dynamic information acquisition, vulnerability modeling, offline analysis, and backtrace analysis. Our evaluation shows the effectiveness and correctness of this approach, as well as better performance.

However, OFFDTAN still has some deficiencies. First, program path coverage heavily depends on test cases; thus test cases will affect the accuracy of our method. Second, our experimental verification is still not sufficient, which needs further verification. Moreover, our approach can only detect stack buffer overflow vulnerabilities and controlled jump vulnerabilities. As future work, we are confident that the similar concepts can also be applied to model other types of vulnerabilities to adapt more scenes.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key R&D Program of China (no. 2016QY07X1404).

References

[1] W. Shizhong, G. Tao, D. Guowei, Z. Puhan et al., *Software Vulnerability Analysis Technology*, The Science Publishing Company, Beijing, China, 2014.

[2] Funnywei, “Buffer Overflow Vulnerability Mining Model [Z/OL],” 2003, http://xcon.xfocus.net/XCon2003/archives/Xcon2003_funnywei.pdf.

[3] F. B. Qemu and F. B. Qemu, “A Fast and Portable Dynamic Translator,” in *Proceedings of the Atec 05: Conference on Usenix Technical Conference*, 2005.

[4] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[5] C. K. Luk, R. Cohn, R. Muth et al., “Pin: building customized program analysis tools with dynamic instrumentation,” *Programming Language Design & Implementation*, vol. 9, no. 8, pp. 190–200, 2005.

[6] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: a transparent dynamic optimization system,” *SIGPLAN Notices*, vol. 35, no. 5, pp. 1–12, 2000.

[7] J. Newsome and D. Song, “Dynamic taint analysis for automatic dedection, analysis, and signature generation of exploits on commodity software,” *Network and Distributed System Security Symposium (NDSS)*, 2005.

[8] J. Clause, W. Li, and A. Orso, “Dytan: a generic dynamic taint analysis framework,” in *Proceedings of the 2007 ACM International Symposium on Software Testing and Analysis (ISSTA ’07) and PADTAD-V Workshop*, pp. 196–206, London, UK, July 2007.

[9] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu, “LIFT: a low-overhead practical information flow tracking system for detecting security attacks,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’39)*, pp. 135–146, December 2006.

[10] M. Kang G, S. Mccamant, P. Poosankam et al., “DTA++: dynamic taint analysis with targeted control-flow,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS ’11)*, San Diego, Calif, USA, February 2011.

[11] D. Song, D. Brumley, H. Yin et al., “A new approach to computer security via binary analysis,” in *Proceedings of the 4th International Conference on Information Systems Security*, pp. 1–25, Springer-Verlag, 2010.

[12] A. Henderson, A. Prakash, L. K. Yan et al., “Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform,” in *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA ’14)*, pp. 248–258, July 2014.

[13] L. Zhiquan, *Study of Fuzzing for Implementation of Stateful Network Protocol Based on Dynamic Taint Analysis*, National University of Defense Technology, 2010.

[14] T. Wang, T. Wei, G. Gu, and W. Zou, “TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection,” in *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP ’10)*, pp. 497–512, IEEE Computer Society, May 2010.

[15] Z. Jianwei, C. Libo, and F. Tian, “Type-based dynamic taint analysis technology,” *Journal of Tsinghua University (Science and Technology)*, vol. 10, pp. 1320–1328, 2012.

[16] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, “ShadowReplica: efficient parallelization of dynamic data flow tracking,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS ’13)*, pp. 235–246, November 2013.

[17] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, “HI-CFG: Construction by binary analysis and application to attack polymorphism,” in *Computer Security—ESORICS*, pp. 164–181, Springer Berlin Heidelberg, 2013.

[18] S. Manolis, P. Groth, and H. Bos, “Decoupling provenance capture and analysis from execution,” in *Proceedings of the 7th USENIX Workshop on the Theory and Practice of Provenance (TaPP ’15)*, 2015.

[19] S. Dawei and Y. Tianwei, “A dynamic taint analysis method combined with coarse-grained and fine-grained,” *Computer Engineering*, vol. 40, no. 3, pp. 12–17, 2014.

[20] W. Fuwei, *Research on Taint Analysis-Oriented Binary Program Analysis and Vulnerability Mining*, Beijing University of Posts and Telecommunications, 2015.

[21] J.-X. Ma, Z.-J. Li, T. Zhang, D. Shen, and Z.-K. Zhang, “Taint analysis method based on offline indices of instruction trace,” *Journal of Software*, vol. 28, no. 9, pp. 2388–2401, 2017.

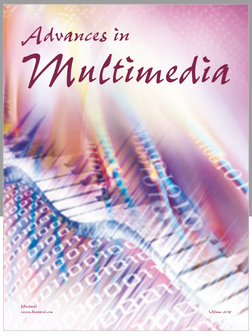
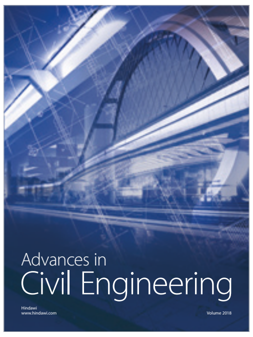
[22] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: decoupled offline symbolic taint analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*, pp. 308–319, September 2016.

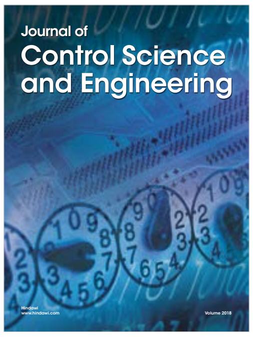
[23] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," *Computer Science*, 2014.

Journal of
Engineering

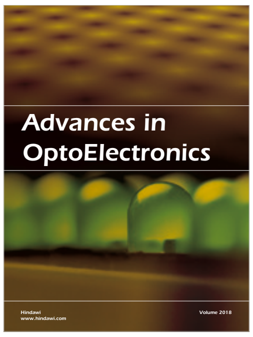
The Scientific World Journal

International Journal of
Rotating Machinery

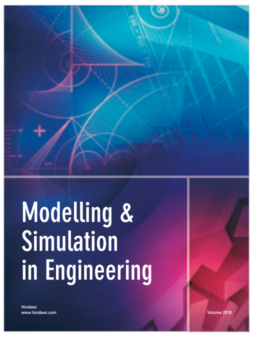
Journal of
Sensors

Advances in
Multimedia

Advances in
Civil Engineering

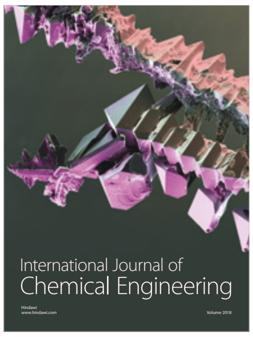
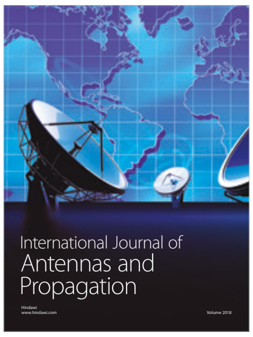

Journal of
Control Science and Engineering

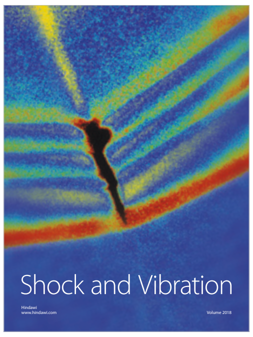
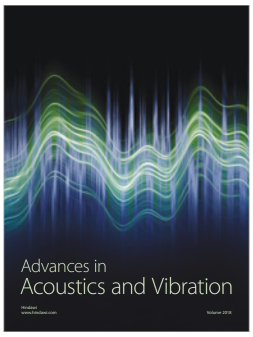
Journal of
Robotics

Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration